# Adaptive Execution in Complex Dynamic Worlds

R. James Firby

YALEU/CSD/RR #672

January 1989

This work was supported in part by Defense Advanced Research Projects Agency grant DAAA15–87–K–0001.

# Adaptive Execution in Complex Dynamic Worlds

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Robert James Firby
May 1989

# Abstract

## Adaptive Execution in Complex Dynamic Worlds

Robert James Firby

Yale University

1989

A robot acting in the real world must use flexible plans because actions will sometimes fail to produce desired effects, and unexpected events will sometimes demand the robot shift its attention. A plan is usually construed as a list of primitive robot actions to be executed one after another but in a complex domain, a plan must be structured to cope effectively with the myriad unpredictable details it will encounter during execution. However, adding structure to a plan involves more than augmenting the primitive plan representation; it requires a complete model of interaction with the world called situation-driven execution. Situation-driven execution assumes that a plan consists of tasks with three major components: a satisfaction test, a window of activity, and a set of execution methods that are appropriate in different circumstances. Execution of such a plan proceeds by selecting an unsatisfied task and choosing a method to achieve it based on the current world state. A task may be executed as many times as necessary to keep it satisfied while it is active.

This thesis proposes a plan and task representation based on program-like reactive action packages, or RAPs. A plan consists of RAP-defined tasks and each RAP generates primitive robot actions at execution time by selecting its most appropriate method. Within such a system, execution monitoring becomes an intrinsic part of the execution algorithm, and the need for separate replanning on failure disappears. RAPs are more than just programs that run at execution time, however, they are also hierarchical building blocks for plan construction. The RAP representation is structured to make a task's expected behavior evident for use in planning as well as in execution. The RAP execution system described includes a sensor memory, representation language and interpreter. Examples and experiments demonstrate a wide range of adaptive system behavior.

# Acknowledgements

Many people have contributed to my education and well-being here at Yale. My advisor, Drew McDermott, has shaped my tenure the most. In the beginning, he helped get me started on the FORBIN planning project with Tom Dean and Dave Miller. Later, when I began to think about reactive planning, he gave me the freedom and resources to pursue my ideas. Most recently, he has given me the pushes I needed to get everything finished and written down. Along the way, his intuitions helped get me unstuck many times.

I have benefitted immensely from the AI community at Yale. Roger Schank and Drew McDermott bring very different points of view to many AI problems and reconciling those views has shed more light on AI as a whole than anything else I have done. The fact that things usually reconcile so well has been most intriguing. The other faculty have joined in many fruitful discussions and I would like to thank Chris Riesbeck and Padmanabhan Anandan for reading and commenting on my thesis, and Larry Birnbaum for explaining how wrong everything is. I must also thank Ted Linden at ADS for putting up with "it's in the mail" for months and then having to read and comment on my whole thesis in a couple of weeks.

Graduate school would have been impossible without my fellow students. Dave Miller helped me to get started in research and racquet-ball and I had many discussions with Tom Dean and Yoav Shoham while they finished their dissertations. Brad Alpert, David Leake and I had many fruitful debates while we studied for the qual and Brad and I have done some great climbing together. Charles Martin, Denys Duchier, Chris Owens, and I have done terrific hacking on entirely useless projects — just the thing to keep one sane. Rick Mohr, Jim Philbin, and Richard Kelsey have endured many questions about T and many discussions and tirades about graphics. Larry Wright wrote some good graphics code and played some good squash games. I thank them all.

Steve Hanks and Alex Kass have been especially important, both as friends and as associates. Our discussions have helped get my research moving again on more than one occassion and the restaurants we have been to have taught me a lot about food. Sharing AI's coolest office with Steve has been invaluable; without his comments and ideas, I would never have gotten finished.

Finally, I must thank my wife, Joanne, for her support and encouragement through it all. She has endured four years at school that turned out to be six, and a quaint New England town that turned out to be New Haven.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

A robot acting in the real world must be able to muddle through the everyday uncertainties it will inevitably encounter. A car may not start on the first try; a door may be open or closed; a street along the usual route may be closed for repaving; which elevator will come after pushing the up button is unknown. All of these situations require a robot to adapt its behavior while in the middle of executing its plans. The robot must wait for the uncertainty to resolve itself, take stock of the situation, and choose an action appropriate at that time.

Robot planners usually make the assumption that the world can be predicted infinitely far into the future. Such an assumption is often reasonable in domains completely under the control of the planner but, in a complex, dynamic domain like the real world, an assumption of predictability is unwarranted. A domain is termed "complex" when its initial state is largely unknown and it contains items and processes that behave in ways that are only partially understood. A "dynamic" domain contains agents not under the planner's control that can change the world without warning in unpredictable ways. It is clear that the real world is both complex and dynamic, and that predicting its future in detail is not possible. Therefore, a robot control system that intends to act in the real world must not depend on such predictability.

When a planner cannot project future states of the world in complete detail, it cannot construct a detailed plan either. Many primitive actions in the plan will have to be left unspecified until the situation in which they are required arises and the details of the world state can be determined by direct observation. A plan that

includes taking an elevator cannot specify the actions required to enter until after the elevator door opens and the robot knows which elevator it will be using. Thus, a robot acting in a complex, dynamic domain must be controlled with unfinished, flexible plans that allow many different completions depending on the actual situations encountered at execution time. Choosing appropriate plan completions at execution time to cope with the unpredictable details and unavoidable obstructions of everyday life is the central topic of this thesis.

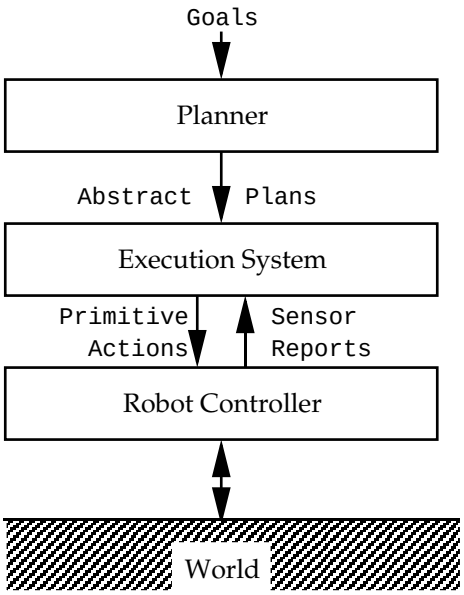



**Figure 1.1**: A Complete Robot Control System

The generation and control of robot actions can be divided into three major components as shown in Figure 1.1:

- *Planning*: deliberates as much as possible and generates sketchy, high-level sequences of tasks to carry out particular goals.
- *Execution*: takes a plan at whatever level of detail the planner can generate and expands each task into more detail based on the situations encountered.
- *Hardware Control*: translates discrete actions from the execution system into concurrent, coupled motor and sensor processes as required to control the robot hardware.

In the past, research has focused on getting the planning component to generate a plan with sufficient detail so that it could be handed directly to the hardware control component. However, in complex, dynamic domains, plans cannot be made that detailed in advance. This thesis proposes the RAP execution system to fill in plan details at run-time. The system copes with uncertain and changing situations, occasional robot failures, and interruptions caused by special problems and opportunities.

## 1.1 Coping With the Real World

Consider the coping behavior involved in starting a car. Few of us understand everything that happens under the hood of our cars, yet we are capable of starting them. We can even start unfamiliar cars in unusual situations like extreme cold or driving rain. The reason we can do this is because we just get in, put in the key and turn it. If the car starts, we're done; otherwise, we turn the key again. If the car doesn't start after a few tries, we give up. Planning these actions in detail is clearly impossible. Different cars place the key in different locations, a different number of tries might be required before the engine will catch, and in extreme cold, the engine may have to be cranked longer on each try. However, in most cases, simply getting in the car and muddling through will start it just fine. (In fact, cars are built that way on purpose so they will be tractable to naive users.)

Another typical example of coping behavior is getting a glass of water from the kitchen. In my apartment, the TV set is in the bedroom so I am usually there too. Getting a glass of water means walking out of the bedroom, through the living room, and into the kitchen. Along the way I will usually have to walk around furniture and packages placed randomly by my wife and myself. I may also have to avoid the cat which sometimes jumps out unexpectedly. Finally, in the kitchen there will be a glass to find somewhere in the cupboard and dishes to be moved out of the way in the sink before I can pour a glass of water. Again, this simple task cannot possibly be planned to more detail than "go to the kitchen", "get a glass from the cupboard", "fill it with water at the sink". Obstacles along the way and the detailed location of the glass in the cupboard cannot be predicted until they are encountered. However, avoiding the obstacles and finding the glass are simple tasks so there is no need to worry about them in advance. I can wait for the situation to arise and choose my actions then.

A further point is that execution of starting the car and getting a glass of water can both be stopped in the middle if the telephone rings. Answering the telephone is an example of a task that can come up at any time and usually takes priority over whatever is being worked on at the time. Such tasks obviously cannot be anticipated and would disrupt a plan that was carefully worked out in advance. The world is filled with interruptions and emergencies and a robot must be able to cope with them.

Coping behavior like that described above works well in everyday situations because the obstacles and interruptions encountered are generally minor and do not require actions that significantly affect the future of the robot. Furthermore, situations will usually supply enough cues so that execution of actions producing detrimental effects can be consciously avoided, even without significant look-ahead. For example, when interrupted, a robot should know that it is all right to drop a shovel but not a drinking glass. It doesn't have to infer that dropping a glass will cause it to break; it merely has to know that glasses shouldn't be dropped. It might even adopt the strategy that nothing should be dropped, and then nothing will ever get broken. The important thing is that just muddling through day-to-day situations and actions is an effective way to cope with the complexities of the real world.

### 1.1.1 Characteristics of Coping Behavior

The examples of coping behavior described above illustrate the characteristics of robot control in the real world that an execution system must capture:

1. Plans given to the robot must be "sketchy", and include many tasks to be expanded into more detailed actions as the situation demands.

2. While executing the plan, the robot will be uncertain about many of the details of its surroundings. This will sometimes lead to choosing incorrect actions. Furthermore, actions may fail to change the world in the way they were intended.

3. The robot will often be interrupted and must respond to the interruption, later returning to what it was doing before.

4. The robot can usually count on the uncertainties, obstacles and interruptions it runs into to be fairly benign. Thus, coping with situations as they come up should not lead to ruin.

One result of assumption four above is that a robot coping with an incomplete plan cannot be expected to solve complex, puzzle-like problems or generate optimal sequences of actions while carrying out a task. Such activities require looking into the future to see how every action being contemplated affects all of the rest of the plan. Searching for efficient plans to accomplish complex tasks is the province of traditional planning research, and is designed specifically to get around the shortcomings of trying to muddle through a difficult task. The assumption in this thesis is that the complex puzzles and scheduling constraints that might crop up in a plan have been sorted out in advance by a more traditional planner. This leaves the robot with many smaller, simpler problems to cope with. In fact, unless the world in which the plan will eventually be carried out is completely under the planner's control, even the most carefully thought out plans must remain incomplete with the details to be muddled through at execution time.

A further consequence of assuming that execution complications will be relatively benign is that we can ignore the problem of temporal look-ahead. Partially complete plans arise from the inherent inability to project the future of complex, dynamic domains in detail. The same difficulty in projecting the future makes it as hard to predict the effects of executing the next primitive action as it is to put them into the plan in the first place. Therefore, we make the assumption that look-ahead at execution time is unnecessary. Some form of limited look-ahead will undoubtedly be a part of a real robot controller, but since it cannot be complete, and the purpose of this research is to propose a system that will muddle through even in the worst case, a stronger assumption of no look-ahead is warranted. By no look-ahead we mean that coping behavior will include no explicit notion of trying to calculate future states of the world. However, action choice may be controlled with knowledge that incorporates "compiled-in" estimates of the future. When we lock the door at night, it is not because we project the future and estimate the chances of a burglar walking in, it is just part of our routine for going to bed. It may have become part of our routine because of such a projection, but that is not important.

We will call the process of muddling through a sketchy plan in a relatively benign environment *situation-driven execution.*

## 1.2 Situation-Driven Execution

The first step in building a model of situation-driven plan execution is to define a plan. A plan is a set of partially ordered tasks for the robot to perform. Each task may be designed to achieve a state in the world, to maintain a state in the world, or to carry out some specific action. "Clear off the table", "keep plenty of full fuel-drums at the fuel-depot", and "push the red button" are all examples of tasks that might be part of a plan. Furthermore, many different sequences of task execution may be possible, including concurrent execution of some tasks, that satisfy the temporal constraints in the plan. A sketchy plan is a plan that includes tasks which cannot be executed directly by the robot hardware controller (*i.e.*, tasks that are not primitive actions). In a complex, dynamic domain, all robot plans must be sketchy to some extent.

Execution of a sketchy plan begins with the assumption that there is a prescribed method for carrying out every task in every situation. A method is a set of actions that will accomplish the task in the given situation. Thus, execution consists of choosing a task to work on, assessing the current situation, choosing an appropriate method, and carrying it out. The assumption of known methods makes sketchy-plan execution very much like hierarchical plan-expansion except that it takes place at execution time rather than being done in advance. Shifting as much expansion as possible to execution time allows the expansions to adapt to actual situations rather than to uncertain projected situations.

### Tasks

A planned task can best be thought of as a goal to be achieved. The goal may be a state to achieve, a state to maintain, or an action to carry out. At execution time, the task is used as an index for choosing an appropriate method to achieve the goal. However, to support situation-driven execution, a task must also know when it has succeeded and failed. Since execution is expected to cope with method failures, a task cannot be considered complete just because the method chosen for it has finished. For example, a method for picking up a box may say to grasp it with the robot's gripper. If the box slips out of the gripper, or another agent wrestles it away, the grasp action will still have completed, but the task will not have been successful. When tasks evaluate their own success, they take control of their own destiny.

Therefore, a task must have two parts:

- An index to use with the current situation for selecting a method (*i.e.*, its goal)
- A test for determining when the task should be considered for execution.

The test for consideration must also include two things: a check for whether the task is satisfied in the current situation, and a time-window within which the task is relevant. The satisfaction test and time-window together tell the task selection mechanism whether a task should be considered for execution. If a task is satisfied in the current situation it need not be worked on; however, it might need to be worked on in the future if it becomes unsatisfied. If a task's time-window is past, then the task need not be considered anymore. We will call a satisfied task with a valid time-window *suspended*, an unsatisfied task with a valid time-window *active*, and a task with an expired time-window *inactive*.

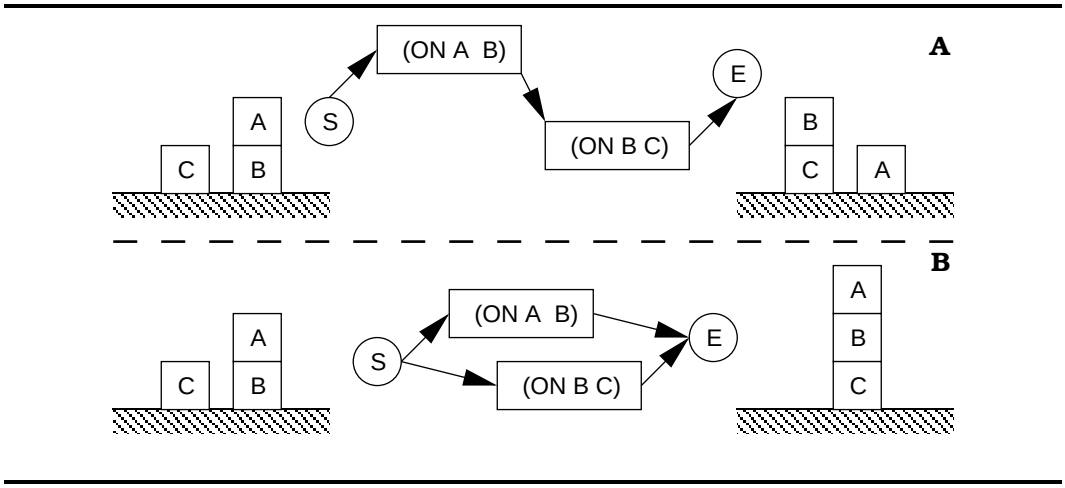



**Figure 1.2**: An Example of Task Success and Time-Window Checks

As an example, consider the two different blocks world sketchy plans shown as partial orders in Figure 1.2. Both plans include the tasks `(ACHIEVE (ON A B))` and `(ACHIEVE (ON B C))`. The first plan orders the two, and the second plan does not. We will assume that each task is satisfied as long as its goal is satisfied and not satisfied when its goal needs to be achieved. However, defining appropriate time-windows for the various tasks is a tricky matter of specifying the semantics of the plan. Let us assume that the time-windows intended extend in both directions from the task as far as the ordering-constraint arrows in the diagram. Thus, in the first plan, the first

task becomes inactive as soon as the second task starts, while in the second plan, neither task becomes inactive from start to finish. These two plans result in different behavior when executed in a situation-driven fashion. In the initial situation given in the diagram, the first task in the first plan is already satisfied so execution goes on the next, expiring the first task's time-window and eliminating it from further consideration. Then, when the second task completes, block `B` will be on block `C` but block `A` will be on the table. The second plan, however, leaves `(ACHIEVE (ON A B))` suspended and after `(ON B C)` has been achieved it will become active and place block `A` on the top of the stack. Notice that this is exactly what the plans say to do; the difference in behavior comes from the semantics of the task descriptions, as specified in the tests for when the tasks should be considered active.

### Methods

A method is a prescription for changing the world situation so that a given task becomes satisfied. There may be many different methods for the same task, each applicable in different situations. Having different methods for different situations allows task execution to adapt to changing situations.

It is assumed in this thesis that all of the methods available to the robot are known in advance and contained in a large library. New methods are not learned, or constructed from old ones, when a novel situation arises. The robot must muddle through with what it knows or give up. Robust behavior can be achieved even with a fixed set of methods as long as every task has one or more general methods for satisfying it in most situations. When a general method cannot be constructed and the robot must have a specific way of carrying out a task in every different situation, it will often be unable to complete its tasks. Starting a car is an example of a task for which a general method can be constructed and will usually work. You put the key in the ignition and turn it. On the other hand, opening a combination lock usually requires a different method for every lock. If you don't know the lock's combination you simply can't open it.

An important point about methods is that they are not guaranteed to succeed in accomplishing their corresponding tasks. For example, turning the key may not start the car, and dialing a lock's combination may not open it on the first try. Furthermore,

other agents may interfere with a method and cause it to fail even though it would have worked fine in isolation. The fact that methods cannot be guaranteed to succeed is the reason that goal satisfaction must be the responsibility of the task and not the method. Tasks select methods to satisfy their goals and test the resulting situations to see whether the goals are satisfied. Methods simply describe the ways of attempting to satisfy the goal.

### Characteristics of Situation Driven Execution

The basic situation-driven plan-execution algorithm can be summarized as follows:

1. Consider all active tasks and select one to execute.
2. Use the current situation to index into the method library and select a method for the chosen task.
3. Execute the method and return to step 1.

This algorithm allows execution of sketchy plans while coping with the uncertainties and obstructions of the real world; it is adaptable, interruptable, and able to cope with method failure.

The algorithm adapts its behavior to the situation at hand by delaying selection of a method for a task until it is ready to start executing the task. At that time it uses the situation, as well as the task, to select an appropriate method. As long as there are appropriate methods for the existing situation, appropriate actions will be executed. In fact, the system will keep trying methods until either the task is satisfied or its time-window passes and the task is no longer relevant. This repeated execution while a task remains unsatisfied, coupled with the ability to index into different methods each time the task is retried, allows for a flexible, adaptable response to method failure.

The algorithm also allows easy task interruption and resumption. If we assume that tasks can be assigned a priority, then a new high-priority task, like dealing with an emergency, can be added to the plan at any time. The priority can be used to direct execution to the task, since the execution cycle always starts by choosing a task on which to work. If high priority tasks are favored, they will jump in and be

first in line. After a high-priority task has been executed, the system must resume what it was doing. Once again, the first step in the execution algorithm is to select a task and, now that the interrupting task is out of the way, previous tasks that are still available will be considered. Furthermore, since the situation is also reassessed and methods rechosen, the fact that the interrupting task may have changed the situation considerably is not a problem.

### 1.2.1 Sensing and Assessing Situations

An important activity that comes up over and over in discussion of the situation-driven execution algorithm is that of assessing the current situation. In a real robot, assessing the situation will involve sensing operations out in the real world. These operations will take time and a commitment of resources and so must be carefully considered. For instance, if task and method selection choices were to require consideration of every detail of the the whole world state, the sensing overhead would be overwhelming. Each time that a method was to be chosen the robot would have to spend time and energy sensing every aspect of the world around it. It would spend almost all of its time sensing and very little of its time actually acting. Fortunately, only a fraction of the world state is usually relevant to any given execution choice. The problem is deciding which fraction, and how it should be sensed. Evaluating sensing strategies automatically and separately from the normal flow of task execution may be quite difficult.

To overcome these problems, we discard the idea of independently choosing sensing operations and insist that all sensing be incorporated into task methods. This places sensing tasks and strategies in exactly the same position as effector tasks and strategies. Many task methods will consist of actions designed to acquire more sensory data, permitting a better method choice to be made later. In addition, almost every method will include sensing tasks to gather the information required to judge its own effectiveness. For example, the last action in the method for picking up an object will be to sense the gripper and see whether it is really holding the object. This information will then be used by the task to judge whether or it is satisfied.

The inclusion of sensing tasks in methods, along with the assumption that sensing tasks require time and other robot resources, requires that there be a sensor memory.

This memory will record sensory data and keep track of it while the robot is either making decisions or sensing something else. The memory then becomes a representation of the current situation that is updated by sensory operations. Thus, methods include sensor actions to insure that the relevant parts of the memory's situation representation remain up-to-date. This use of memory as a sensory information buffer, and the use of sensory tasks like any other task, is an interesting result of limited sensory capacity coupled with the need to use the current situation as a basis for execution-time decisions.

### 1.2.2 An Example

As an example of situation-driven execution, consider a sketchy plan that contains the task "pick up a fuel-drum". To perform this task, we assume our robot must move its arm to a fuel-drum, grasp it, and lift it up. Therefore, a method for this task must contain the steps: "move arm to drum", "grasp", and "lift". We will also assume that the step "move arm to drum" must be handed the detailed position of a drum, requiring that the location of the drum already be known in memory. The robot may or may not know the locations of nearby fuel-drums already; that knowledge depends on what sensing operations have been executed in service of previous tasks. To account for this potential lack of knowledge, we add a sensory operation at the beginning of the method to look for nearby fuel-drums. Thus, we have the method: "scan for drum", "move arm to drum", "grasp", and "lift". Now, the method is applicable in many situations because the built in sensing operation will update the sensor memory with the latest fuel-drum locations just before the "move" operation takes place.

The method, "scan", "move", "grasp", "lift", is applicable in many different situations, but there remains the problem of assessing its success after it completes. The original task, "pick up a fuel-drum", will want to check that it has been satisfied when the method is finished. Since the best assessment of the current situation resides in the sensor memory, the task checks there to see if the robot arm is believed to be holding a fuel-drum. The "lift" operation itself might place such information in memory when it completes, but we will assume that it does not. Therefore, the method must contain an additional sensing step to prepare the sensor memory for

the task's satisfaction check. This is just as it should be: the task requires certain aspects of the world be known to ensure it has been satisfied, and the method it uses generates relevant sensing operations. The method now becomes: "scan", "move", "grasp", "lift", "check what the arm is holding". Finally, to prevent too much flailing about, if any of these operations fail at execution time, the rest will be abandoned – not the overall "pickup" task, only the rest of the active method.

Let us follow this method through a typical execution attempt. The task "pick up a fuel-drum" is selected for execution, and the method "scan", "move", "grasp", "lift", "check" is selected. The first of these operations, "scan", comes up for execution and is passed to the hardware. The robot looks around and records in memory that there is one fuel-drum off to the left. The "move" operation comes up next and the robot moves its arm to the location of the fuel-drum. Following through, it grasps the drum, lifts it up, checks the arm, and records that the arm is holding a fuel-drum. The overall "pickup" task consults the memory and sees that it is satisfied because the arm is holding a fuel-drum. Everything works fine under normal circumstances.

Suppose, however, that everyday problems begin to creep in. After the "scan" operation the robot believes there is a fuel-drum off to the left so it moves its arm to the left. Let us assume that in the meantime another agent has moved the drum to the right. The robot will attempt to grasp the fuel-drum and find nothing but air. At this point, either the "grasp" operation will fail because the hardware detects that nothing has been grasped, or else things will continue and the "check" step will eventually record that the arm is holding nothing. In both circumstances, the overall task will find that it has not been satisfied and will come up for execution again. The same method will be selected, and the "scan" operation will now find the fuel-drum off to the right. The system has both caught an execution error, and adapted its actions to a changed situation. The same sort of coping behavior results if the drum slips out of the robot's gripper during the "lift" operation, or if the robot gets called away in the middle of the method and must start the "pickup" task again.

This robust behavior in the face of uncertainties, failures and interruptions is a result of three things. First, the incorporation of sensing operations directly into methods means that relevant parts of the robot's assessment of the current situation are updated in a timely manner. Second, task methods are chosen based on this assessment and thus, tasks adapt their behavior as the situation changes. Third, the

tasks do not rely on method completion as a measure of success in the world; they contain their own satisfaction checks and continue to run until the world itself changes appropriately. These three features form the basis of situation-driven execution.

## 1.3 Reactive Action Packages

Reactive Action Packages, or RAPs, are proposed in this thesis as the basic blocks for building a situation-driven execution system. A RAP is a representation that groups together and describes all of the known ways to carry out a task in different situations. When a task is generated, its goal is used as an index to select a RAP, and when the task is selected for execution, the situation surrounding the robot is used to index into that RAP and select the most appropriate method it packages. A RAP includes the methods for a task and the contexts in which they are appropriate in a single package, so there needs to be only one RAP for every task type. However, there may be many independent instantiated tasks described by the same RAP definition.

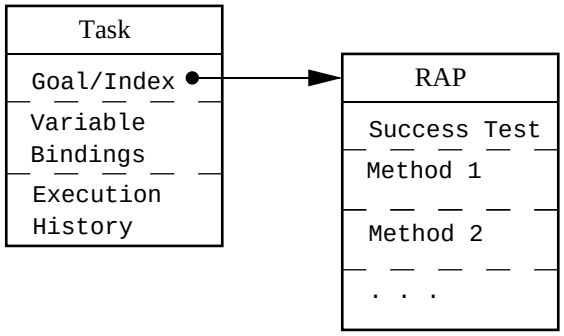



**Figure 1.3**: A Task in the RAP System

A RAP also includes a description of the situations in which its task is satisfied. A task has two parts: a goal, or index, and a test that defines when the task should be active. The activation test itself consists of two parts: a satisfaction test and a time-window. The satisfaction test tells whether or not the task's goal is satisfied in a given situation. If a task's goal is not satisfied, the task should be an active candidate for execution. A satisfaction test is the property of a particular task but since there is only one RAP for each task, the satisfaction test is described in the RAP. Thus, a RAP becomes a complete description for the execution of a task. It contains the task's

goal satisfaction test and all of the methods to achieve the task's goal in different situations. The time-window for a task must remain outside of the RAP for the task because it generally consists of start times and deadlines which will be different for every instantiation of the task. The various parts of a task are shown in Figure 1.3.

```
(DEFINE-RAP
   (INDEX  (load-into-truck ?object))
   (SUCCESS (location ?object in-truck))
   (METHOD
      (CONTEXT (or (not (size-of ?object ?size))
                   (and (size-of ?object ?size)
                        (<= ?size arm-capacity))))
      (TASK-NET
         (t1 (pickup ?object)
             ((holding arm ?object) for t2))
         (t2 (putdown ?object in-truck))))
   (METHOD
      (CONTEXT (and (size-of ?object ?size)
                    (> ?size arm-capacity)))
      (TASK-NET
         (t1 (pickup lifting-aid)
             ((holding arm lifting-aid) for t2))
         (t2 (pickup ?object)
             ((holding arm ?object) for t3))
         (t3 (putdown ?object in-truck)))))
```

**Figure 1.4**: An Example of the RAP Language

A simple RAP is shown in Figure 1.4. Its definition is split into three major sections: the INDEX which corresponds to the task-goal that the RAP is to satisfy, the SUCCESS clause which describes a test on the memory to determine if the goal is satisfied in the current situation, and any number of METHOD clauses that describe possible ways of carrying out the behavior under different circumstances.

Each method within a RAP is further broken down into two sections: the CONTEXT which describes a test on the memory that can be used to determine whether this is an appropriate way to achieve the desired behavior, and a TASK-NET which contains the detailed steps involved in the behavior. Task net steps consist of primitive actions which will be executed directly, calls to other RAPs to perform some behavior, and

constructs that allow loops and conditionals.

In this example, the RAP defines a task to make sure that a given object is inside the truck. This RAP is used for tasks with index `(load-into-truck ?object)` and will be satisfied when `(location ?object in-truck)` is true in the memory. There are two different ways of actually putting the object into the truck and the one chosen depends on the object's size. Note that if the object's size is not known, the first method is tried anyway. With luck (and an appropriate definition for the pickup RAP), the size of the object will be discovered during the pickup attempt and, if the object is too big, the second method will be tried as an alternative.

### 1.3.1 The RAP Execution Algorithm

Each RAP-defined task can be thought of as an independent entity, pursuing its goal concurrently with other tasks in the system by consulting the current world state and choosing a method to issues commands and alter that state. The RAP execution system shown in Figure 1.5 embodies this view of task execution. The system has three major components: the memory, the interpreter, and the RAP library.

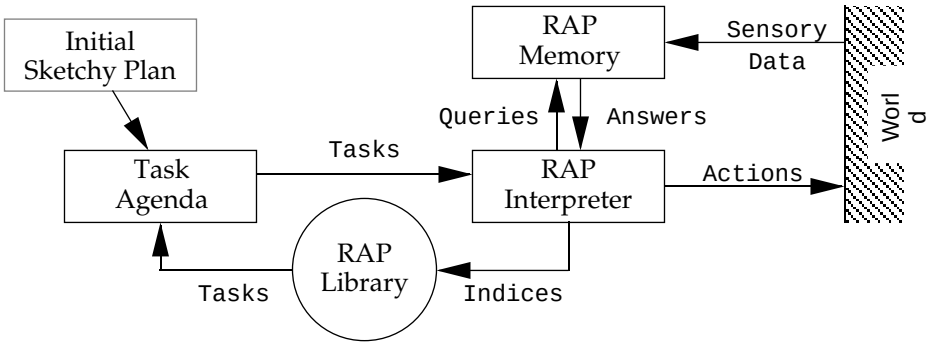



**Figure 1.5**: The RAP Execution System

The RAP memory contains the execution system's best estimate of the current world state, and thus represents the system's assessment of the current situation surrounding the robot. The RAP interpreter and task agenda provide a mechanism for coordinating task execution. A task waits in the agenda to be selected by the interpreter for its turn to run. When a task comes up for execution, it consults the memory and chooses a method as defined by its RAP code. Methods may contain

either primitive actions, which are sent to the robot, or subtasks which are sent to join the task agenda. When primitive actions are sent to the robot, it executes them immediately, interprets feedback such as sensor reports and effector failures, and updates the world model accordingly. When subtasks are sent to the agenda, they are associated with appropriate RAP code and become waiting processes just like any other task. After generating primitive actions and subtasks as method steps, the executing task suspends until all of the steps have completed. Situation-driven behavior is ensured because subtasks join the other tasks already on the agenda and the interpreter may select either a new or an old task to execute next, depending on which the situation favors.

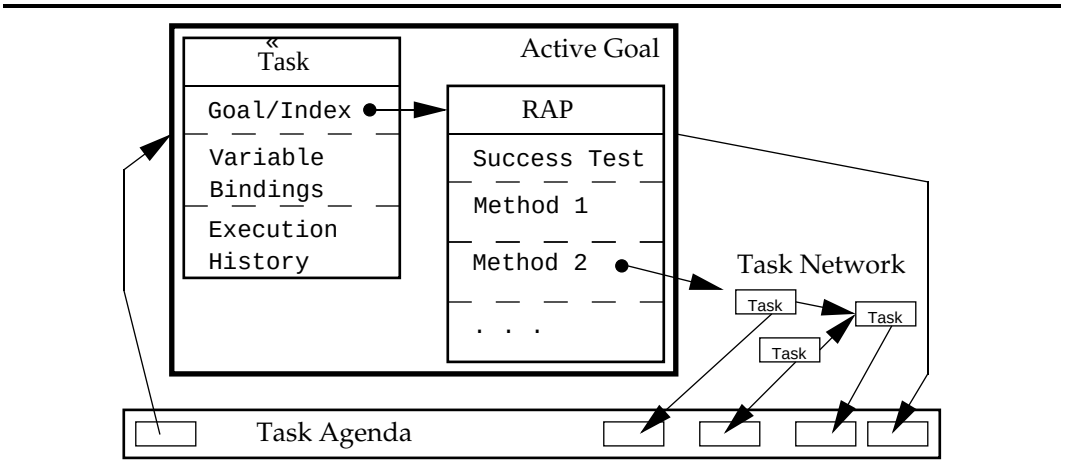



**Figure 1.6**: One Task Execution Cycle

RAP execution follows the algorithm illustrated in Figure 1.6. First, a task is selected by the interpreter from the RAP execution agenda. Selection is based on approaching temporal deadlines and on the ordering constraints placed on tasks by task nets. If the chosen task corresponds to a primitive command it is passed directly on to the hardware. Otherwise the task's RAP code is executed. As shown in the illustration, each RAP consists of two parts: a success test and a task net selector. RAP execution always begins by consulting the memory to see whether the RAP success test is satisfied. If it is, the task finishes immediately with success. Otherwise, a task net is chosen from the RAP. If no net is applicable in the current situation, the task must fail, but some net usually applies and it is sent to the task agenda. At this point, a plan has been selected for the task, and the task must wait to see how things

turn out. To wait, the task is returned to the execution agenda by the interpreter to be run again after its task net has finished. When the task comes up again, it executes exactly as before. Thus, a task keeps choosing task nets until either its goal is achieved, as determined by its success test, or the world state rules out every task net that it knows.

This method of specifying and running RAPs allows for a hierarchical and parallel pursuit of RAP goals, but raises the problem of coordination among the different subtasks in a task net. If an early member of a task net fails, then it is probably pointless to execute those that follow; the method represented by the task net simply isn't working. To deal with this situation, the system keeps track of task net dependencies and removes all the members of a task net from the queue when any one of them fails to achieve its goal.

Another problem with the algorithm is that a failure may not always change the world enough to guarantee that a different task net will be selected the next time the failed task is run. In that situation, the task will restart, note that its goal has not been satisfied and choose the same task net over again. If nothing intervenes to change the world in some way, such a loop could continue indefinitely. The current system has an execution-time loop detector that flags any RAP that selects the same task net repeatedly without success. Once flagged as a repeat offender, the RAP is made to fail so its parent can try and choose a different task net for its goal.

## 1.4 Objections to the RAP Approach

At this point, it is worth addressing two possible objections to the RAP execution system as described in this thesis. The first objection is that RAPs are "just another robot programming language" and the second is that "the RAP approach cannot be evaluated until RAPs have been used to control a *real* robot". To some extent, both objections do apply to the RAP system — RAPs do define robot control procedures, and the research reported in this thesis does not include experience on real robot hardware. This section discusses each objection in turn.

### 1.4.1 RAPs and Robot Programming

The objection that RAPs are just another programming language implies that the only contribution of this thesis to robot control is a system for representing action schemas. This same objection can be leveled at other current efforts to design adaptive robot control systems: the REX system [Kaelbling, 1986b] and the PENGI system [Agre and Chapman, 1987] in particular. The REX system is specifically designed to be a robot programming language, and as such is defended by its ability to flexibly describe very general robotic tasks. The PENGI system is an implementation of a theory of "situated activity", [Chapman and Agre, 1986] and is defended, not by its contribution to robot programming, but by the theory's power to describe real-world behavior.[1] The RAP system is both a programming language and a proposal for structuring behavior descriptions and can be defended on both counts.

The RAP representation syntax is clearly a robot programming language. The RAP library describes all of the tasks the system knows how to carry out, and the RAP interpreter executes a task by interpreting its RAP description as a program. As a programming language, the RAP system is well-structured, flexible and easily extensible. RAP syntax is well-structured in the sense that a RAPs basic behavior *in the world* is usually readily apparent from the way it is coded. The various components of a RAP clearly define when it is applicable, when it will finish, the methods it can employ to carry out its goal, and the situations in which each method will be used. While being rigid enough for easy interpretation, the RAP syntax is also flexible enough to describe a wide variety of task behaviors. The well-structured, yet flexible, nature of the RAP syntax makes it a perspicuous language for many robot programming tasks, particularly those that require sensing operations. The structured nature of RAP descriptions, and their interpretation, also makes it reasonably easy to extend the RAP library as new tasks, or methods, are required. This is in direct contrast to current robot control work based on the construction of specialized circuit descriptions [Agre and Chapman, 1987, Brooks, 1986].

The RAP system is also designed to be more than "just a programming language". The interpreter algorithm is fixed and unchanging, and therefore, the semantics of

[1] The adaptive robot control work done by Kaelbling and Agre and Chapman is relevant to, but quite different from, the RAP system. The relationship between this work and the RAP system is discussed in detail in Section 7.5. Other recent research is discussed there as well.

RAP-defined behaviors are well-defined. Thus, the RAP system makes a strong claim about the way behaviors are represented and interact with one another. In fact, this claim was a design goal of the RAP system since it enables RAPs to be treated as separate, declarative planning operators as well as program descriptions. Specifically, RAPs are designed to be used in two different ways: as abstract planning operators, and as programs to be run by the RAP interpreter. To support both uses, the semantics of the RAP interpreter must to be fixed and the RAP syntax must be structured enough so that RAP behavior in the world is readily apparent. The use of RAPs as planning operators is discussed in Chapter 7. More experience in planning with RAPs is required before strong claims can be made in this regard, but many of the algorithm and representation design decisions evident in the next chapters were made to support the use of RAPs as planning operators (see Hanks [1989] for more on this topic).

Thus, the RAP system is more than a robot programming language. It defines a well-structured, flexible and extensible mechanism for describing modular behaviors that can both be executed and reasoned about. The RAP system itself treats these descriptions as programs, but the ability to reason about them independently provides a hook for interfacing the system to more deliberative planning and problems solving processes. The modular nature of RAP behavior descriptions also makes it possible to add new RAPs to the library and hence new behaviors to the robots repertoire.

### 1.4.2 Experimenting with the RAP System

Another objection to the RAP system is that it has not yet been used to control a real robot. A major difficulty with designing execution systems to run in complex worlds is finding a world in which to test them. The real world is best, but unfortunately, using it requires expensive immature hardware and facing many low-level sensor and control problems that have little to do with critical issues in reactive execution. To get around these problems, the RAP system has been extensively tested using a world simulator. The simulation consists of a set of locations, roads and objects and a robot delivery truck under external control. All interaction between the RAP system and the simulated world takes place through the delivery truck.

The objection is that it is just too easy to leave out types of complexity that are *inherent* in the real world when constructing a simulator. Methods and algorithms

for solving the simulated problems may then make assumptions that cannot be generalized and result in a system incapable of solving problems that fall outside the simulated domain. There is also a great temptation to incorporate unrealistic sensors (or no sensors at all) into the simulator; thus giving the system powers of perception unobtainable in actual hardware. The argument is that building an execution system for such a simplified physics and technology will contribute nothing to our understanding of acting in real domains and therefore the only way to validate a system is to make it cope with the real world.

Are simulators a useful tool for program verification or are they a seductive deception which lead researchers to devise methods and algorithms that contribute nothing to our understanding of robot control? Since simulators have undisputed value as program and theory verification tools in other areas, the real question is whether or not the use of simulators leads robot control researchers astray. Arguments over this point are really debates about basic research methodology and not about simulation at all. To oversimplify things somewhat, the study of intelligent action can be approached in two ways: either one can choose an abstract, high-level behavior, produce algorithms that exhibit that behavior in simple domains and then expand the scope of the algorithms until real world activities are covered, or one can start with simple, low-level activities, produce systems that perform those activities and then build on those systems until more complex behavior emerges. Planning research is a result of adopting the first strategy, while robotics and sensor research are examples of the second. It has always been assumed that the application of planning to more realistic domains and the combination of robotic techniques into more complex machines would meet somewhere "in the middle", and the problem of intelligent action as a whole would be solved.

Those arguing against simulation do not subscribe to this belief. Whether they concede that simulators are useful aids for the verification of planning algorithms is irrelevant, for they claim that algorithms developed for restricted domains simply will not generalize to the real world. The only way to show that a planning theory is relevant is to test it in the real world and not in a simulated world. The real issue however, is not whether simulation is a good idea, but whether or not researchers can learn anything from studying robot control in restricted domains. At this point the issue remains unresolved and it probably will remain unresolved until the debate is

taken up by our robots. In the meantime, I hope and believe that the RAP system can indeed be extended to the control of real robots, even though it has been extensively tested only in a simulated robot delivery truck domain. I feel confident that delivery truck domain represents a real step away from blocks style micro-worlds and towards the complexity and uncertainty of the "real world."

The delivery truck simulator models objects in the world in some detail, including liquids, containment to arbitrary depth, variable permeability to given factors (like radiation), and internal processes. Space is modelled in somewhat less detail, being just complete enough to give objects a place to be and to allow for the truck to move from one place to another. Coupled with the uncertainty that can be associated with all simulated behavior, this simplified world model supports even the most complex problems we hope to address.

### The Simulated World

The world simulation is built around the concept of independent objects. Everything in the world is an object and each object can change with time according to its own internal program. Some objects are passive, like fuel drums and truck tires, but others can be quite active, like enemy tanks, manufacturing machines and garbage disposals. For example, there might be a machine in the world to assemble medical supply kits. The robot truck would bring rolls of bandages and vials of medicine to this machine, put them inside and then turn it on. Sometime later, independently of what the truck goes off to do in the meantime, the machine would generate fresh supply kits. Like everything else, the robot truck is an object within the world.

The world is divided into locations and roads that link the locations together. All objects, including the truck, can appear only at locations or along the road. There is no notion of open country-side and once started, the delivery truck travels down a road without any additional supervision. Although this simplification eliminates the problems of road following, navigation through unknown territory can still be simulated with a dense network of short roads and country-side locations.

Locations and roads together form a map within which the simulation takes place. This map is divided into sectors which are used to model large-scale phenomena like weather, darkness and enemy activity. In general, the behavior of objects and roads

may depend on these global characteristics. For example, while travelling on some roads the delivery truck is more likely to get stuck when it is raining than when it is sunny. Similarly, enemy troops may appear unexpectedly at a location in a sector of high enemy activity or a machine may only run properly in the daylight.

### The Delivery Truck

The robot delivery truck is the executor's access to the simulated world. The truck responds to external commands and returns sensory data about its surroundings. It can move from one location to another and has several cargo bays, loading arms and a vision-like sensor. The loading arms are used to manipulate objects near the truck and objects can be carried from one location to another in the cargo bays. The sensor is used to determine which objects are at the current location and when new objects arrive. It can also examine particular objects in detail. For example, the truck might arrive at a new location, scan the area with its sensor and discover a gasoline container, pick the container up and examine it to determine how much gas is inside.

More details of the robot delivery truck domain can be found in Appendix B and Firby and Hanks [1987b, 1987a]. The important feature of the domain is that it is substantially more complex than traditional blocks-world simulators. A task such as "take ten fuel drums from the fuel-depot to the warehouse" will typically take hundreds of primitive actions at the level of "move-arm", " grasp", and "turn-east". Furthermore, primitive actions are not guaranteed to succeed and may in fact fail in several different ways. For example, the "arm-grasp" operation might fail because the item slips out its gripper, because the item is too heavy, because a tool is needed, or because the item can't be found. Moreover, the inherent uncertainty in the world, like the possibility of dropping an object, of rain, or of enemy troops showing up, means that the robot can never be sure exactly what its future holds.

The RAP execution system must be able to cope with all of these problems while muddling through a sketchy plan.

## 1.5 Issues in RAP Execution

Within the context of robot control, the goal of traditional planning research has been to develop a procedure that will transform any task into a set of simple actions for the robot to use in accomplishing the task in a given situation (see Chapman [1985] and Joslin *et al.* [1986] for excellent summaries of this work). Substantial progress has been made towards this goal using the concept of expanding each task from a high-level general description into levels of more and more detailed subtasks. At each level in the expansion hierarchy, undesirable interactions between proposed subtasks are identified and eliminated, either by reordering the subtasks [Sacerdoti, 1975, Vere, 1983] or choosing different expansions [Dean *et al.*, 1988, Wilkins, 1988]. The end result of the expansions is a detailed set of primitive robot actions to be executed without further deliberation. Two problems have remained, however, when such plans are used to control an actual robot: monitoring execution to make sure that the actions being executed are actually carrying out their intended effects, and local replanning to patch the plan when execution monitoring determines that things have gone awry.

Execution monitoring and local replanning are precisely the problems that the RAP system addresses. Both problems arise primarily because planning and primitive action execution take place at very different levels of abstraction. A primitive action is supposed to instruct the robot hardware to perform a well-defined, concrete action in the real world and, thus, requires an appropriate initial state. On the other hand, the purpose of planning is to take very high level goals and produce sequences of simpler tasks to carry them out. The mistake is asking planning to generate task sequences at the level of primitive actions. In complex, dynamic domains, planners cannot produce accurate plans in such detail; therefore, inappropriate actions will inevitably be included.

As domains under consideration become more complex and more dynamic, the problems become more critical: primitive actions must remain simple and concrete, but planning must become more and more general. The RAP system is designed as a bridge between these levels by allowing planning and problem solving to generate tasks that are more abstract than primitive actions. The tasks are then translated into appropriate concrete actions only as the relevant world state makes itself evident

at execution time. Between the abstract level of the planned tasks and the concrete level of the primitive actions the RAP system is free to adapt its actions, monitor their progress, and try them again if the situation warrants. Thus, execution monitoring to choose task methods and monitor their success is an intrinsic part of RAP execution, as is local replanning in the form of reselecting task methods when previous choices don't work out.

This returns us to the three layer model of robot control mentioned earlier (see Figure 1.1). At the lowest level is a robot controller to translate primitive actions into coupled, concurrent instructions to robot motors and sensors. At the highest level is a planning system to translate system goals into task sequences at whatever level of detail uncertainty and lack of information allows. Between these two levels the RAP execution system translates the planned tasks into primitive actions. The planner relies on the RAP system to monitor task execution and adapt actions to the situation at hand so that it can treat any tasks as directly executable. Similarly, the RAP system relies on the robot controller to carry out primitive actions using whatever continuous manipulation of robot motors and sensors is necessary so that the RAP interpreter can treat primitive actions as discrete operations. The robot controller hides the complexity of real robot hardware from the RAP system and the RAP system hides the complexity of execution monitoring and local replanning from the planner. This view of robot control is discussed in some detail in Chapter 7 after the RAP system has been described.

### 1.5.1 Problems in Adaptive Execution

In tackling the traditional planning problems of execution and local replanning, the RAP execution-system must address several new issues and problems. The system must confront problems associated with sensing in the world and it must relate sensing to memory. It must grapple with local uncertainties in the memory and the resulting execution failures, and there must be a way for the system to synchronize task execution with specific states and events in the world. Difficult semantic issues involving task execution windows and the meaning of plan-time protections must also be sorted out. All of these problems are inherent in dealing with any complex, dynamic domain, and they cannot be avoided, even in the simplest everyday situations.

### Sensing and Memory

It goes without saying that the RAP system must connect intimately with robot sensors and effectors. In particular, the robustness of the system depends on having a detailed, up-to-date description of the current situation represented in the RAP memory. This description must be derived from sensor data and not merely from simulating the robot's own actions. Correct execution of specific actions cannot be taken for granted, and the actual results of an action may not be those anticipated. Therefore, the way the RAP memory is updated and the assumptions it makes in its representation must be carefully examined.

### Uncertainty and Method Failure

RAPs choose actions to execute based on a local view of the world which is certain to be incomplete and may even be incorrect. Therefore, execution of those actions may produce inappropriate results, or even turn out to be impossible, in the real world. Detecting and recovering from such errors must take a central place in the RAP system. In particular, RAP representations must include sensing tasks specifically designed to reduce memory uncertainty and examine the memory features relevant to testing execution success.

Furthermore, the RAP system must include ways of describing tasks that actively, and repeatedly, monitor states of the world and react when they change. For example, enemy troops do not announce their presence automatically. The robot delivery truck can find out about new items in the vicinity only if it actively scans for them. If the truck is to react quickly when enemy troops arrive, it must spend time watching for them.

### Synchronizing with the World

Tailoring the robot's actions to take place at the proper time in response to the proper stimulus is another necessary capability. Acting on an emergency like a fire, recognizing an opportunity like twenty dollars on the sidewalk, or just waiting for an elevator to arrive, are all situations that depend on generating or suspending a task based on the changing state of the world. The RAP system must allow the

representation and implementation of methods that can properly synchronize their behavior with shifting opportunities. When the kettle finally whistles, it is time for the robot to stop what it is doing and go make tea.

#### Planning Protections and Task Semantics

An unexpected but critical issue at execution time is the precise meaning of a task description. What does something like `(ACHIEVE (ON A B))` really mean? Does it mean that `(ON A B)` should be achieved once and then forgotten (the usual blocks world semantics), or does it mean that `(ON A B)` should be achieved and protected for some interval of time? If it means achieve and protect, what does protect mean? In traditional planning research, protections are often used to prevent the planner from placing tasks into the plan that will upset a state of the world that should be preserved. At execution time, however, such a protection can mean a variety of things. It might mean that nothing should be done to change the state, it might mean it is all right to change the state temporarily but it should be changed back, and it might mean that if the state is changed some other task should be forgotten. Similarly, the RAP representation should be able to describe all of the possible meanings of a task like "keep the factory free of empty drums". It might mean to remove an empty drum if it is discovered while doing other things, it might mean never generating an empty fuel drum, or it might mean continually searching for empty drums and tossing them out.

### 1.5.2 Outline of Thesis

The need to cope with complex, dynamic domains while executing a robot plan leads naturally to a theory of coping with sketchy plans. A sketchy plan is a collection of tasks to be executed by the robot, but which cannot be turned into detailed primitive actions because uncertainty about the future of the domain prevents foreseeing which actions would be appropriate. Execution of a sketchy plan must therefore adapt primitive actions to the situations the robot encounters at execution time and must cope with robot execution failures and interruptions. To handle all of these problems, a theory of situation-driven execution is necessary. This theory suggests the algorithm:

- Choose a task from the sketchy plan to execute.
- Use the task and the current situation as indices into a library of known methods for achieving the task in different situations.
- Execute the method and choose a new task.

If tasks remain active for consideration as long as their goal is not accomplished, even after a method has been chosen and executed for them, they will be repeatedly attempted with different methods using this algorithm.

The RAP system is proposed as a concrete implementation of situation-driven execution theory. A RAP is a compilation of all of the methods for accomplishing a task along with the context in which each method is applicable. When a task is introduced into the system as part of a sketchy plan, it is associated with a RAP to form a process that pursues the task. RAP execution involves matching the current situation to each of its contexts and choosing the corresponding method. The RAP system itself breaks down into three parts: the memory, or world model, that holds the current assessment of the situation around the robot, the interpreter which chooses an available RAP to execute next, and the RAP library which contains the RAP representations themselves.

The rest of this thesis consists of six chapters. Chapter 2 presents the RAP memory system and discusses the way it interacts with sensor representations of the situation around the robot. Issues of uncertainty, object identity and object identification are also addressed. Chapter 3 describes the RAP interpreter and the way that RAPs are created, handle failures, and interact with each other. Chapter 4 concentrates on the RAP representation in detail and illustrates the way different execution-time behaviors can be described. These three chapters together describe the entire RAP execution system.

Chapter 5 contains a discussion of task semantics and gives examples of ways many different types of tasks, protections, and policies can be represented as RAPs. Chapter 6 brings everything together, describing some actual experiments run within the robot delivery truck domain. This chapter argues that situation-driven execution, and the RAP system in particular, really does produce robust behavior in the face of complex, dynamic domains.

Finally, Chapter 7 presents a discussion of the ways RAP representations might be used as plan-time operators as well as execution-time method selectors. The three-layer robot control system is presented in more detail along with current research that supports each layer. A detailed discussion of other current proposals for adaptive execution systems is also included in this chapter. Leaving the discussion of related work until after the RAP system is described, allows for more useful comparison.

# Chapter 2

# The RAP Memory

Ideally, a situation-driven execution system would consult the real world to evaluate the current situation. However, consulting the real world means examining it through sensors, and limited sensor bandwidth and range restrict direct access to only small parts of the robot's immediate locale. A robot cannot expect to see everything at the same time, and generally, things far away will be completely inaccessible. Therefore, if a robot is to base execution decisions on a reasonably large picture of the world, it must maintain a model of what it cannot see. Such a model, or memory, forms a buffer between the executor and its sensor systems. While the sensors are busy acquiring new information about a small, local feature of the world, the memory keeps track of what has already been observed about the rest.

The use of memory as a buffer between execution decisions and sensing hardware requires the RAP system to be structured as shown in Figure 2.1. Hardware sensing systems acquire information about the world near the robot and send it to the memory system. The memory integrates the new information into its model of the world; keeping the overall model consistent and up-to-date. Executing tasks consult the memory system for the best information available about the current world state and choose robot actions based on that information.

This relationship between the sensor hardware, the memory, and RAP decision making has two important features. First, RAP-defined tasks do not directly affect the memory. They change the contents of the memory only indirectly, by issuing sensing requests to the hardware. When a sensing operation is complete, the hardware hands

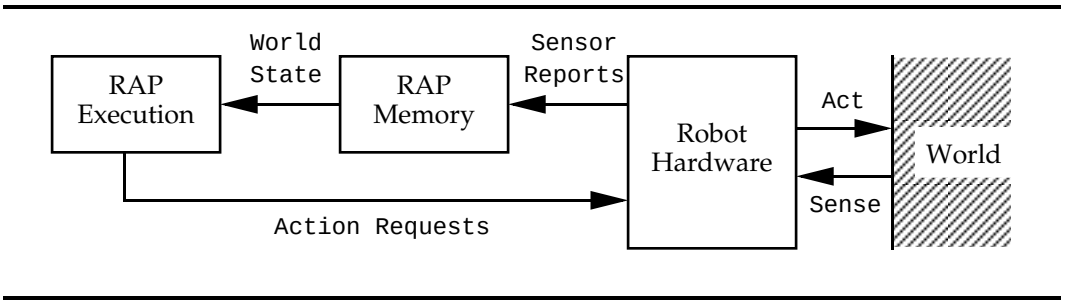



**Figure 2.1**: Memory in the RAP Execution System

the resulting information to memory and the RAPs can examine it there. Second, the memory itself does not actively solicit information about the world. The memory relies on executing tasks to issue enough sensing requests to keep relevant portions of the world model up-to-date. Thus, the memory maintains a passive, stable model of those aspects of the world that executing tasks choose to keep up-to-date.[1]

An interesting result of using memory is that previously encountered items must be recognized when they are seen again. Memory can only be kept consistent as new sensor information becomes available if that information can be compared with what is already believed. To make such comparisons, new and old pieces of information must be recognized as corresponding to the same aspect of the world. Traditionally it has been assumed that the sensors themselves do this recognition automatically. The RAP memory does not make this assumption and must do the recognition itself.

This chapter discusses the RAP memory and the way it assimilates new information. From the viewpoint of RAP execution, the memory looks like a stable, assertional database. However, the memory must take new sensor reports from the hardware, actively match them against previous information, and decide what to add and delete from its database. The first section describes the RAP memory model and the issues involved in its construction. The next section discusses memory as the assertional database seen by the RAP interpreter, and the final section describes the process used by the memory to match new information with old.[2]

[1] The RAP system and memory model support the notion of unsolicited sensory information as well. That is, the robot hardware might generate sensory data without being asked for it. An advantage of unsolicited reports would be to make the RAP system more responsive to its environment, but there is no way to make all sensing unsolicited. Therefore, the memory is built to behave exactly the same way with or without unsolicited reports — none of the discussion that follows either assumes or prevents such reports.

[2] As a note of clarification, the RAP memory described in this chapter is designed primarily for

## 2.1 The RAP Sensor Model

The form that a robot memory might take is heavily dependent on the type of information that can be expected from the robot's sensor systems. Two alternative sensor models explored in the planning literature are the fixed-designator model, and the indexical-functional model [Chapman and Agre, 1986, Agre and Chapman, 1987]. The fixed-designator model assumes that sensors can recognize individual items in the world and attach the same unique name to each whenever and wherever it is encountered. The indexical-functional model assumes that individual items can be differentiated and classified, but that item identity cannot be tracked by the sensors, even from moment to moment. For example, imagine the robot enters a warehouse and discovers three fuel-drums that look the same. The assumption of the fixed-designator sensor model is that the robot can identify the fuel-drum on the left as the very same one that it saw somewhere else last Wednesday, even though it is indistinguishable (in sensory terms) from the fuel-drum on the right. In contrast, the indexical-functional model allows the robot to refer to "the fuel-drum on the left" but makes no claim as to whether the fuel-drum on the left is the same item should the robot turn away and then look back.

Both the fixed-designator and indexical-functional assumptions present problems as useful sensor models. The strong recognition assumption in the fixed-designator model is clearly unrealistic. No sensor system can be expected to differentiate between indistinguishable items. On the other hand, the indexical-functional model goes too far the other way by giving "the fuel-drum I am watching" no permanence at all.

The sensor model used by the RAP memory system assumes capabilities between these two extremes: a temporary name is assigned to each individual item in the world and it remains unchanged as long as the item stays within sensor range. However, once the item moves out of sensor range, its temporary name is lost and it will be assigned a different name when it is encountered again. For example, when the robot arrives at the warehouse and discovers three fuel-drums, they will be assigned temporary names such as `obj-1`, `obj-2` and `obj-3`. The fuel-drum named `obj-1` will

maintaining a model of the external world. The RAP system's internal state is tied up in the active tasks and the task agenda. There is currently no memory that contains assertions recording the system's intentions or past activities. This separation is not strictly enforced, however, and there are some exceptions (see Sections 4.1 and 4.9.)

remain `obj-1` in all sensor information reports until the robot leaves the warehouse or loses track of the drum in some other way. When the robot returns and sees the same fuel-drum, the sensors will not know it is the same one and will assign it a different name, perhaps `obj-45`. The temporary sensor name assigned to an item might be thought of as a detailed position that identifies the item within the robot's immediate frame of reference.[3] It is assumed that an item can neither be sensed nor acted upon unless it has a current temporary name. An item with a current temporary name will be termed *accessible*.

### 2.1.1 Constructing a Local Model

When an item becomes newly accessible, and is sensed through some robot operation, two things happen. First, the robot hardware assigns it a local sensor name. All sensor reports about the item will refer to it using the sensor name, and all robot commands to act on the item must use the same sensor name. Second, the memory system is notified that a new item with the given sensor name has become accessible. The memory system then generates a new globally unique symbol to represent the item and asserts the sensor name as a property of the symbol. For example, when the robot rolls into the warehouse and uses its scanner to look around, it may see a fuel-drum. The robot hardware will assign this fuel-drum a sensor name such as `obj-45` and report to the memory that a new item called `obj-45` has become accessible in the current situation. The memory generates a unique symbol, say `item-178`, and asserts that `item-178` has sensor name `obj-45`. As more sensor reports come in about `obj-45`, more assertions are made about `item-178` and a detailed picture of the fuel-drum will emerge.

The memory name of the item links assertions about its various properties together. When initially detected, an item's class and location are immediately available, but more specific sensing actions are required to determine the item's other properties, like size and color. It may even be necessary to open an item up to see what is inside. Each sensing action generates a separate message from the hardware

[3] An item maintaining a fixed sensor name while accessible is an approximation. The assumption is that an accessible item remains within sensor range so that any changes in its position or properties will be noticed and its individual identity can be tracked directly. Only when the item is out of sensor range, and can no longer be directly perceived does it lose its identity.

and the memory assembles them into a description of the item as a whole using the item's memory name.

While an item is accessible and has an active sensor name, it is said to belong to the memory's *local model*. The local model contains all item descriptions still under construction and forms a detailed description of the situation currently accessible to the robot. By definition, the hardware only generates information about items represented in the local model.

As long as an item remains accessible to the robot, it will retain the same local sensor name and its description in the local model will represent the best information known about it. When the item becomes inaccessible, however, the memory must be changed so that the robot knows that the item can no longer be acted upon directly. The local name translation between the sensor and memory names for the item can also be dropped because no more sensor reports with that sensor name will be generated. Continuing our example, when the robot leaves the warehouse and loses contact with the fuel-drum, the sensor name `obj-45` becomes useless. There are two choices at this point: erase the whole description of the item because it can no longer be acted upon (*i.e.* all assertions that mention `item-178`), or just erase the single assertion that `item-178` has sensor name `obj-45` and leave the rest of its description intact for future reference.

### 2.1.2 The Case for Long-Term Memory

Erasing the entire description of an item from memory when it becomes inaccessible is correct in the sense that it keeps the description of the accessible local situation in RAP memory accurate. However, there are at least two reasons why it makes sense to keep the description of an item around after it becomes inaccessible. First, the item, or the same type of item, may be needed at some point in the future. By keeping descriptions around, the robot will know that it has encountered such an item, and will know where to look for it. The memory will represent not only the accessible local situation, but what used to be true in past situations as well. For example, after leaving the warehouse, the robot may develop the need for a fuel-drum. If none are known in the local situation, the robot will still know that there used to be a fuel-drum in the warehouse and it can go there to look.

The other reason the robot might want to keep descriptions of inaccessible items around is because they may be encountered again. If a newly accessible item can be *recognized* as an item that has been seen before, all information acquired about the item in the past can be brought forward without having to discover it again. For example, suppose that on a previous trip to the factory, the robot encountered two boxes, a red one and a yellow one, and put a bomb in the yellow box to detonate on opening. Upon re-entering the factory, the robot will discover that two boxes are accessible, a red one and a yellow one. If the memory can make the appropriate identification that the new yellow box is the same as the old yellow box, the robot might avoid accidently blowing itself up.

A crucial point is that retaining past item descriptions in memory necessarily entails the problem of identifying new item descriptions with old ones. If old item descriptions are not matched with new ones, the memory will rapidly become cluttered with multiple descriptions of the same item and the whole thing will become meaningless. Consider what happens when the robot enters a warehouse and discovers a single blue fuel-drum. The robot leaves, but retains the description so that it will know where to look if a blue fuel-drum is ever required. Later, the robot re-enters the warehouse and encounters a blue fuel-drum. If no attempt is made at identification, and the robot again leaves the warehouse retaining the blue fuel-drum description, there will be descriptions of two blue fuel-drums in memory. If the robot returns again there will be three and so on. To prevent multiple descriptions for the same item, the memory must try and match new descriptions with old ones so that the old ones can be removed or merged together.

### 2.1.3 The RAP Memory Model

This RAP memory model is summarized in the diagram in Figure 2.2. As a real world item becomes accessible to the robot, it is assigned a local sensor name by the hardware and the memory begins to construct a description of it in the local model. As more information is generated by the robot sensors, the local model description of the item becomes more and more detailed. When the item becomes inaccessible, the hardware notifies the memory and the item's local sensor name is removed from its description. Removing the item's sensor name moves the item's description out

of the local model and into long-term memory. Meanwhile, the memory system is continuously trying to match long-term memory item descriptions with new local model descriptions as they become more detailed. When a match is discovered the old description is merged with the new to form a single, unified description.

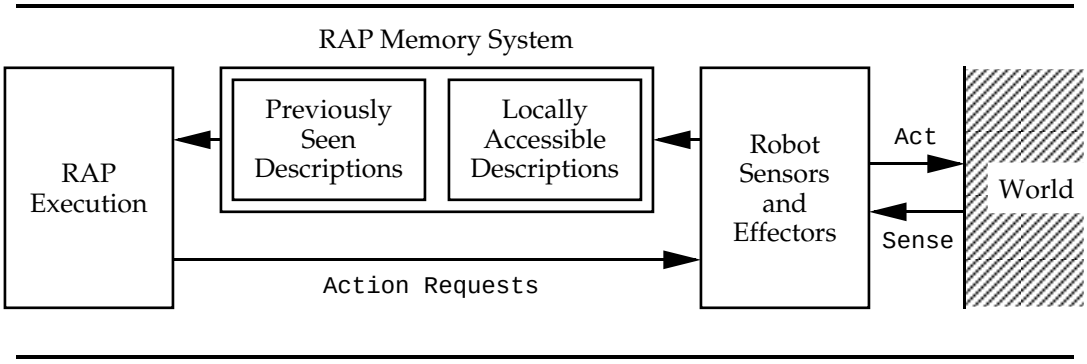



**Figure 2.2**: The RAP Memory Model

The rest of this chapter is concerned with two aspects of this memory model. First, there must be a uniform query mechanism for both the local model and long-term memory. The division between the two should not be visible at the memory-as-database interface. RAPs should be able to direct robot activity based on the best description available without having to worry about whether it is based on new or old information. Section 2.2 describes memory queries.

The second aspect of memory discussed is the way item descriptions from long-term memory are matched with those being constructed in the local model. Given our assumptions about the sensor hardware, recognizing a newly encountered item as one that has been seen before is the responsibility of the memory system. Section 2.3 discusses the RAP memory algorithms for description matching.

## 2.2 The RAP Interpreter/Memory Interface

The RAP memory must be tailored to the acquisition and representation of item descriptions but it is more convenient for RAP descriptions to see memory as an unstructured database of ground assertions. Therefore, the complications inherent in matching local and long-term item descriptions is completely hidden within the database. The RAP interpreter simply queries database assertions to evaluate the state of the world.

Assertions in the database are propositions that contain no variables. Proposition may take any form, but most describe item properties and are represented as fluents [McCarthy and Hayes, 1969]. For example, the fact that `item-45` holds two other items is represented as:

```
(contains item-45 item-23 true)
(contains item-45 item-98 true)
```

Queries to the RAP memory consist of proposition patterns to be matched against assertions in the database and may be combined using the connectives `and`, `or` and `not`. When a pattern contains free variables, they are treated as existentially quantified and are bound, if possible, so that the pattern matches some assertion in memory. An executing RAP uses the success and failure of such matches to determine when specific situations are true in the world. Queries may also include predefined boolean functions like `=`, `<`, and `>` so that variable bindings can be restricted to particular values. For example, to check whether `item-45` contains an item bigger than `item-51`, one might use the query:

```
(and (contains item-45 ?item true)
     (size ?item ?size1)
     (size item-51 ?size2)
     (> ?size1 ?size2))
```

There are standard problems with using an assertional database as a memory. First, there is the question of what it means when a query fails to unify with facts in the database. Does it mean that the fact isn't true, or does it mean that the robot has never had a chance to check the fact and doesn't know anything about it? Second, there is the problem of deciding how long a particular fact should remain in the database without ongoing confirmation. For example, believing there is twenty dollars on your dresser is warranted until you remove it, but believing there is twenty dollars on the subway seat beside you is only reasonable until you get off the train.

The RAP memory does not attempt to solve the general problem of missing assertions. It is up to each RAP definition to decide what it means when a query is unsatisfied. However, by defining a special, restricted syntax for item properties, the memory makes it possible to define defaults for missing item information. Defaults derive from functions that are called when the value of an item property is required

but is not available in memory. The problem of dealing with aging assertions is handled in a somewhat more general fashion. Each memory assertion is tagged with the time it is asserted, and that time can be accessed using a special form called `believe`. This section discusses the RAP memory representation for item properties and goes on to discuss default functions and the `believe` construct.

### 2.2.1 Item Properties and Descriptions

An item property assertion takes the general form:

*(property-name item-name [specifiers] property-value)*

The *property-name* is a symbol to identify the property being represented, the *item-name* is the name indicating the item in memory and the *property-value* is the value of this property for this item. Specifiers are included when more arguments are required. For example:

```
(color      item-23 red)
(works-okay item-23 true)
(contains   item-45 item-23 unknown)
(contains   item-45 item-98 true)
```

are all reasonable property assertions.

Item property values are treated in a distinguished way by the RAP memory. Properties are assumed to take on only one value at a time and the necessary inference rules to enforce this are built into the memory. In particular, when a new property assertion is made, previous assertions for that property are removed from the database. If `item-17` is thought to be yellow, the assertion `(color item-17 yellow)` will exist in memory. Later, when `item-17` is painted red, the assertion `(color item-17 red)` will be made. Since `color` is declared to be a property, the old assertion `(color item-17 yellow)` is erased. The value of a property may also take on the distinguished value "unknown" meaning that its current value is explicitly uncertain.

The structure of item properties allows important simplifications in memory processing. First, all of the assertions making up an item description can be pin-pointed

and grouped together easily. This is important when descriptions from the local model must be compared and combined with descriptions from long-term memory. Also, default values can be assigned to unsensed item properties without having to build general purpose default reasoning machinery. This reduces both the complexity and the computation required within the memory itself.

### 2.2.2 Default Property Values

Making clear what is meant when an assertion is sought but not found is often a problem in an assertional database. If a specifically contradictory assertion is present in the database then the assertion sought is obviously false. However, when the assertion being queried is simply not there, it might mean either that the assertion is false or that the particular property described by the assertion has never been sensed and the robot just doesn't know anything about it.

One solution to the problem of assigning truth values to missing assertions is to use the rules: "true if present", "false if contradicted", and "unknown if neither present nor contradicted". This solution implicitly makes the default assumption that the best inference when a property has not been sensed is to label it "unknown". While always correct, such an assumption is too strong in many planning situations. Most items in the world have a large number of properties that are normally true and can be taken for granted. The classic example of this is "birds fly". If we know that `item-21` is a bird, we would prefer the assertion `(can-fly item-21 true)` to appear in memory by default rather than `(can-fly item-21 unknown)`. A more general method must be used to supply default values to property assertions in memory.

There is a substantial literature on default inference in predicate calculus databases [McCarthy, 1981, Lifschitz, 1986, Reiter, 1980, McDermott and Doyle, 1980, McDermott, 1980], but much complexity is avoided in the RAP memory system by allowing default values only for unsensed item property assertions. That is, when a query about a property is made and no matching, or contradictory, assertion is found, a default is generated for the property's value. Although assertions may be used to represent many things in the memory besides item properties, no default reasoning is done with assertions that are not declared as item properties — the RAP memory does not do general purpose default reasoning.

### Specifying Default Values

Item property assertions have two parts: an index and a value. The last argument of the assertion is the value, and everything before it is the index. Every property-type is assigned a function for calculating its default value and that function is called at query time, if necessary. For example, it is generally reasonable to believe that items in the robot delivery truck domain function as they were designed to. If the property `works-okay` is used to signify a working item, it can be assigned a default function that returns "true", so that when an item is encountered in the world it will be assumed to work. Hence, queries of the form:

```
(works-okay item-34 true)
```

will succeed for all items that do not have specific `works-okay` assertions in the database.

Functions are used to supply defaults so that the values generated can be made context-sensitive. For example, the inference that an item works may be good in general, but if the item is found at the junk-yard it may be better to infer that it's broken. Similarly, if the item is encountered at the repair shop it may or not be fixed and it would be best to infer that its `works-okay` property is "unknown". The syntax of default functions is described in Appendix A along with examples from the delivery truck domain. The two most commonly used functions during experiments were context-insensitive and returned "false" and "unknown".

### Inferring Default Values

Only the value argument of a property assertion is allowed a default. This means that every item known to the memory will have a virtual assertion for each declared property type. Ruling defaults out of the index portion of a property assertion prevents the memory from generating properties for items that it does not know about. For instance, the query:

```
(works-okay item-45 ?okay)
```

will succeed with `?okay` bound to "true" when no assertion has been made regarding `item-45` working, but the query:

```
(works-okay ?item true)
```

will not succeed if all known items are broken. It would be unreasonable for the memory system to invent a new item name just because `works-okay` is "true" by default.

The algorithm for calculating default property values thus consists of the following steps at query time:

- The query pattern is matched against the database with free variables treated as existential quantifiers. If a direct match is found the query succeeds.
- Contradictions are then sought by looking for direct matches of everything but the pattern value. If there are matches off everything but the value, the query fails.
- If the pattern is not matched, it is split into its index and value portions. If there are unbound variables in the index, the query fails.
- If the index does not contain unbound variables, the appropriate default function is called with the index as an argument generating a default value.
- If the value part of the pattern matches the default value, the query succeeds and if not, the query fails.

### Query Failure and NOT

A final point with respect to default property values is what it means when a query fails. Because defaults are allowed only for property values, a failed query does not necessarily mean that its pattern is contradicted in the database. In some cases, it may mean that no matching pattern exists. For example, the query

```
(color item-23 ?color)
```

can never fail because `?color` will always match the default,

```
(color item-23 red)
```

may fail if `item-23` is known not to be red (or red is not the default), and

```
(color ?item blue)
```

will fail if there is no item in the database with the color blue.

The connectives used to construct queries combine the success and failure of pattern matches, not the truth or falsity of the patterns. Thus, `and`, `or` and `not` combine the success of separate simple queries. For example, the compound query:

```
(and (color item-23 red)
     (not (size item-23 big)))
```

will succeed when the pattern `(color item-23 red)` is matched in the database and `(size item-23 big)` is not.

Query failure due to contradiction is quite different from query failure due to no matching pattern. The first means that the query is known to be false in the world while the second means the query is not known to be true. The RAP's themselves must realize that both situations may occur and generate queries carefully.

### 2.2.3 Persistence of Belief

Assertions made to the RAP memory should not necessarily remain there without continuing support from the sensor system. For example, the robot may encounter a full fuel-drum at the warehouse and record that fact in its memory. The question is how long it should continue to believe there is a full fuel-drum at the warehouse. If the warehouse is very seldom used, it may be reasonable to assume the drum will remain there for a long time, but if the warehouse is very busy, it may disappear after only a short while.

The problem of finite assertion persistence has been recognized for some time in the literature, and different solutions have been suggested for different situations [Nilsson, 1980, McDermott, 1982, Dean, 1985, Shoham, 1986, Hanks and McDermott, 1987, Firby and McDermott, 1987]. The simplest solution is to assign each assertion type in the database a specific life-time. For example, assertions of type `full` might be assigned a life-time of 1 hour. If a fuel-drum is discovered to be full and the assertion

`(full item-23 true)` made, queries of the form `(full item-23 true)` will succeed for one hour. After the hour is up, the assertion would disappear from the database. The idea of default assertion life-times is attractive because of its simplicity, but in many situations is not realistic and results in too much lost information.

Consider the difference between leaving a twenty dollar bill on the subway and leaving it on the dresser. The different contexts suggest that the location of the bill be given different life-times in the two situations. The assertion `(location item-12 subway)` should have a very short life-time, perhaps 5 minutes, while the assertion `(location item-12 dresser)` should have an essentially infinite life-time, assuming the robot lives alone. It is possible to construct a database that automatically assigns life-times to assertions based on context, but the database and the necessary rules become very complex. An alternative is to label each assertion with its time of assertion and have the database determine the truth of each fact at query time. The RAP memory system uses a persistence scheme of this type.

### Assertion Timestamp vs. Probability of Belief

There are two similar, but different, ways that fact persistence might be varied at query time: the use of *timestamps* and *probabilities*. Both schemes would use a query-time construct we will call `believe`. The term "believe" is used because the construct is used to extract information on how much faith should be put in a particular memory assertion, *i.e.* how strongly it should be believed. The `believe` construct takes the basic form:

`(believe` *assertion degree*)

which succeeds if the *assertion* is believed with force *degree* in the database. Using probabilities to vary persistences would put a probability in the degree, while using timestamps would put an interval of time in the degree. The first of these would define the construct to succeed when the assertion follows from the database with at least the given probability. The second form defines the construct to succeed if the assertion was last made within the given interval of time.

The use of probabilities in the `believe` construct requires a complex, active database that contains rules specifying the way assertion probabilities vary with time

and the acquisition or alteration of related facts. For example, deciding whether a fuel-drum at the warehouse is full with probability 80% requires the database to model the types of events that might empty a fuel-drum in the warehouse and the likelihood that such events will have taken place since the drum was actually measured to be full [Hanks, 1989]. On the other hand, the use of temporal intervals in the `believe` construct moves contextual knowledge out of the memory and into the hands of the inquirer. Rather than asking whether a $20 bill is in some location with a 90% chance, one asks whether its location assertion was made within the last 5 minutes if it is on the subway, or within the last year if it is on the dresser. The timestamp model is a special case of the full-blown probability model, with the problems of context management outside the memory.

The RAP memory database adopts the timestamp method of assertion persistence because it is makes the database simpler. The timestamp model moves the knowledge required for context evolution out of the database and into the RAP descriptions themselves. Different RAPs can decide for themselves how long an interval to use when checking the truth of an assertion. Short intervals are required for high confidence that the fact is true, while a long interval can be used when a lower confidence will do for the task at hand.

### The BELIEVE Construct

The believe construct is the workhorse of the RAP memory database query system. The basic form of the construct is:

(`believe` *assertion time-interval*)

where *assertion* is a single assertion and *time-interval* is a length of time. The semantics of the construct, when used as a query, is to succeed if the assertion was last added to the database within the time-interval specified. If the assertion is older than time-interval, or if the assertion has never been made, the construct will fail. Should the assertion contain free variables, they are treated as existential and bound in such a way as to make the query succeed, if possible.

Only simple, non-nested assertions can appear in a believe predicate. More complex queries are constructed as combinations of believe forms. For example, to find

out whether the database has heard about a big, red box within the last 10 minutes, the following query might be used:

```
(and (believe (class ?item box) 10)
     (believe (size  ?item big) 10)
     (believe (color ?item red) 10))
```

The believe form can also be used to determine exactly when an assertion was made to the database. When the time-interval argument is replaced with a variable, the believe form will succeed if the assertion involved has ever been made, and the variable will be bound to the time-interval since it was asserted. For example, to see if the level in a fuel-drum was measured 30 to 45 minutes ago, the following query can be made:

```
(and (believe (quantity-held item-12 ?amount) ?time)
     (> ?time 30) (< ?time 45))
```

## 2.3 Long-Term Memory

Item descriptions in long-term memory record what has been learned about the world in the past. However, the RAP memory must address a complex naming problem in its attempt to maintain a coherent long-term memory: when the sensors cannot recognize an item as one that has been seen in the past, how can the memory prevent the generation of multiple descriptions? The only solution is for the memory to recognize previously encountered items itself.

The behavior that the RAP long-term memory is designed to capture is illustrated in the following example. Suppose the robot goes to the fuel-depot and finds three identical fuel-drums there. The robot looks inside each one and finds that two are empty and the other is full of fuel. The robot then puts a bundle of papers into the empty drum on the right. While the robot remains in the area, there is no difficulty keeping track of which drum the paper is in. Later, the robot leaves and doesn't return until the next day to again find three fuel drums it cannot tell apart. The robot looks into one fuel drum and finds it full of fuel and looks into another and finds it empty. What should it conclude? We would like it to conclude that the

full and empty drums are the same ones it saw yesterday, and furthermore, that the remaining drum contains the bundle of paper. Most importantly, we do not want the robot to have no idea where to look for the bundle of papers.

### 2.3.1 Issues in Long-Term Memory

The algorithm for identifying old item descriptions with new ones illustrated by the above example is based on heuristic assumptions that are good in some situations but not in others. For example, if I put a quarter in my pocket and later reach in and take a quarter out, I am justified in assuming it is the same quarter. On the other hand, if I deposit a quarter in the bank and later withdraw a quarter, making the same assumption seems less appropriate. The RAP memory system makes the strong assumption that items previously encountered in a situation will be encountered again in the same situation without significant change. When situations are completely under the robot's control, such item continuity can be guaranteed, but in other situations it is only a guess.

When item descriptions are assumed to persist unchanged, mistaken identifications between local and long-term memory descriptions will be inevitable. However, the price of making such mistakes is kept fairly low by merging matched descriptions carefully. The most important point is to maintain timestamps on item properties when descriptions are merged: each property should keep the time it was last actually measured. That way, even when very old descriptions are matched to newly detected items, the properties brought forward from long-term memory will remain old and exert an appropriately small effect on subsequent behavior. By preserving assertion timestamps when merging descriptions, believe queries will still return appropriate information about merged properties. Furthermore, RAPs can often take matching errors into account. For example, if it is important to pick up a full fuel-drum from the warehouse and the warehouse contains some full and some empty drums, the RAP can include a sensor check to confirm that the drum selected is actually full.

#### Expectation Sets and Item Matching

To facilitate item matching, long-term memory is divided into *expectation sets*. An expectation set is a list of those items that are expected to be accessible to the robot

in a given situation. When the robot moves from one situation to another, local descriptions that become inaccessible are placed in the expectation set for the old situation. When the old situation is encountered in the future, its expectation set will predict those items to be present. As new item descriptions are constructed, they are checked against the predicted items to see whether they have been seen before.

The delivery truck domain naturally partitions item descriptions into sets based on location and class. All items at the same location are equally accessible to the truck. The most primitive sensing operations in the domain also deliver an item's basic class: fuel-drums are recognized as fuel-drums, enemy-troops as enemy-troops and so on. Thus, it is natural to partition long-term memory into expectation sets consisting of all items of the same class at the same location. For example, there will be an expectation set containing all fuel drums at the warehouse, another for all fuel drums in the cargo bay and another for the fuel drums inside a box. When the truck arrives at the warehouse and encounters a fuel drum, the drum's local-model description need only be compared with past descriptions from the warehouse/fuel drum expectation set.

As a local description is constructed, it will often be consistent with many different items in its corresponding expectation set. For example, a fuel drum found at the warehouse might be any one of the fuel drums seen there in the past. However, if the new fuel drum is found to be red it will probably match fewer candidates and if it is found to be full it will match fewer still. To determine how well two item descriptions match, a matching function is required. This function must take two item descriptions and return a score measuring how well they match. A high score means a good match and a low score means a poor match.

To actually identify a local description as matching an expectation set member, there must be a unique maximal match between it and the member's candidate group. A candidate group consists of all those item descriptions within an expectation set that are *identical*. A local description will have the same matching score for every member of a group, and if that score is higher than any other match within the expectation set, then the local description can be merged with a member of the group. Once a local description is matched with a member of the expectation set, it no longer participates in matches. This ensures that a local description is only matched with a single past description.

Matching local item descriptions with long-term memory descriptions is made slightly more complex by the fact that expectation sets must be *layered*. A layer is a set of descriptions that cannot be merged with one another because they are known to have been separate items in the real world. When an expectation set becomes inaccessible, local item descriptions that also become inaccessible are moved into the set. The difficulty is that the local descriptions might correspond to descriptions already in the set that they have not yet matched. To keep things straight, expectation sets are maintained in layers, where all of the descriptions within a layer are for different items, but descriptions across layers may apply to the same item. Layers are generated when the information about new items is not enough to make identifications with past descriptions possible before the robot moves on. For example, on one visit to the warehouse, a red and a blue fuel drum might be seen, and on the next visit, a full and an empty fuel drum. The result is an expectation set consisting of two layers, one containing red and blue fuel drum descriptions, and another containing full and empty drum descriptions. The red and blue drums can't be the same and neither can the full and empty drums. However, the red one may be full or empty and so may the blue.

### Database Queries and Expectation Sets

An important question that arises when expectation sets are added to memory is how the RAP interpreter's view of the memory as a database is affected. First, when an item description moves from the local model to long-term memory is that its sensor name disappears. To allow RAPs to determine whether an item has a current sensor name, the memory maintains the special predicate:

(`know-sensor-name` *memory-name*)

In a query, this predicate is satisfied if the sensor name of the item involved is known (*i.e.*, the item is known to be accessible) and is unsatisfied if the item is inaccessible. Other properties in the item description remain the same and it can be queried just as before. The situation becomes more complicated, however, when the item is encountered again.

When an item becomes accessible for a second time, the sensors assign it a new name and it is introduced into the memory again. At this point, there is no way for

the memory to know that the item has been seen before, and therefore, it generates a new memory name and begins to construct a new description in the local model. Now the memory contains two separate descriptions for the same item, and until they are matched and merged, queries will return answers that have to be considered carefully. For example, suppose a box is encountered at the factory, given memory name `item-45` and found to be big and red. Later, the robot returns to the factory and encounters the same box, giving it memory name `item-98` but does not immediately check its color. At this point, the query:

```
(and (class item-45 box)
     (location item-45 factory)
     (color item-45 red))
```

will succeed, but the query:

```
(and (class item-98 box)
     (location item-98 factory)
     (color item-98 red))
```

will fail even though `item-45` and `item-98` correspond to the same real item (assuming that `red` is not the default color for boxes).

This problem is not as serious as it might seem, as long as RAPs are aware of it and make queries carefully. In particular, the `know-sensor-name` predicate can always be used to ensure that a query is about a real, accessible item. Furthermore, if a RAP finds that an item is predicted to be at a location, but the item does not have a sensor name (like `item-45` in the example), a method can be chosen that generates sensor operations to aid the identification and merging process. The only thing that cannot be done properly is counting. There is no way to tell whether there are one or two boxes in the factory in the above situation. A robot that must count boxes will have to actually go to the location involved and count items with valid sensor names. However, in a dynamic world the number of boxes at the location cannot be known with certainty unless they are checked anyway.

The rest of this section discusses the algorithm for matching item descriptions across expectation sets. Matching descriptions between the local model and one expectation set is described first, followed by mechanisms for handling the complexities introduced by expectation set layers.

### 2.3.2 Matching Item Descriptions

When matching one item description against another, there are three main considerations:

1. There must be a way of comparing the two descriptions. The comparison must be based on counting assertions the descriptions have in common, and must take into account the fact that some contradictory assertions will veto a match while others will be the result of normal change.

2. An identification between a local description and an expectation set member should take place only when the local description matches that member better than any other. Local-model descriptions get more detailed over time and, in their early stages, are likely to match every member of a expectation set.

3. After a local description has been identified with an expectation set member, the two descriptions must be merged. Setting memory symbols equal is not enough, contradictions must be sorted out as well.

Consider the situation shown in Figure 2.3-a. The robot has visited the warehouse once in the past and noted two boxes there: a big red one and a little red one with some chocolate bars inside. The robot now returns to the warehouse and discovers that it contains two boxes. The local model holds the descriptions of two boxes: `item-45` with sensor name `box-1` and `item-46` with sensor name `box-2`. The expectation set for boxes at the warehouse holds the previously seen descriptions, `item-12` and `item-17`. What kind of matching should take place? At the moment, both local descriptions match the past descriptions equally well, but it would be a mistake to match them up arbitrarily. Suppose the robot now does a measurement and discovers that both boxes are empty as in Figure 2.3-b. At this point, both local descriptions are consistent with `item-12`, but neither appears consistent with `item-17`. However, depending on what the robot knows about the warehouse and boxes of chocolate bars, it may be perfectly consistent that someone else came in and took the chocolate bars out of the box. If so, both local descriptions still match both expectation set members. If the robot now does some sensing and discovers that `box-1` is big and `box-2` is small, matching is no longer ambiguous, and `item-12` and `item-45` can be

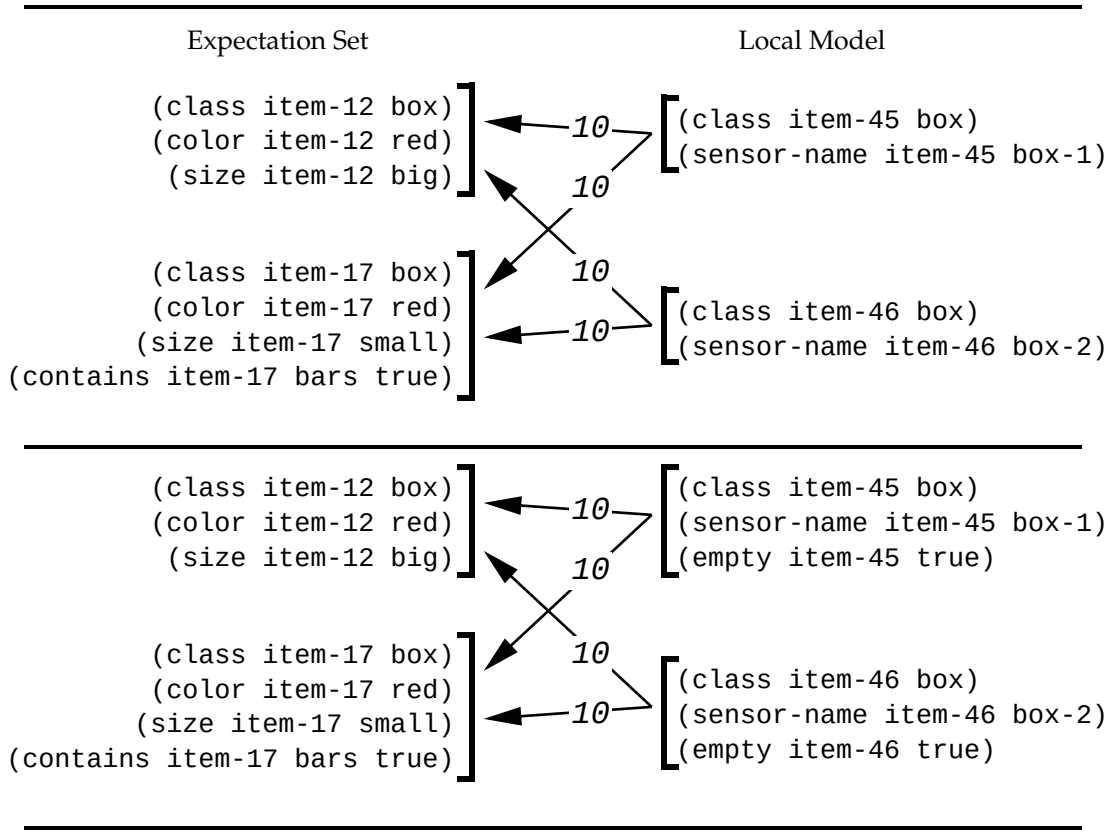



**Figure 2.3**: A Simple Matching Example

identified as can `item-17` and `item-46`. When merging their properties, the assertion about `item-17` holding chocolate bars must be discarded, and both `item-45` and `item-46` should inherit assertions about being red.

The process of discovering the correspondence between local and long-term memory item descriptions is designed to remain internal to the memory database so that RAPs do not have to worry about which descriptions fit which real items. The basic algorithm to accomplish automatic matching is:

- A robot action generates some sensory data about a property of some item in the local model. The local model item description is updated accordingly.

- Whenever the local model is updated, the memory system automatically checks to see if the altered description maximally matches a member of its corresponding expectation set.

- If a maximal match is found, the local item description and the matching expectation set member are merged. This removes both descriptions from further matching procedures. The local model and expectation set have now changed and they are checked again to see if any new maximal matches exist. This process repeats until no more identifications are found.

- If no maximal match is found, everything is left alone until more sensor data comes in.

To determine how well new and old item descriptions match, a context-sensitive matching function is used.

### The Item Matching Function

It might seem that counting assertions in common and ruling out matches with contradictory assertions would be enough for comparing item descriptions. However, as shown in the example above, real items generally have both permanent and transient characteristics: a box is a box but chocolate bars may come and go. Permanent characteristics must not contradict during a match but transient characteristics can be allowed to change. Unfortunately, the properties of an item which are permanent and those which are transient depend heavily on the item and context. For example, a box's color is a fairly permanent property that serves as a good identifier in most situations. However, taking a box to the painter's to have its color changed should not rule out matching the newly painted box with its past descriptions.

Matching functions are used to allow the necessary flexibility in comparing item descriptions. A matching function returns a high number for a good, complete match, and a low number for an incomplete or contradictory match. The RAP memory defines a matching function for every expectation set. For example, there might be a function for comparing boxes at the warehouse and a different one for comparing boxes at the painter's. The warehouse function would only score descriptions as matching if the box color remained the same, while at the painter's, box descriptions would match even if the color changed between visits.

The only arguments needed by RAP matching functions are the two item descriptions to be compared. Each function checks properties that must not contradict and,

if contradictions are found, scores the match zero. If the descriptions being compared do not contradict, each property shared by the two is assigned a score based on its ability to distinguish items of that class, and the scores are totalled. For example, the expectation set for boxes in the warehouse could be assigned a matching function that ignores contradictions between `contains` and `empty` assertions — scoring `empty` matches zero and assigning other matching assertions 10 points. Given the example from the last section, this function would result in `item-45` and `item-46` both matching `item-12` and `item-17` with score 10 in Figure 2.3-a and still matching both with score 10 in Figure 2.3-b. Once the robot discovers that `item-45` is big and `item-46` is small, however, `item-45` would match `item-12` with score 20 and `item-17` with score 10, while `item-46` would match `item-17` with score 10 and `item-12` with score 20. The difference in scores would allow `item-45` it to be identified with `item-12` and `item-46` to identify with `item-17`.

The syntax of matching functions, and their assignment to item classes, is described in more detail in Appendix A. The matching functions used in the delivery truck domain are also described there.

### Finding a Maximal Match

Scoring functions by themselves are not enough to match item descriptions. The problem is that sparse local descriptions will share all of their properties with many members of the expectation set and hence, match each with the same score. In the example above, when the two boxes become newly accessible in the warehouse, each matches the two past descriptions with a score of ten based on the `class` property alone. However, the two past descriptions are quite different and to identify either with a new local description would be premature. The trick is to make identifications as soon as possible, but not so early as to ignore salient differences.

The approach used in the RAP system is to divide the expectation set into candidate groups containing item descriptions that are *identical*. When a description shares its properties with several candidate groups, it is consistent with each group but identification is delayed until more information places it in one group or another. The more properties that a description and group have in common, the more likely it is that they correspond. Only when a description matches a single group with a

higher score than any other is there reason to believe it is a member of only that group. Therefore, identification waits for such a maximal match. Since candidate group members are identical, they are also indistinguishable and there is no need to worry about which one the local description should be identified with: any member will do equally well.

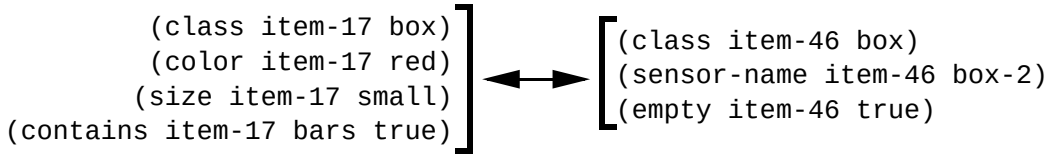
Consider the two-box example once again. The two past descriptions, `item-12` and `item-17` are not the same (they differ in contents) so they form two separate candidate groups. When the real boxes, `item-45` and `item-46` become accessible, they match both candidate groups with a score of 10. However, identifications cannot be made because `class` alone is not enough to distinguish between the past descriptions. The descriptions will always match each other this well. Adding new facts to the local descriptions of `item-45` and `item-46` will not change their `class` and hence cannot diminish their matching scores with `item-12` and `item-17`[4]. When the additional fact that both boxes are empty is discovered, their matching scores stay the same because the matching function ignores `empty` assertions. However, when `item-45` is learned to be big, it matches `item-12` with score 20 and `item-17` with score 10. A maximal match now exists between the two and `item-45` and `item-12` can be identified. This takes `item-12` out of the running, and `item-46` matches `item-17` with a score of 10 and there are no other candidate groups. Another maximal match thus exists, and `item-46` and `item-17` can also be identified.

### Making Items Equal

Once two descriptions are identified as corresponding to the same real item, they must be merged. In the example above, the local description for `item-46` is identified as corresponding to long-term memory description for `item-17`. This means that `item-46` and `item-17` refer to the same item out in the real world and their descriptions should be merged to reflect that fact. One way to merge them is simply to assert that the two symbols, *i.e.* `item-46` and `item-17`, are equal within the database. However, that is not really enough because the descriptions may contain contradictory properties.

[4] An exception to this is when contradicting facts are discovered. For example, if size is is treated as a permanent box fact and `item-46` is learned to be big, its matching score with `item-17` may be reduced to zero by the matching function.

The RAP memory merges descriptions by erasing any assertion from either description that is contradicted by a *newer* assertion from the other. This process preserves the newest information known about the item in question while keeping the memory database consistent. For example, after identifying the local description `item-46` as the item previously encountered and named `item-17` the following descriptions need to be merged:




All assertions about `item-46` are newer because they were noted during the current encounter with the item, and the `empty` assertion contradicts the earlier belief that the item `contains` chocolate bars. Hence, merging the two produces the description:

```
(= item-46 item-17)
(class item-46 box)
(sensor-name item-46 box-2)
(color item-17 red)
(size item-17 small)
(empty item-46 true)
```

The memory database now states that the accessible item with sensor name `box-2` is a small, red, empty box. Notice that the conclusions that box is red and small come from the identification of the two descriptions and they have not been directly sensed during this encounter with the item.

### 2.3.3 Manipulating Expectation Sets

The algorithm for matching items in the local model with those in long-term memory is fairly straightforward and produces a memory database that keeps item descriptions consistent with new sensor data. However, the simplicity of the algorithm is complicated by several additional situations that can arise. First, items may become inaccessible to the robot before their local descriptions can be maximally matched with corresponding expectation set members. This produces multiple descriptions of the same item that must be dealt with somehow. Second, the robot may act on an item in the local model so as to move it from one expectation set to another. This

means the item's previous description must be removed from its expectation set, even when that description has not yet been identified. Finally, problems may arise when sensors return the cardinality of a local model item type. When the sensors report that there are exactly two fuel-drums in the warehouse the local model will show two descriptions. However, the expectation set for fuel-drums in the warehouse may contain some other number of descriptions. It must be adjusted to hold the correct number. This section discusses each of these difficulties in turn.

### Adding an Item to a Set

The first complexity in description matching is that expectation sets are divided into layers. A layer is a snapshot of the local model. Every time that an expectation set's situation becomes inaccessible, a snapshot of the model is taken and pushed into the expectation set. If a new snapshot is equivalent to a previous expectation set layer, meaning it contains exactly the same descriptions, it is discarded as redundant. However, if the snapshot contains descriptions not known to be equal to previous ones in the set, a new layer is formed. Thus, expectation set layers represent disjunctions; each layer contains a description for every real item in the local model, but "which is which" will often be unknown. This is the only form of disjunction allowed into the RAP memory, but it requires that all operations on the set be applied to each layer.

### Changing an Item's Properties

Expectation set layers result in a more complex algorithm than the one given previously for identifying new item-descriptions with old ones. Consider the expectation set sketched in Figure 2.4. This set describes what is known about fuel-drums at the warehouse. The figure shows no item properties, only item names, and the same name is used in different layers to show descriptions that have been identified as corresponding to the same real item and set equal. The local model contains three fuel-drum descriptions as does each layer in the expectation set. The first layer holds the descriptions of the fuel-drums from the robot's first trip to the warehouse. The second layer shows that on the next trip to the warehouse, three drums were again seen, and one was identified as being seen before, `item-34`. The other two drums, `item-65` and `item-78`, must correspond to the previous visit's `item-56` and `item-23`,

but presumably it could not be determined which was which without more sensing. Similarly, the third visit identified one drum as having been seen on the second visit, `item-78`, but did not have enough information to disambiguate the other descriptions. Finally, on the current visit, `item-65` has been identified as seen on the second visit as well.

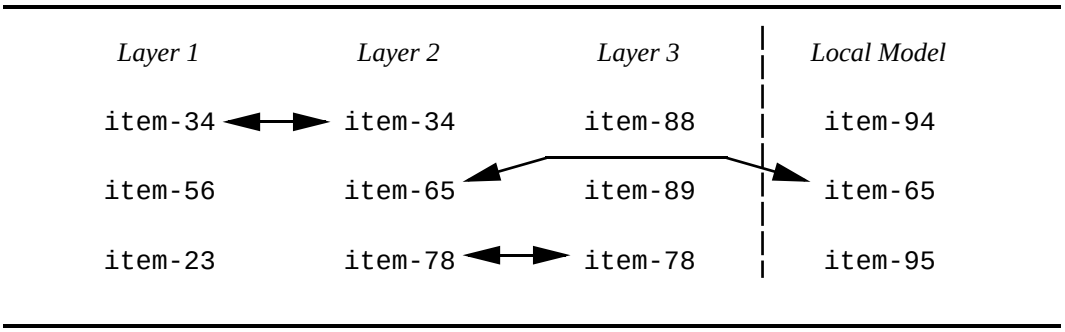



**Figure 2.4**: A Simple Expectation Set

Notice that items in the local model can be identified with expectation set descriptions in any layer. This follows from the fact that items within a layer must be distinct and that each layer describes the same set of items. The existence of layers, each of which contains a match for every local model description, means that the basic algorithm for discovering identifications must be generalized across all layers. Furthermore, when a layer is changed by an identification, it may become sufficiently restricted so that its members match the members of other layers and the whole process can cascade. Once all identifications that can be made have been made, there may be redundant layers in the set that can be removed. Each time that a new local model item property is sensed, the process must begin again. The resulting algorithm is:

- When a new item property is sensed, its local model description is updated appropriately.
- Each layer of the expectation set corresponding to the description is checked in turn from latest to earliest:
  - If the layer already contains a description that has been identified with the altered description, the layer is left unchanged.

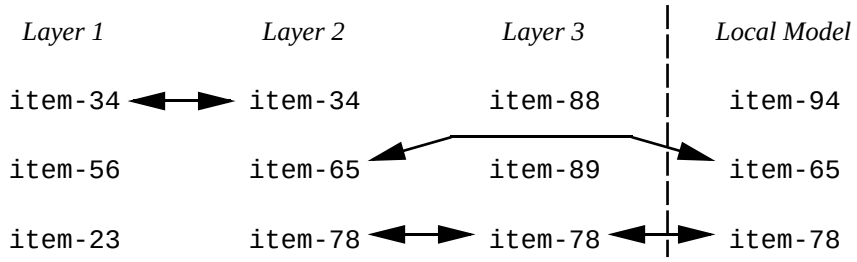
  - If the layer contains a maximal match with the altered description, the match and the description are identified. Maximal matches with descriptions that appear in other layers where the altered description already has been identified must be ruled out, because the description cannot appear in a layer twice.

  - If there is no maximal match, the layer is left alone.

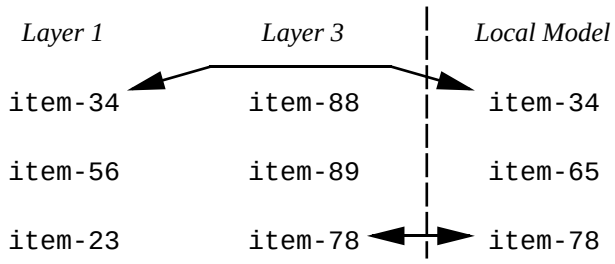
- If any layers are changed during the previous step, every member of the layer must be checked against every other layer for maximal matches. New matches may arise because of those that get ruled out by the description only being allowed once in each layer. Repeat until no more layers change.

- Check for redundant layers and take them out of the expectation set.

Consider the expectation set in Figure 2.3 once again. Suppose that a sensor action by the robot adds a new property to `item-95` in the local model and the first step in the algorithm described above finds that `item-95` maximally matches `item-78` in layer three. The two descriptions will be identified giving the new picture:




Assuming that `item-95` doesn't match any other descriptions, layers 2 and 3 are examined next because they have been altered. We will assume that no new matches are discovered by checking the other items in layer 3, but when layer 2 is checked, a maximal match is found between `item-34` and `item-94` in the local model. This new match results from the fact that `item-34` cannot be `item-65` or `item-95`. Since `item-34` and `item-94` have been identified, layer 2 and the local model are identical and layer 2 can be dropped, resulting in:



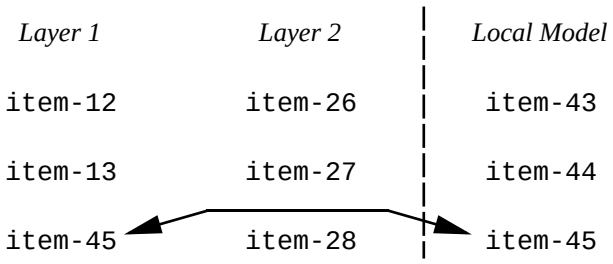

Everything that can be done now has been done and the matching process will wait until new information comes in from the sensors.

Every time that a new piece of sensor information comes into the memory and alters a local model description, this algorithm for discovering identities must be executed.

### Moving an Item Out of a Set

New information about an item will not usually change the expectation set it belongs to, but an item description must sometimes be moved from one expectation set to another as a result of the robot's own actions. In the delivery truck domain, the robot can move items from place to place with its arms, and each time the item changes location it moves from one expectation set to another. For example, a red box may be encountered in the warehouse where several other red boxes have been seen previously. If the robot picks up the box and moves it to the factory, its description (and its unmatched descriptions in various expectation set layers) must be moved from the box/warehouse expectation set to the box/factory expectation set.

If the item to be moved from one expectation set to another has a single description, the problem is quite simple. The local description of the item is moved from the old set to the new one, and all copies of the description (including equality) are removed from the old set. This reduces the membership in every layer of the old expectation set by one and the item will no longer be predicted by the set. However, if there are multiple copies of the item description, because it has not yet been identified in every layer of the old expectation set, removing it is quite a bit more complex. A description must still be removed from every layer in the old expectation set, but figuring out which one to remove from each layer is a problem. Consider the situation below:




If `item-45` is moved to another location, it must be removed from this expectation set and placed in a new one. Changing the expectation set associated with the local model description of `item-45` is simple, as is deleting it from the first layer of the expectation set. However, `item-45` must match one of the items in the second layer and that item must also be deleted. The question is: which one?

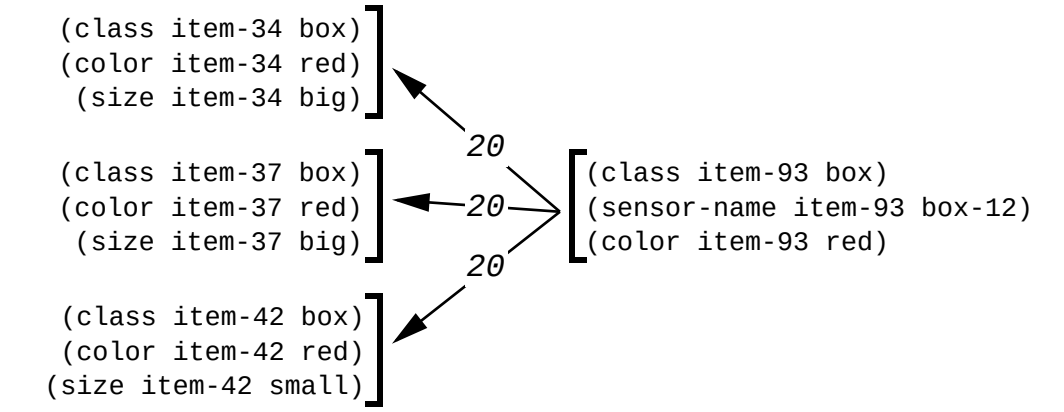



**Figure 2.5**: Removing an Item From an Expectation Set

To understand the answer, consider the single layer situation in Figure 2.5. The item to be removed from this expectation set is `item-93`. We will ignore all other members of the set and assume that `item-93` might be only `item-34`, `item-37` or `item-42`, previously seen red boxes. When `item-93` is removed, though, one of these past descriptions must go as well. Unfortunately, there is no way to decide between them and removing one at random could make the memory very wrong. For example, removing `item-42` from the expectation set would mean that all future red boxes encountered would be assumed to be big. It is better to explicitly represent the information that becomes unknown as a result of removing `item-93` and leave the database correct but more uncertain.

The algorithm for figuring out the smallest set of assertions that must become

unknown in the database as a result of moving an item from one expectation set to another consists of three steps.

1. A collection of descriptions is made by taking one description from every candidate group the item may belong to. In the example above, this collection consists of `item-42` and one of `item-34` or `item-37`, say `item-34`. Only one member from each candidate group need be considered because they are identical.

2. The collection of descriptions is forced into a single candidate group by making all properties not shared by every member in the collection unknown. Both `item-42` and `item-34` in the example will be assigned the property: `(size ... unknown)`. By explicitly labelling the properties "unknown", rather than simply deleting them, the database will accurately reflect the fact that they are now uncertain but defaults should not be used in their place.

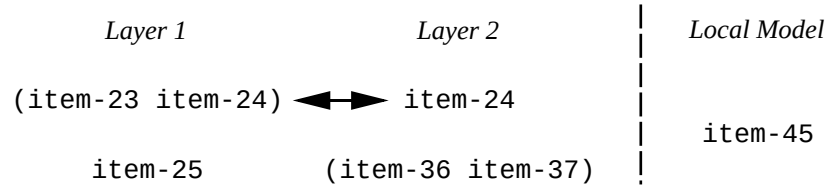
3. All descriptions in the collection are now indistinguishable and one is chosen at random to be deleted and merged with the item being moved, say `item-34`. This final identification will give the item being moved any properties shared by every description it was consistent with.

In the example, the final result is that `item-93` and `item-34` are merged into a single description and moved into another expectation set. The old expectation set is left holding descriptions for `item-37`, which is still a big red box, and `item-42` which is now a red box of unknown size. To see that this result is correct, imagine that the three original descriptions correspond to red boxes in a sack: two big ones and one small one. Someone lets you peek at a box, letting you see it is red but not letting you see its size. The person then reaches into the sack and takes out a box exactly like the one you were allowed to peek at. You now know that the sack contains one big red box (because both cannot have been taken out) and one red box that could be either big or small (because you did not get to peek at the size of the box removed).

An interesting example of what the above algorithm will do comes from the delivery truck domain. Consider a situation where the truck has ten blue fuel drums in a box: five are known to be full and five are known to be empty. The expectation set for fuel-drums in the box will hold two groups, one for full drums and one for empty,

with five members in each. The truck now moves one drum out of the box and into the cargo bay without looking at its color or check whether it is full or empty. To adjust the expectation set, the algorithm above says to first collect one description from each group; the remaining descriptions cannot have changed because the truck only moved one item. The collection will contain one full and one empty description and one of them must correspond to the drum actually moved. The next step in the algorithm is to force all descriptions in the collection to be identical. This step eliminates properties that distinguish items in the collection and results in two descriptions of blue fuel drums with unknown contents. One of these corresponds to the drum moved, and one to a drum still in the box — either a full or empty drum was moved out, leaving an extra full or empty drum in the box. The third step in the algorithm is to take one description out of the collection and merge it with the local description of the drum moved so the moved drum will be known to be blue. The truck will now believe there are four full and four empty blue fuel drums in the box, one blue fuel drum with unknown contents in the box, and another in the cargo bay. If the truck moves four more fuel drums out of the box without looking at them, it will now longer know the contents of any of the drums — but it will still know they are all blue.

This algorithm for moving an item out of a one layer expectation set is made substantially more complex by the addition of more layers to the set. One would like to apply the algorithm separately to each layer in turn, but that doesn't work. The problem is that candidate groups in one layer may share members with other layers. If the layers that share members are treated separately, either too many or too few descriptions may end up being merged. Consider the expectation set fragment below:




If `item-45` is to be removed from the expectation set and we try to apply the simple algorithm to each layer in turn, `item-24` and one of `item-36` or `item-37` will be merged in layer 2. At the same time, `item-25` and one of `item-23` or `item-24` will be merged in layer 1. The problem arises if `item-23` is chosen in layer 1 because `item-24` could have been chosen from both layers. The merging of `item-23` unnecessarily will

result in an unwarranted loss of information.

To merge the fewest number of descriptions possible (and hence make the fewest item properties "unknown"), the algorithm for removing an item must be applied to all layers at the same time. First, a collection of descriptions is selected from the expectation set, such that one description from every candidate group in every layer is included, but the collection itself holds as few descriptions as possible. In the example, this would be `item-24`, `item-25` and one of `item-36` or `item-37`, say `item-36`. Next, as with a single layer, these descriptions are merged into one candidate group and unshared properties are set to "unknown". Finally, the item being removed must be identified with members of the collection so they can be taken out of the expectation set. With a single layer, one member of the collection could be chosen at random, but with a collection chosen from multiple layers, members must be chosen so that one, and only one, comes from each layer in the expectation set. When these descriptions are removed, every layer in the expectation set will lose exactly one member as required. With the further provision that the descriptions chosen form the smallest group possible, the least information will be lost from the expectation set. In the example above, `item-24`, `item-25` and `item-36` get collected and merged, but only `item-24` gets removed. The final result is that `item-23` is unchanged in layer 1, `item-37` is unchanged in layer 2, and `item-25` and `item-36` become identical except that `item-25` stays in layer 1 and `item-36` in layer 2.

### Determining the Cardinality of a Set

The final piece of information the memory system can learn about an expectation set is its cardinality. In particular, the sensor system in the robot truck domain will report that the items currently in the local model are all of the items that are accessible. Since local model descriptions are each associated with an expectation set, the total number of accessible items in each set is known once all accessible items are known. At that point, the number of items actually accessible can be compared to the number of descriptions in each layer of each set, and adjustments can be made to keep expectation set predictions up-to-date. If more real items are accessible that predicted by an expectation set, new descriptions must be added to the set, and if fewer items are accessible than predicted, descriptions must be removed.

Adding new descriptions to an expectation set is very easy and is done whenever the sensor system introduces a new item to the local model. When the item first becomes accessible, the memory system puts it in the local model and looks up its associated expectation set. If the new item increases the number of items associated with the expectation set beyond the number in each layer of the set, a new description with no properties is added to every layer in the set. This description will eventually be matched to the unpredicted item and take on the correct properties. Whenever an expectation set layer is generated, or an extra item is associated with the set, the correct number of empty description is added so that all layers always hold the same number of items.

Decreasing the number of items in an expectation set is also straightforward once the machinery described in the last section for removing an item is in place. When the sensors declare that all accessible items in a set are known, and the number of items in the local model is smaller than the number in each set layer, some descriptions must be removed from the set. Removing a description in this case is exactly like removing a description when an item is moved from one expectation set to another. An empty item is added to the local model and associated with the expectation set to be reduced. This item is then moved to a null expectation set. The algorithm for moving an item removes exactly one description from every layer in the original expectation set, while minimizing the information lost, which is just what we are looking for.

## 2.4 Summary

Situation-driven robot control relies on the ability to accurately assess the situation surrounding the robot at all times. Ideally such an assessment would come from constant, direct sensor observation of the entire world. In practice however, execution systems like the RAP system must face the fact that real sensors have a limited bandwidth and range. Limited bandwidth means the robot can examine only a few items in the world at any one time, and limited range means that items outside the local area cannot be sensed at all. Therefore, if the RAP system is to base its action choices on a reasonable picture of the larger world, it must maintain a memory of past sensory information. This memory acts like a buffer between the RAP decision

procedures, which want a stable view of the whole world, and the sensors, which move from item to item examining only a small part of the world at a time.

In addition to robot sensors having limited bandwidth and range, they cannot uniquely identify every item in the world. We assume that when an item is encountered it is assigned an identifying name by the robot hardware, and that name remains unchanged as long as the item remains accessible. However, as soon as the item becomes inaccessible the sensor name is lost, and if the item is encountered again it will be assigned a different name. We call this the local identification assumption.

The local identification assumption, along with the robot's need to remember information about items seen in the past, requires the RAP memory system to recognize items when they are seen again. When items are encountered the memory system begins to build descriptions of them and if new descriptions cannot be matched with old descriptions of the same item, the memory will become cluttered with duplicate descriptions and rapidly become meaningless. Recognition is the process of comparing a new item description with an old one and identifying that they both correspond to the same real item.

Recognition in the RAP memory is based on the assumption that items will remain pretty much unchanged at the location where they were last encountered. The items seen in a situation thus form a set of items to be expected when the situation is encountered again. As new items are sensed they are compared to items from those expectation sets to see if they have been seen before. When a match is found, the description of the expected item and the description of the new item are merged to form a single record of the item in memory. Matching is mediated via context-sensitive functions and proceeds automatically within the RAP memory.

An important note is that recognition will sometimes be mistaken. The RAP memory recognition algorithm assumes that the same items seen in a situation will be encountered there again. In the real world this may or may not be true. Sometimes items really will remain unchanged and be recognized without problem. In other situations, items may be moved, changed, or substituted by other agents while the robot is gone. Under these circumstances, the robot may mistakenly identify substituted items with the items that were there before. The rest of the RAP system must be aware that such mistakes can be made and generate appropriate actions when mistakes cannot be tolerated. The important thing is that mistakes are not a

result of the RAP memory assumptions, but are inherent in the process of recognizing items. People have this same problem when faced with indistinguishable items, like fuel-drums or boxes that all look the same. The particular errors the RAP memory makes are a result of its particular algorithm, but no algorithm can be perfect because the world is just too uncertain.

Automatic item recognition in the RAP memory allows the rest of the RAP system to treat memory as a simple database of item property assertions and the rest of this thesis takes item recognition for granted. The only concession to item recognition will be that RAPs will sometimes have to contain sensing operations specifically designed to gather information to ensure that recognition will take place. For example, when a diamond is in a box and the robot wants the diamond, the robot must gather enough information to recognize the box when it sees it, and then will have to check inside to make sure that the recognition is correct.

The next chapter moves on to discuss task generation and scheduling. A sketchy plan will often leave tasks unordered, and execution will generate new tasks and actions to perform. The RAP interpreter coordinates these operations and decides which task to work on at any given time. The next chapter also describes primitive actions and the way primitive action execution affects the contents of memory.

# Chapter 3

# The RAP Interpreter

The RAP situation-driven execution system has two major components: the RAP interpreter, including the memory and task agenda, and the RAP library from which task definitions are drawn. Tasks might best be thought of as instantiations of RAP defined programs that are to be executed by the interpreter. This chapter discusses the problems involved in designing the interpreter so that RAP-defined tasks behave in the desired situation-driven manner.

Recall the description of the RAP execution system illustrated in Figure 3.1. An initial sketchy plan is given to the system and combined with RAPs from the library to form tasks that are placed on the RAP task agenda. The RAP interpreter selects tasks from the agenda one at a time and executes them. Execution consists of either spawning new subtasks to add to the agenda, or issuing a primitive action to the robot hardware.

There are two major algorithms to discuss with respect to the RAP system: the RAP interpreter for executing RAP defined tasks, and the hardware/memory interface which handles primitive action results. These two subsystems are quite separate — the interpreter queries the memory and issues primitive action requests, but it plays no part in the analysis of primitive action results or in the resulting memory changes. The hardware/memory interface, on the other hand, is continually altering memory in response to new information about the world, but does not have any control over robot behavior. The algorithms behind each of these processes is described below.

Unfortunately, to understand the interpreter algorithm, some notion of the RAP

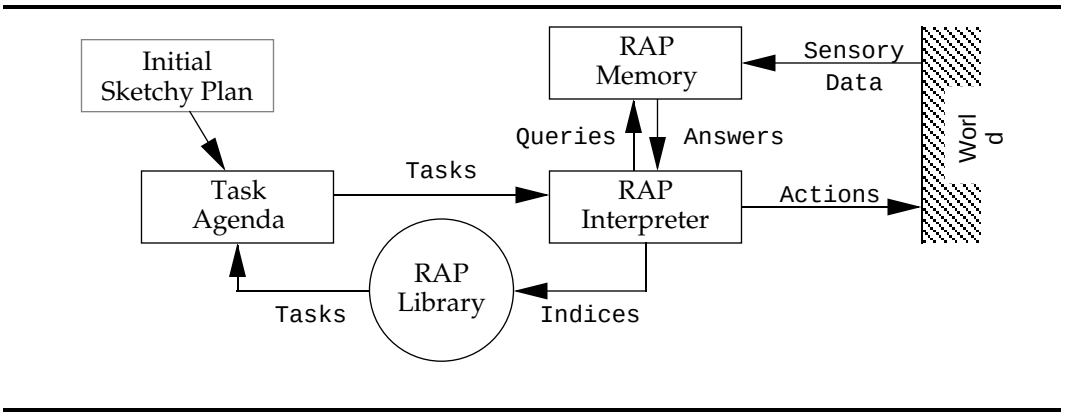



**Figure 3.1**: The RAP Execution System

syntax is required and vice versa. Therefore, this chapter introduces the basic RAP syntax before concentrating on the interpreter required to execute it. The next chapter uses the resulting description of the interpreter as a basis for describing the complete RAP syntax and the behaviors it enables.

## 3.1 The Basic RAP Syntax

Every execution goal given to the RAP system, either as part of a sketchy plan or part of a method expansion is instantiated as a task before it is processed. The representation used to describe task instantiations is the RAP. As goals are introduced into the system, each is used as an index into the RAP library and the RAP indicated is instantiated to create a task for the goal. There may be several active instantiations of the same RAP when several goals of the same form are active simultaneously. Many goals require complex behavior such as loops, pauses, synchronization, or counting to fulfil them and the full RAP syntax described in Chapter 4 has such capabilities. This section introduces only the basic syntax.

A reactive action package is defined using a form with several different clauses. Each clause describes a different aspect of the instantiated RAP's behavior at execution time. An outline of the syntax is shown below:

```
(DEFINE-RAP
   (INDEX index)
   (SUCCEED formula)
   (METHOD one plan for carrying out RAP behavior)
```

(METHOD *another plan for carrying out* RAP *behavior*)
*... other methods*)

The INDEX clause is mandatory and identifies the goal that the RAP is designed to handle. The SUCCEED clause specifies the conditions that must be true for the RAP to have achieved its goal. These conditions are specified as formulae that must match assertions in the RAP database. Each METHOD clause describes a different way to accomplish the goal and contains subclauses which specify the steps in the method and the situations in which it is applicable.

### 3.1.1 The RAP Index

The index takes the form:

(INDEX (*name in-vars* => *out-vars*))

where the *name* is unique to the RAP being defined and can loosely be interpreted as the goal that the RAP is designed to achieve. The RAP name is used by both sketchy plans and other RAP method clauses to specify execution subgoals. An index clause may include both input and output variables. Input variables come first and output variables follow the special symbol =>. Input variables must be bound when the task described by the RAP is first selected for execution and output variables become bound when the task completes successfully.

For example, the following are possible RAP indices:

```
(INDEX (handle-low-fuel))
(INDEX (arm-pickup ?arm ?thing))
(INDEX (find-object ?object => ?place))
```

The first index might be used in a RAP designed to watch for the fuel level of the robot truck to get low. It doesn't need any arguments assuming that "low" is defined within the RAP itself. The second example is the most typical form for a RAP index and might be used in a RAP to pick an object up with either of the robot's arms. The third example might be used to describe a RAP designed to discover the location of an object (maybe by looking, asking someone, or reading a book) and return the location by binding it to the variable ?place once found.

### 3.1.2 The RAP Succeed Clause

The SUCCEED clause describes conditions that must be true in the world for the RAP to have properly carried out its intended behavior. The basic succeed clause syntax is:

(SUCCEED *formula*)

An executing task loops through its methods until its succeed clause is satisfied or none of its methods apply.

The formula is used as a query to the RAP memory and may contain free variables (*i.e.* variables not contained in the input variables of the RAP index). When the query is successful, the RAP is completed and will finish, preserving free variable bindings that correspond to output variables in the index.

### 3.1.3 Specifying RAP Methods

Every method in a RAP is represented by a separate METHOD clause which is further broken down into a CONTEXT and a TASK-NET or PRIMITIVE. The context specifies those situations in which the method is appropriate and the task-net or primitive describes the sub-plan that makes up the method. The basic syntax for a method is:

```
(METHOD
   (CONTEXT formula)
   (TASK-NET task network description))
```

or

```
(METHOD
   (CONTEXT formula)
   (PRIMITIVE primitive action request))
```

#### The Context Clause

The CONTEXT for a method is a formula that must be true in memory before the method is appropriate. At execution time, the RAP interpreter looks at each method's context and chooses one whose context is true. For example, a method that is applicable when `arm1` is holding something might have the context:

```
(CONTEXT (arm-holding arm1 ?thing)))
```

which will be satisfied only when there exists an object that `arm1` is holding.

When the context formula is matched against the RAP memory, free variables will become bound. These bindings are preserved and can be referenced in the contained task-net or primitive. In the example above, `?thing` is a free variable and after the match it will be bound to whatever object `arm1` is holding. The associated task-net or primitive can then perform operations on that object by referring to `?thing`.

#### The Primitive Clause

A method that contains a PRIMITIVE clause indicates that a primitive action request is the appropriate way to carry out the task. Each primitive action clause describes a single action request to be passed to the hardware interface. For example, `arm-grasp` is a primitive action for the simulated robot truck and the lowest level task used to pick something up contains the method:

```
(METHOD
   (CONTEXT   (and (know-sensor-name ?thing true)
                   (arm-at ?arm ?thing)))
   (PRIMITIVE (arm-grasp ?arm ?thing))))
```

This method is only applicable if the sensor name for the object to be grasped is known and if the arm is positioned appropriately. Note that the sensor name might not be known if the object is in an old expectation set and has not yet been matched to a currently accessible item. More sensing would be required in that situation.

### 3.1.4 Describing a Method Task-Net

The TASK-NET clause is used inside a method to describe a sketchy plan for carrying out the RAP's intended behavior. In general, a task-net will contain a number of steps to be executed. Each step is a separate entry and special annotations are used to order the steps with respect to one another, specify preconditions that one step sets up for another, and post metric temporal constraints between steps. The basic syntax for a task-net is:

```
(TASK-NET
   (step1-tag priority step-specification annotations)
   (step2-tag priority step-specification annotations)
   ...)
```

The *tag* in a task-net entry identifies the step within the task-net: each step must have a different tag. Each step also has an optional *priority* consisting of a single integer value. This value is added to the priority of the enclosing RAP to give a new priority for the step. The default priority value is zero which gives the task-net step the same priority as its enclosing (parent) RAP. Priorities are discussed in more detail in the next section.

A step *specification* is a pattern that must match a RAP index. The specification may contain variables, but they must be bound at the time the task being specified is executed. Input variables in the second or later steps may be unbound at the time the task-net is spawned, but they must become bound, as output variables in earlier steps, before execution of the step begins. The step specification indicates what the step is supposed to do.

Task-net steps are connected together to form a plan using *annotations*. Annotations specify temporal-ordering constraints, preconditions and metric temporal constraints. Annotations are discussed in the next chapter and only the simplest is given here:

(*formula* `FOR` *tag*)

The `FOR` annotation performs two functions: it specifies a task ordering constraint and it describes an execution-time precondition. The intended intuitive meaning is that the step being annotated is designed to generate the specified formula for the tagged step. For example, the task-net fragment:

```
(TASK-NET
   (t1 (arm-grasp arm1 ?thing)
       ((arm-holding arm1 ?thing) FOR t2))
   (t2 (arm-crush arm1 ?thing)))
```

states that the grasp step is included specifically to make sure that `arm1` is holding the object for the crush step. The implication is that the crush must follow the grasp,

and that if the thing is not being held when the crush begins, something has gone wrong. Hence, formulae that appear in FOR annotations become preconditions on the tagged steps. These formulae are an execution-time analog to traditional plan-time protections. One step is designed to set up a condition (specified by the formula) for another, and if the condition is violated then the second step ceases to be valid. Whenever a FOR annotation formula is found to be false at the beginning of the step it points to, the enclosing task-net fails.

### 3.1.5 An Example

As an example consider the RAP shown in Figure 3.2. This RAP defines a task that can be used to move an object from one location to another in the robot delivery truck domain. For instance, to move `box-1` to `factory-5` the goal `(move-to box-1 factory-5)` might be specified. When this goal is specified the RAP is instantiated to form a task with `?thing` bound to `box-1` and `?place` bound to `factory-5`. The task will only finish when the RAP memory contains the assertion `(location box-1 factory-5)` showing that the box is at the correct factory.

The first method in the RAP is applicable when the location of `box-1` is known. In such situations, the method's context will match the appropriate assertions in memory and `?loc` will be bound to the box's location: let's say `lot-45`. The task-net for the method specifies four steps that are totally ordered by the FOR annotations.

```
t0: (goto lot-45)
t1: (pickup box-1)
t2: (goto factory-5)
t3: (putdown box-1)
```

The FOR annotations also specify that when the pickup task comes up for execution the truck must be at `lot-45`, before heading off to the factory it must be holding the box, and it must be at the factory and holding the box when the putdown task is executed. If any of these facts are not true at the appropriate time, the method is assumed not to be working as intended (possibly because of a primitive failure or an interruption), and the rest of the method is abandoned. However, if all of the subtasks succeed, it does not mean that the task is finished. After a method completes, the task itself is brought up for execution again and the succeed clause is checked. If

```
(DEFINE-RAP
   (INDEX   (move-to ?thing ?place))
   (SUCCEED (location ?thing ?place))
   (METHOD
      (CONTEXT (and (location ?thing ?loc)
                    (not (= ?loc unknown))))
      (TASK-NET
         (t0 (goto ?loc)
             ((truck-location ?loc) for t1))
         (t1 (pickup ?thing)
             ((truck-holding ?thing) for t2)
             ((truck-holding ?thing) for t3))
         (t2 (goto ?place)
             ((truck-location ?place) for t3))
         (t3 (putdown ?thing))))
   (METHOD
      (CONTEXT (and (location ?thing unknown)
                    (not (truck-location warehouse))))
      (TASK-NET
         (t0 (goto warehouse)
             ((truck-location warehouse) for t1))
         (t1 (pickup ?thing)
             ((truck-holding ?thing) for t2)
             ((truck-holding ?thing) for t3))
         (t2 (goto ?place)
             ((truck-location ?place) for t3))
         (t3 (putdown ?thing)))))
```

**Figure 3.2**: A Simple RAP for Moving an Object

the method really did work as intended, this clause will be true and the task will complete, otherwise another method is chosen and executed.

The second method in the RAP is similar to the first but is applicable when the box's location is not known. This method states that it is a good idea to go look in the warehouse when an object's location is not known. If the truck is already at the warehouse and the box isn't there, neither of the RAP methods apply and there is no alternative but to give up. The RAP fails in the hope that its parent (the task that spawned it) will know how to deal with the problem.

The next section describes the way the RAP interpreter retries methods, succeeds

and fails tasks and keeps from getting caught in futile loops.

## 3.2 The RAP Interpreter

A task consists of a RAP description and variable binding information and is executed according to the algorithm shown in Figure 3.3. A task is chosen from the agenda for execution and its succeed clause is checked against memory. If the succeed clause is satisfied the task is finished, but if not, a method is selected for the task. The method generates subtasks that are added to the task agenda followed by the task itself, which returns there to wait for its subtasks to complete. A new task is then chosen from the agenda for execution and the process repeats.

As a task executes, it moves back and forth between the task agenda and the RAP interpreter according to the algorithm:

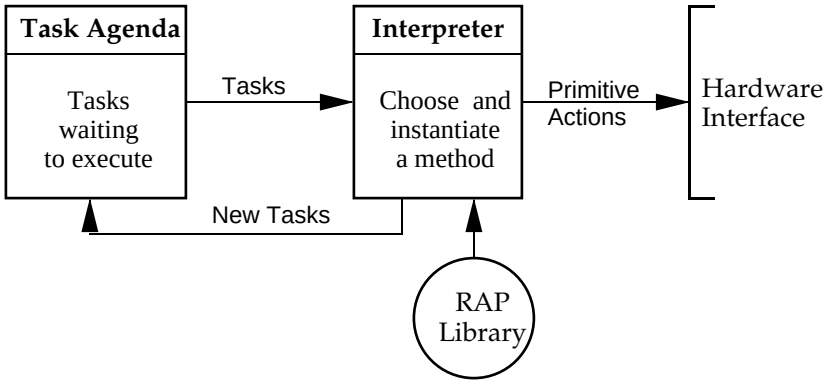
1. If the succeed clause is satisfied finish.

2. Otherwise choose a method to be executed.

3. Wait for the method to finish.

4. Go to step 1 and repeat.

Essentially, a task tries its various methods repeatedly until its goal is satisfied. This inherent looping behavior gives the RAP system robust situation-driven execution behavior. No matter how a task method interacts with the world, the task will not stop trying to achieve its goal until it "knows" it has succeeded. Other agents may interfere with execution, another task may interrupt in the middle of a method, or the robot hardware may fail to properly carry out an action but the task will not finish until it finds its success clause satisfied. On the other hand, this looping behavior also involves problems that must be overcome to keep the RAP interpreter from getting stuck working on the same task forever. These problems and their solutions are discussed below.




1. Choose a task to run from the task agenda.
2. Check the task against RAP memory to see if it is finished.
3. If the task is not finished, check its methods and choose one that is appropriate in the current situation.
4. If the method is a primitive action:
   - Send it to the hardware for execution and wait for the result.
   - If hardware execution fails, fail this task.
   - If hardware execution succeeds, put this task back on the task agenda.
5. If the method is a network of subtasks:
   - Put each subtask on the task agenda.
   - Put this task on the task agenda to execute again after all of its new subtasks are finished. (Opportunities for early success are not recognized automatically. See Section 5.2)

**Figure 3.3**: The RAP Interpreter Algorithm

### 3.2.1 Issues in Task Execution

There are four major issues that must be addressed in implementing the RAP interpreter algorithm. First, there is the problem of choosing between methods within a task. Second, there is the issue of processing task success and failure. Third, there is the problem of deciding which task to next choose from the task agenda. Finally, there is the potential for getting caught in a futile loop.

To clarify these issues, consider the situation depicted in Figure 3.4. The robot is sitting at location A and knows there is box named X at location D.[1] The robot also has several tasks waiting on the task agenda. One task exists to handle encounters with enemy troops. The robot must be ready to shoot them or run away. Another task exists to deal with low fuel levels. Normally the robot does not worry about fuel consumption and this task ensures that when the fuel level gets low the robot will make a side trip to get more. Finally, there is a task to make the robot periodically look around and check on the local situation. If the robot is engaged in a task, or sitting with nothing to do, it may not be using its sensors and might not notice when items and agents come and go around it. To make sure the RAP memory always represents a reasonable picture of the world nearby, this monitoring task causes the robot to check the situation at regular intervals. These three resident tasks are typical examples of the maintenance goals that any realistic robot is likely to have. The first maintains a local situation free of enemy troops, the second maintains a safe fuel level, and the third maintains an up-to-date awareness of the local situation.

Into this situation we introduce a new goal asking the robot to move the box to location B using the goal `(move-to X B)`. This goal is combined with the RAP in Figure 3.2 to form a new task which is added to the task agenda as shown in Figure 3.5. The agenda now holds four tasks and the system must decide which one to work on. Making this choice is the first issue that must be addressed in implementing the RAP interpreter. We will assume that the enemy-troops task is not eligible to run until enemy troops are actually present and that the low-fuel handling task is not eligible to run because there is plenty of fuel. We will also assume that the situation monitor only runs every ten minutes and has just completed. Therefore, only the move task

[1] In this discussion we will ignore problems raised by the lack of fixed designators like X. Additional sensing operations must be included in the various RAPs to ensure proper identification of a particular box.

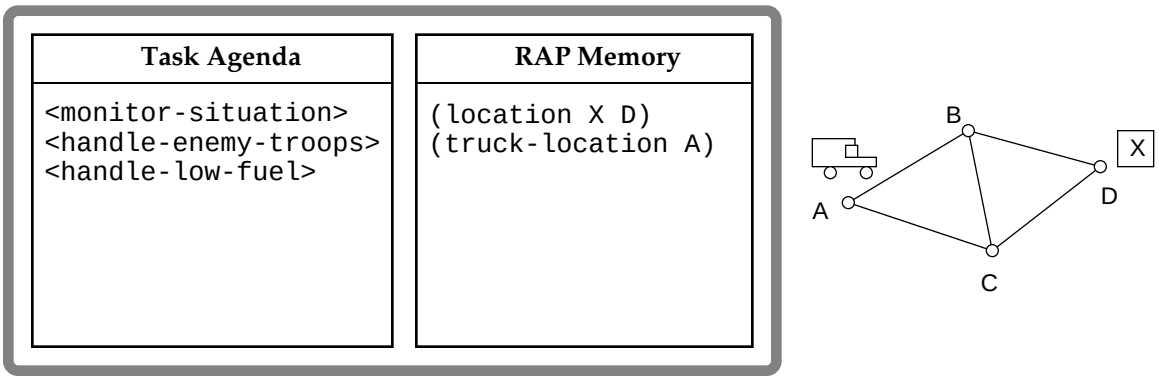



**Figure 3.4**: A Typical Robot Delivery Domain Situation

is currently eligible to run and the interpreter selects it for execution.

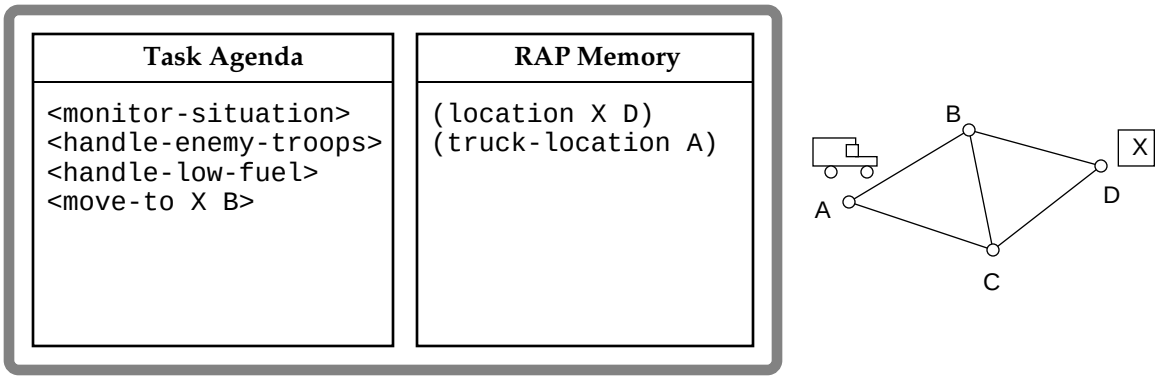



**Figure 3.5**: Adding a New Task to the Agenda

The first step in executing the move task is to check its succeed test against memory. Recall that the succeed clause in the `move-to` RAP (with variables bound) has the form:

```
(SUCCEED (location X A))
```

This clause does not unify with anything in the RAP memory so the task is not finished and a method must be chosen. Choosing among methods is the second issue that must be addressed in implementing the interpreter. In this particular example, it is very simple because the two methods are exclusive and cannot both be appropriate

at the same time. However, this is not always the case and the interpreter must sometimes decide between several candidates.

The method contexts for the `move-to` RAP are:

```
(CONTEXT (and (location X ?loc)
              (not (= ?loc unknown))))
```

and

```
(CONTEXT (and (location X unknown)
              (not (truck-location warehouse))))
```

The first method is chosen because it unifies with the RAP memory, binding `?loc` to D, and giving:

```
(METHOD
   (CONTEXT (and (location X D)
                 (not (= D unknown))))
   (TASK-NET
      (t0 (goto D)
          ((truck-location D) for t1))
      (t1 (pickup X)
          ((truck-holding X) for t2)
          ((truck-holding X) for t3))
      (t2 (goto B)
          ((truck-location D) for t3))
      (t3 (putdown X))))
```

This method is not a primitive action so nothing is sent to the robot hardware. Instead, four new subgoals are generated indexed by: `(go-to D)`, `(pickup X)`, `(goto B)`, and `(putdown X)`. Each of these is associated with RAP code from the library to make a new task and is placed on the task agenda. The annotations in the task-net description place ordering constraints on these new tasks so they must be executed one after another. The annotations also specify preconditions that must be true at the start of the second, third and fourth tasks. Finally, the `move-to` task is placed back on the task agenda, ordered so that it will not be chosen until all of these new subtasks are complete. The result is shown in Figure 3.6.

After one complete cycle through the interpreter algorithm, the task agenda holds eight tasks: the original three maintenance tasks, the `move-to` task, and the four

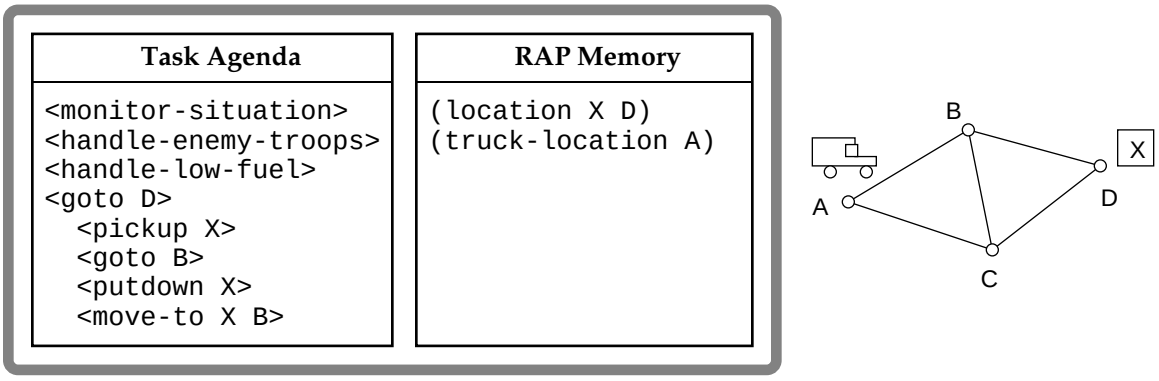



**Figure 3.6**: Expansion of The Move Task

subtasks used to expand the `move-to` task. The interpreter's next step is to again choose a task to run. If no enemy troops have appeared, the fuel level is not low, and it is not yet time to look around, the only task left to select is `<go-to D>`. Note that placing the `move-to` task back on the agenda has had the effect of potentially giving up control to any other task on the agenda. If enemy troops had shown up, the task designed to deal with them would be able to "jump in" and take care of the situation. A given task only has control of the interpreter for that brief period when it is being expanded. Afterwards, control passes either to its children or to some other task on the agenda.

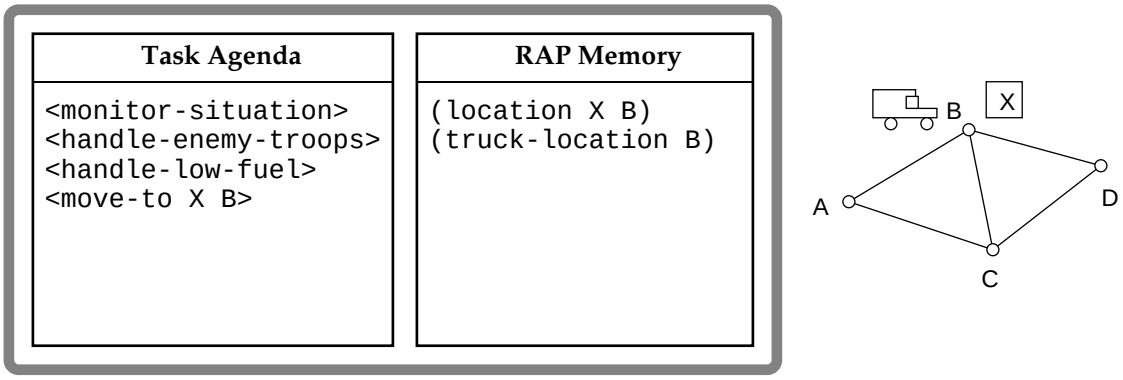



**Figure 3.7**: A Successful Conclusion

Returning to the example, the `move-to` task's children will all execute eventually and several things might happen. First, if they are all successful and nothing else interferes, the execution system and world will be in the state shown in Figure 3.7.

The `move-to` task will again be eligible for execution and, when it is chosen, its succeed clause will unify with the box's location in memory and it will finish. On the other hand, the truck might get to location D and find that the box is not there, as shown in Figure 3.8. The `pickup` subtask will then fail and the `move-to` task should come up for execution again so that its other method can be tried. The third issue in designing the interpreter is making task success and failure behave in this way.

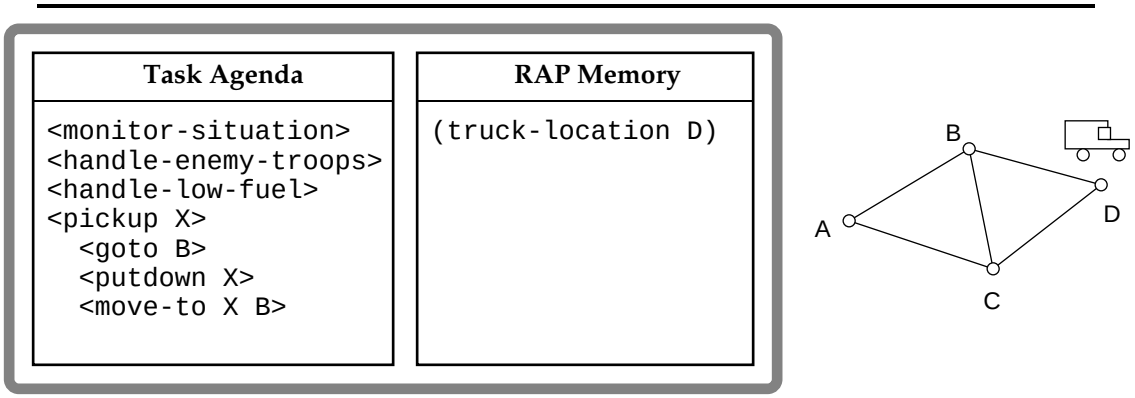



**Figure 3.8**: An Unsuccessful Method Choice

The final issue is detecting and terminating futile loops. Suppose that after reaching location D, the `<pickup X>` task comes up for execution but fails because `X` slips out of the the truck's gripper. The remaining subtasks for `<move-to X B>` will then be removed from the agenda and the `<move-to X B>` task will come up for execution again. It will not yet be satisfied and the same method will be applicable again. The new `<goto D>` task will be satisfied immediately and the new `<pickup X>` task will come up for execution. Suppose that `X` once again slips out of the robot's gripper and, in fact, is just too slippery to ever be picked up. If something isn't done, the interpreter and the robot will be caught in a futile loop trying to pick `X` up forever. Fortunately, situations like this can be detected and the interpreter can stop tasks when they get caught in such loops.

The following discussion goes into each of these four interpreter issues in detail.

### 3.2.2 Choosing a Method

There are several factors that enter into the RAP interpreter's procedure for choosing between methods within a task. By definition, a method cannot be chosen (*i.e.* is not applicable) if its context clause is not satisfied in RAP memory. However, it is often the case that several methods are applicable in a given situation and other criteria must be used to decide between them.

```
(DEFINE-RAP
   (INDEX   (pickup ?thing))
   (SUCCEED (arm-holding ?some-arm ?thing))
   (METHOD
      (CONTEXT (not (too-small-for arm1 ?thing)))
      (TASK-NET
         (t1 (arm-empty  arm1)
             ((not (arm-holding arm1 ?anything)) for t2))
         (t2 (arm-pickup arm1 ?thing))))
   (METHOD
      (CONTEXT (not (too-small-for arm2 ?thing)))
      (TASK-NET
         (t1 (arm-empty  arm2)
             ((not (arm-holding arm2 ?anything)) for t2))
         (t2 (arm-pickup arm2 ?thing)))))
```

**Figure 3.9**: The Pickup RAP

Consider the pickup RAP for the delivery truck domain shown in Figure 3.9. This RAP describes a task to pick an object up in the world and contains two methods: one using `arm1` and the other using `arm2`. In most situations, both arms will be big enough to pick up the requested object and both contexts will be true. Therefore, the interpreter must choose between the two methods some other way. The obvious solution is to choose at random and, when no other information is available, that is what the RAP interpreter does. Assuming neither arm is known to be too small, the first time a pickup task is executed either the method using `arm1` or the method using `arm2` will be selected at random. As an example, assume that the second method is selected and `arm2` is used.

If all goes well, the selected method will work and `arm2` will end up holding the object. Suppose, however, the chosen method completes, the pickup RAP comes up

for execution again, and neither arm is holding the object. This means that for some reason or another the chosen method did not result in the object being picked up. If the failure occurred because `arm2` is actually too small and the RAP memory was incorrect, or incomplete, the memory will have been updated by the hardware interface to include an assertion that `arm2` is too small. In this situation, the pickup RAP will not have a satisfied succeed clause, method choice will occur again, the second method will be ruled out because its context will not be satisfied, and the first method must be tried.

However, another possibility is that the second method will fail even though `arm2` is not too small. This might happen for several reasons, such as another task trying to use `arm2` at the same time, or `arm2` being unable to put down something it is already holding. Either way, it makes sense to try and use `arm1` before attempting to use `arm2` again. To achieve this behavior, the task's past execution history is used to prefer unfailed methods over those that have not succeeded when several are applicable.

Thus, the algorithm used by the RAP interpreter to select a task method is:

- Check each method's context to see which are applicable.
- Check the task's execution history to see whether any methods have already been tried and failed.
- Consider only applicable methods that have failed the fewest number of times.
- Choose randomly between these methods and record which was chosen.
- Execute the chosen method and record its success or failure as an annotation to the task.

### 3.2.3 Handling Success and Failure

Once a task has been introduced into the system it remains under active consideration on the agenda until it either succeeds or fails. When a task succeeds, the task's goal is sure to have been achieved, or at least the RAP memory will believe it to have been achieved. Conversely, when a task fails it means all applicable methods known for executing the task have been tried and nothing worked. However, task failure does

not necessarily signal failure of the goal the task was in service of. In particular, when a subtask fails, it means that the method it was part of is not working, but it does not mean that another method should not be tried. Task failure really means that another way for accomplishing the task should be tried, not that the task is impossible. Only when all methods have been exhausted without success is the RAP system at a loss and forced to abandon the task.

A task can succeed only when its succeed test is satisfied in the RAP memory. Since satisfaction of a task's succeed test means that the task's goal is accomplished, all that the interpreter must do on task success is to remove the task from the execution agenda. When a task is removed from the agenda any tasks waiting for it to finish become eligible for execution.

A task can fail for several reasons, but the most common problem is for a task to have no methods applicable in the existing situation (*i.e.*, no method context is satisfied in memory). In some situations, it might be appropriate to wait and try the task again later, but the interpreter does not do that automatically. To get waiting behavior, the RAP describing the task must explicitly include waiting as one of its methods (see Section 4.7).

A task can also fail for two other reasons: it might get caught in a futile loop, trying the same method again and again (see Section 3.2.5), or it might come up for execution without its preconditions being met. Preconditions are specified either explicitly in the RAP code for the task (see Section 4.4) or as annotations an a task when it appears in a task-net. Recall one method from the pickup RAP discussed previously:

```
(METHOD
   (CONTEXT (not (too-small-for arm2 ?thing)))
   (TASK-NET
      (t1 (arm-empty  arm2)
          ((not (arm-holding arm2 ?anything)) FOR t2))
      (t2 (arm-pickup arm2 ?thing)))))
```

The `FOR` annotation in the task-net means that task `t1` is designed to generate a state in which `arm2` is not holding anything when task `t2` executes. When such an annotation is encountered in a task-net, the RAP interpreter attaches an explicit check for the indicated state as a precondition on the subtask it is being generated

for. In the example, task `t2` gets the precondition that `arm2` not be holding anything when it starts to execute. When task `t2` is chosen for execution, the interpreter first checks its succeed test and if satisfied the task finishes. If the task is not finished, the interpreter checks the attached precondition that `arm2` be empty. If `arm2` is empty then a method is chosen, but if `arm2` is not empty, it means the method is not working as planned, and task `t2` should fail.

No matter why a task fails, the result is always the same. The task is removed from the task agenda and any tasks waiting for it to complete become eligible to execute. Furthermore, if the task was generated as part of a method all other tasks in the method are also removed from the agenda. A task's siblings are removed when it fails because the failure suggests very strongly that the method is not proceeding as planned. Hence, the whole method is abandoned to enable the parent task to try another. Therefore, to summarize, the algorithm used by the RAP interpreter to handle task success and failure is:

- If a task succeeds, remove it from the task agenda.
- If a task fails, remove it from the task agenda along with all other tasks that are part of the same method.
- Enable all tasks on the agenda that were waiting for the removed tasks to finish.

When a task succeeds this algorithm results in the next step in the method to which it belongs starting up, or, if it is the last step in a method, its parent starting up. On the other hand, when a task fails, the method it is part of is aborted and its parent is started up to try and make a better method choice. Primitive actions are treated exactly the same way as other tasks except that they are always the only step in a method.

### 3.2.4 Managing Tasks on the Agenda

The execution of a single task must be concerned with selection of methods and response to method success and failure. When several tasks are active concurrently, there is the additional problem of choosing which task to work on at any given time. The robot delivery example discussed in Section 3.2.1 began with a task agenda

containing four tasks: the delivery task itself, a task to deal with enemy troops, a task to deal with low fuel, and a task to periodically sense the local situation. These tasks were completely independent and corresponded to separate top-level goals assigned to the system in no particular order. The task-net syntax used for RAP method descriptions also allows subtasks to remain unordered with respect to one another. Thus, when the RAP interpreter comes to choose a task from the agenda for execution, there will usually be several candidates available to choose from.

There are two major assumptions made by the RAP system with respect to task selection. The first is that only one task can be executed at a time. More specifically, there is only one interpreter and no task is chosen from the agenda before the task chosen previously has completed, or been returned to the agenda to wait. This assumption follows from letting only one primitive action execute at a time. Since primitive actions cannot be overlapped, there is no reason to try and execute tasks in parallel. However, tasks can *appear* to execute in parallel in exactly the same way that multi-processing operating systems share one computer between several processes. After a task chooses a method and installs its subtasks on the agenda, it returns there to wait for the method to complete. If another task does the same, the two sets of subtasks may interleave, with the interpreter choosing first a subtask from one method and then a subtask from the other. Thus, on a micro-level only one task can execute at a time, but on a macro-level unordered tasks appear to execute concurrently.

The second assumption in task agenda management is that looking into the future can be neglected. Sometimes the precise order in which tasks are chosen from the agenda can have a significant effect on the overall efficiency of their pursuit. The simplest example is when two tasks require being in different locations and each expands into many subtasks. If the subtasks are allowed to interleave, the robot will end up moving back and forth between the two locations and waste a lot of time. A better strategy might be to complete one task before changing locations to work on the other.[2] Whenever unordered tasks are competing for a resource, such as location, careful scheduling to avoid unnecessary conflicts can make execution more efficient. However, such scheduling requires looking into the future, considering

[2]Unless each task involves a process that contains plenty of waiting (baking bread for example) where it might be better to shuttle back and forth while nothing else is happening.

different possible task orderings and their resource use, and then deciding on the most efficient. This is exactly the projection problem that situation-driven execution, as embodied in the RAP system, is designed to do without. Therefore, tasks are selected from the agenda without any attempt to consider the consequences of alternative orderings.

Appeal has often been made to the notion of a task on the agenda being "eligible" to run. When tasks are placed on the agenda they are usually ordered with respect to other tasks and must wait for them to complete. A task waiting for another to complete is said to be ineligible to run and cannot be selected by the interpreter for execution. Tasks can also be made ineligible for execution by making them wait for states of the world to become true or for specific intervals of time to pass. Tasks that are ineligible to run have not failed, they are explicitly waiting for some state to become true in the world before they continue. The RAP interpreter only chooses between eligible tasks.

### Task Selection Constraints

There are three types of constraint that can be placed on a task to control its eligibility for execution:

- Explicit ordering constraints
- Temporal constraints
- Memory content constraints

Explicit ordering constraints arise from method task-nets and from the interpreter itself, which adds constraints to make tasks wait for their methods to complete. Temporal constraints are added by method task-nets (see Section 4.5) and attach a start-time or deadline to a task. A task does not become eligible to run until after its start-time has passed and, once eligible, the interpreter tries to complete the task before its deadline passes. Memory content constraints are added as part of the RAP code for a task (see Section 4.7) and consist of states that must be true in the RAP memory before the task can be executed. The example tasks for handling

enemy troops and handling low fuel use memory content constraints to control their eligibility for execution.

When selecting a task for execution from the agenda, the interpreter cannot choose a task that has constraints which make it ineligible to run. Specifically, a task is ineligible for execution when any of the following situations hold:

- Any task it has been constrained to follow has not yet completed.
- All of its assigned start times have not yet passed.
- A state it has been constrained to wait for is not currently satisfied in the RAP memory.

The use of constraints allow task behavior descriptions to include complex temporal relationships. Ordering constraints permit the generation of subtasks that must be completed one after another and temporal constraints permit the addition of time delays and deadlines. Sometimes a task requires that one step follow another but not before a small delay has passed. For example, buying a soda from a vending machine might include the steps `insert-money` followed by `press-soda-button`. However, the button should not be pressed until a short interval has passed to allow the money to fall down to wherever it goes. To guarantee such a pause, a temporal constraint might be used which states that the second step cannot start execution until at least one second after the first step has completed. Memory content constraints allow task execution to be synchronized with external states of the world. Refueling the truck when the level gets low, or waiting for an alarm or timer to sound can be represented as tasks that wait on the agenda until the appropriate fact becomes true in memory. The refuel task can wait for an assertion that fuel is low, and the task waiting for a timer can wakeup when an assertion is made that a ringing noise has occurred.

### Task Selection Heuristics

Once a task is eligible to run, the RAP interpreter must still use heuristics to chose between it and other tasks that are also eligible to run. The heuristics used, in order of application, are:

- Prefer higher priority tasks over lower priority tasks.
- Prefer tasks with deadlines that are closer.
- Prefer tasks that have never failed.
- Prefer tasks from the same family as the last task chosen.
- If there are still several tasks in contention, choose one at random.

The first heuristic allows important tasks to interrupt less important ones. The second heuristic attempts to ensure that deadlines are met as best they can. The third switches the system's focus of attention to give external interference a chance to go away before retrying a task, and the fourth focuses the system's attention on a single task to prevent gratuitous interference between concurrent tasks. These heuristics are similar in nature to those used for conflict resolution in production system languages like OPS5 [Brownston *et al.*, 1985, McDermott and Forgy, 1978]. Deciding which production to fire and which task to execute involve many of the same problems. In particular, both systems must make compromises between being *sensitive* to their environments and being *stable* in their behavior [Bowen and Kang, 1988]. The first three selection heuristics make the RAP system sensitive to changes in its situation, and the fourth keeps the system stable by focusing its attention on one task at a time.

The first factor that the RAP interpreter looks at when selecting a task to run is task priority. The highest priority task that is eligible to run is always chosen — only when there are several eligible tasks with the same priority are other heuristics considered. Task priorities are single integer values with higher numbers defining tasks of higher priority and they are assigned to tasks through the method task-net syntax (see Section 4.5). The usual use for task priorities is to enable high importance tasks to interrupt less important tasks. For example, the simple robot delivery task discussed previously starts with four tasks on the agenda. When appropriate priorities for the four tasks are displayed, the example becomes that shown in Figure 3.10. The three maintenance tasks have higher priorities than the move task so that they will be dealt with first. However, the low fuel and enemy troops tasks are constrained to run only when the fuel is low or when enemy troops are present and the situation monitoring task is usually waiting for a period of time to pass. Only after one of

these tasks become eligible to run will it be chosen because of its higher priority. In fact, the three maintenance tasks have different priorities so that they can interrupt each other as well. Keeping the local situation description up-to-date is the most important, followed by dealing with enemy troops, and finally getting more fuel. While these tasks conceptually interrupt each other, there is no actual interruption in the process/computer-program sense. Rather, when the interpreter is finished with a task and goes to the agenda to choose another, high priorities override any other heuristics that might apply to the choice.

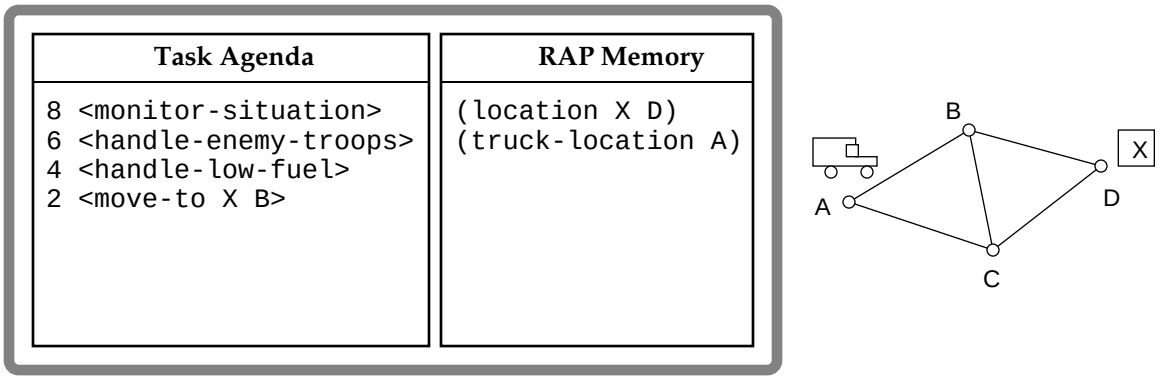



**Figure 3.10**: An Execution Situation with Priorities

If more than one eligible task has the same priority, the interpreter considers any deadlines that have been assigned to each task. Deadlines are assigned through task-net annotations and represent times by which tasks should be completed. The interpreter attempts to honor deadlines by choosing tasks with short intervals between estimated completion time and deadlines over those with longer intervals. Thus, if two tasks are eligible to run with estimated durations of 10 hours, but one is to be finished today and one tomorrow, the task to be done today will be selected first. Subtask deadlines are inherited from their parents and tasks have no deadline by default. An important note is that the use of this heuristic requires the RAP interpreter to have some notion of how long a task will take. This information can be supplied in the form of an annotation in the RAP code for the task (see Section 4.3) but is only an approximation at best. The expected duration of a task cannot be calculated from its subtasks because that would require looking into the future and considering alternate expansions. If a task is not explicitly annotated with an expected duration,

its duration is assumed to be zero. When tasks have estimated durations of zero, those with early deadlines are preferred over those with later deadlines.

After priorities and deadlines, the RAP interpreter prefers unfailed tasks over those that have been tried but failed. This heuristic is based on the assumption that the usual reason for a method failure is that an external influence is interfering with execution. For example, another agent might be trying to manipulate the same object, or an object might be used in a process that has not finished yet. In such cases it is a reasonable idea to work on something else for a while and then come back and try the task again later. The RAP interpreter embodies this idea by preferentially choosing tasks that have not yet failed.

If there are still several tasks to choose from after considering the previous three heuristics, the RAP interpreter selects a task from the same family as the last task chosen. All tasks and subtasks generated in executing a single top-level task belong to the same family. For example, all of the tasks spawned in service of a move task belong to the move task's family. Preferring tasks from the same family has the effect of focusing the system's attention on the execution of one top-level task at a time. This results in execution proceeding from one method subtask to the next without interference from other tasks that might also be eligible for execution. Focusing the system's attention minimizes the number of subtask preconditions that are inadvertently "clobbered" by other tasks within the system. However, focusing the system's attention on a single task also has the effect of putting off other task families, hence the need for priorities and deadlines to shift the system's attention when the situation demands.

To achieve reasonable behavior when interrupted, the system actually maintains a stack of past families, rather than just a single one. Whenever the next task chosen for execution is of higher priority than the last task chosen, the new task's priority and family are pushed on the stack. The top family on the stack then becomes the new focus of attention. If a higher priority task gets chosen in the future, it is pushed on the stack. If a different family of the same priority gets chosen to execute, the old family is popped from the stack and the new family is pushed. When no more tasks at the current (or higher) stack priority are eligible to run, the top priority/family pair is popped from the stack and focus of attention shifts to whatever family was executing previously. The use of a stack means that when a low priority task like a

move task is interrupted by a high priority task like getting fuel, the system returns to the same low priority task — the move task is taken up again and not some other task of the same priority.

Finally, when all heuristics have been considered and there are still more than one task to choose from, the RAP interpreter selects among the candidates at random.

### 3.2.5 Preventing Futile Loops

Another problem inherent in the looping nature of the RAP interpreter's algorithm is the possibility of a task getting caught in an endless futile loop. Suppose a task comes up for execution, consults the RAP memory and chooses a method to try. The method completes and the task comes up for execution again and when RAP memory is again consulted, it has not changed. The task might then choose the same method as before and wait for it to complete. If the RAP memory still has not changed, the cycle might continue indefinitely.

The key to detecting such loops is to realize they arise because the same method is being chosen repeatedly in exactly the same circumstances. In particular, the same method's context is satisfied in the RAP memory with the same variable bindings. If the task's variables are bound the same way each time a method is chosen, execution is having no discernable effect of the world. If such a situation continues, the task will be stuck in a loop and will never make progress towards satisfying its succeed clause. Therefore, when a task comes up for execution and a method is chosen that has been chosen previously, the RAP interpreter checks to see if the same variable bindings are in effect. After a method has been chosen with the same bindings a certain number of times, the interpreter causes the task to fail. Making the task fail stops the looping behavior and gives the task's parent a chance to choose a method more applicable to the current situation. Even if the parent is also looping and chooses to generate the same task again, execution will eventually terminate when its looping behavior is also detected. The examples run in this thesis use a looping threshold of two: once a task tries the same method twice it fails.

For example, consider the situation shown in Figure 3.11. The truck is attempting to pick the box named X up off the ground. However, each time it grasps the box, the other agent grabs the box and puts it back on the ground. Regardless of how

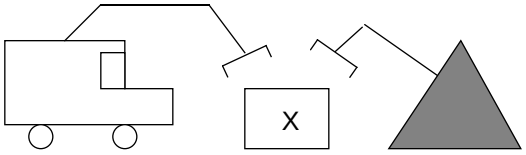



**Figure 3.11**: A Possible Looping Scenario

the pickup task is encoded, when it comes up for execution the box will be on the ground. The pickup task cannot avoid getting caught in a loop for as long as the other agent continues to be difficult. The RAP interpreter will detect the loop after two cycles, however, because the situation at method choice time is always the same and all variable bindings will be identical. The task will then fail and its parent will have a chance to choose a method that takes the other agent into account.

One result of detecting futile loops via the capture of variable bindings is that explicit looping behavior must be coded carefully. To catch all loops, the RAP interpreter must record *all* variable bindings active within a task at the time a method is instantiated. This includes not just the variables bound by the method's context, but also those bound at task instantiation time as well as those bound by the various RAP clauses, such as REPEAT-WHILE, described in the full syntax description in the next chapter. The result is that an explicit loop that invokes the same method repeatedly without changing the situation in a measurable way (like banging on an iron water pipe until it breaks)[3] will be terminated by the futile loop detector. To avoid this termination, loops *must* generate a different binding each iteration to explicitly inform the detector that progress is being made. Thus, arbitrary loops are prohibited by the loop detection algorithm.

Arbitrary task recursion must also be prohibited. If a task is allowed to invoke itself (or one of its parents) as an identical subtask, the recursion may go on forever but bypass the loop detector. Therefore, at task generation time, the RAP interpreter checks each task's parents and if any were invoked with the same bindings, the spawning task is made to fail. This restriction, along with the loop detection algorithm,

[3] I saw this on the TV show *This Old House* once. These guys were banging on an iron pipe fitting with a huge hammer for a long time with no apparent progress and then, quite suddenly, it shattered. I don't know how they knew they were making progress, or when they would have given up if the fitting just wouldn't break. I imagine they were going on the assumption that such fittings always do break and thus one should simply never give up.

prevents task execution from getting caught in infinite loops because of unforeseen interference from the outside world. Non-terminating RAPs can, however, be coded on purpose if care is not taken. Loop detection is not a cure-all for incorrect, or "buggy" RAPs — its purpose is to detect interference that cannot be anticipated in advance.

## 3.3 Primitive Actions and the RAP Memory

An important aspect of situation-driven execution is making sure that memory reflects an accurate picture of the actual world state. The RAP system would ideally base task method choice directly on the real world. However, because of sensor limitations, the world state cannot always be perceived directly and memory must be used to preserve sensor data for later use. The RAP memory represents the world state as an assertional database and attempts to automatically keep the database up-to-date as new assertions are made, even when the sensor names of the items involved change from encounter to encounter. The RAP interpreter uses the database as a basis for method choice and assumes that, as primitive actions generate sensor data, the database will be updated accordingly. The question that remains is how assertions are generated to update the memory when primitive actions are executed.

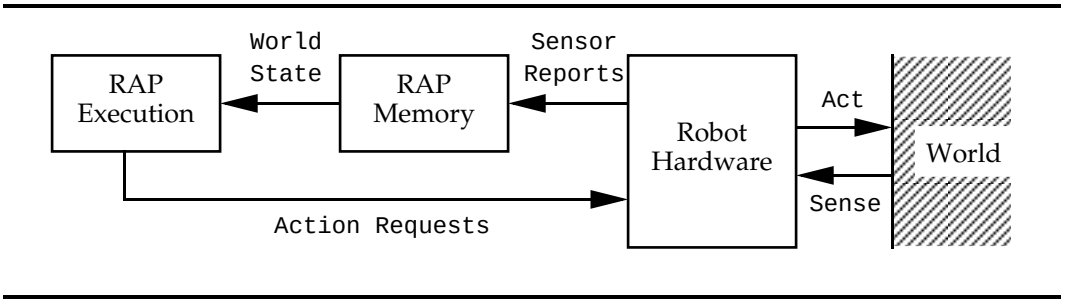



**Figure 3.12**: The RAP Memory Model

Recall the description of the RAP memory as a buffer for sensor data illustrated in Figure 3.12. The RAP interpreter generates primitive action requests and passes them to the robot hardware to execute. After actual execution takes place, the results update the memory's model of the world. For example, suppose the robot truck is trying to pick a box up from the ground beside the truck. It would generate a primitive action to move its gripper to be at the box and success results in the memory being

updated to reflect the fact that the gripper is at the box. The robot would then try to grasp the box and success would update the memory to show that the gripper is holding the box. Furthermore, even when a primitive action fails, the result will update the world model. If the robot moves its gripper to the box with success, but the grasp action fails because the box isn't there, the memory should be updated to eliminate the assertion that the gripper is at the box. Some other agent must have taken the box between the move and grasp commands.

This section discusses the assumptions and algorithms used to update the RAP memory as the system is used to run the robot deliver truck simulator. The specific examples used in the discussion are domain dependent, as is to be expected, but the idea that memory be updated only as a direct result of action in the world is a domain independent concept.

### 3.3.1 Primitive Actions and Sensor Data

From the RAP execution system's point of view, each primitive action is an atomic operation that will return either success or failure when finished. If the action succeeds it simply returns success and if it fails it returns a single symbol that specifies the type of failure as best the hardware can determine. The RAP system executes only one primitive action at a time and waits while execution takes place according to the algorithm:

1. Issue the primitive action request to the hardware
2. Wait for the action to complete and memory to be updated
3. Report success or failure to the issuing RAP

No other primitive actions or further RAP execution takes place until the action completes. This keeps the RAP memory in a consistent state at all times because the results of a primitive action will never be trying to change memory at the same time as those of another action or at the same time memory is being referenced by another task.

The simulated robot truck hardware takes a similar view of primitive action execution. Primitive action requests are executed one at a time and each completes with

a report on its success or failure. However, each primitive action executed by the simulator may also generate an arbitrary amount of sensor data. This sensor data is a list of notifications about individual aspects of the world. Typical notifications contain information about an item being seen, that there is a road in some direction, or that a specific item property like color or size has been measured. Notifications of item properties refer to items by their locally assigned sensor names. Primitive actions and sensor notifications for the robot truck simulator are given in Appendix B and discussed in some detail in Firby and Hanks [1987b].

The hardware interface between the RAP system and the robot delivery truck simulator handles the execution of action requests and the interpretation of their results. The interface takes each action request generated by the RAP interpreter and passes it on to the simulated hardware for actual execution. When execution is complete, the hardware returns either success or failure to the interface along with any sensor data generated during execution. The interface adjusts the RAP memory according to the action's success or failure and then returns the success or failure to the RAP interpreter. The interface also interprets the information contained in each sensor datum and updates the RAP memory accordingly. The overall flow of information through the hardware interface is shown in Figure 3.13.

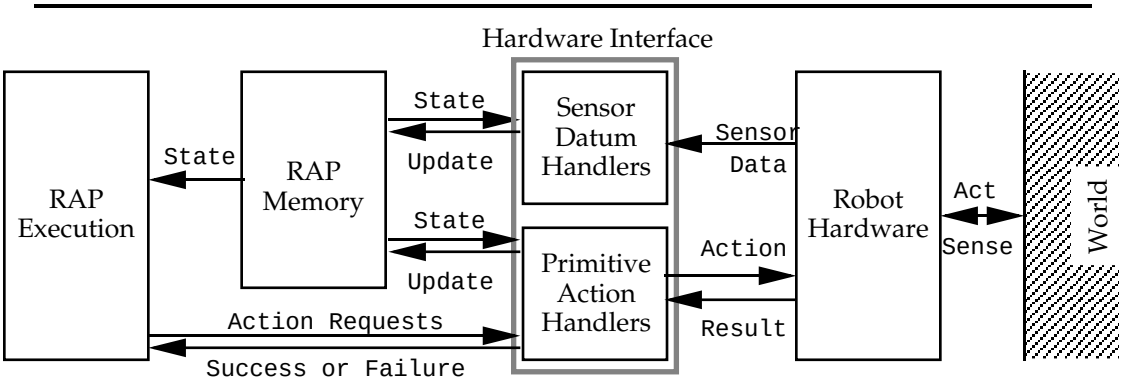



**Figure 3.13**: The RAP Hardware Interface

As an example, consider the primitive action `(arm-grasp arm1 item-35)`. This action requests the hardware to close the gripper on `arm1` around `item-35`. For such a grasp action to succeed, the arm involved must already by positioned at the object to be grasped and must not to be holding too many other things. Execution of the

action may also fail because there is always a chance that during a grasp the gripper will drop the object. The simulated hardware can quite plausibly detect each of these situations and when it does returns one of the failures: `arm-not-at`, `arm-too-full` or `arm-dropped` [Arbib, 1981, Arkin, 1988]. If the `arm-not-at` failure is generated, it means either the arm was not positioned properly, or some other agent snatched the object away. In either case, the hardware interface must erase all assertions from the RAP memory to the effect that `arm1` is positioned at `item-35`. If the `arm-too-full` failure is generated, the interface updates the RAP memory to reflect that `arm1` is holding too much to grasp `item-35` as well. In fact, if `arm1` is not holding anything already, an assertion is added to memory stating that `item-35` is too big to pick up at all. When an `arm-dropped` failure occurs, the hardware interface does not change the memory because nothing has changed in the world. When the action succeeds, the interface must update the RAP memory to state that `item-35` is being held by `arm1` and is no longer sitting on the ground.

A different example is illustrated by the primitive action `(eye-examine external item-35)` which asks the hardware to direct it "eye" at `item-35` outside the truck and look at it closely. This action will fail only if `item-35` is not actually located within sensor range. If the failure occurs the memory is updated to show that `item-35` is not outside the truck, and if the action succeeds, an assertion is made to memory that `item-35` was examined. Besides these simple memory updates, however, a stream of sensor data is usually generated if the action succeeds. The precise data generated will depend on the object being examined and its state. If `item-35` is a rock, separate notifications of its size and color will be generated. If it is an open box, the size of the box and each of its contents would generate a separate notification.

### 3.3.2 Updating the RAP Memory

The procedure used by the hardware interface to update the RAP memory makes two assumptions. First, it assumes that all memory modifications necessitated by primitive action and sensor data processing are independent of the state of the RAP execution system. Therefore, the hardware interface never has to look at the tasks that are executing. This assumption stems from the fact that the memory is supposed to reflect the current state of the outside world and as such, changes to the memory

should only depend on actions actually taken in the world and not those anticipated. Second, the interface assumes that each primitive action and sensor notification can be dealt with in isolation. Although sensor data may be generated in bursts as a result of specific action requests, each datum is processed as if it were generated by itself. This assumption reflects the idea that the same sensor notification may be generated in many different ways, by many different sensors and the semantics of the notification should not depend on where it came from.

These are essentially the same assumptions that justify the use of add/delete lists in other systems [Fikes and Nilsson, 1971, Wilkins, 1988] with one big difference: add/delete lists are used to predict the future, while RAP primitive handlers respond to actual changes in the world. Many planning systems model the effects of a primitive action as a set of assertions to be added or deleted from the system's world model. The assertions in the set depend only on the action involved and each action can be treated in isolation from all others. Most systems also assume that the assertions to be changed are independent of the world state and hence there is no need to query the memory before processing the add/delete list. Recent work has begun to relax this assumption somewhat and include context dependent action effects [Schoppers, 1987b, Wilkins, 1988, Dean *et al.*, 1988]. In the RAP system, the effects of primitive actions are inherently context dependent so the notion of add/delete list is generalized to that of arbitrary *handlers* designed to process each type of primitive action and sensor notification.

Most handlers in the delivery truck domain consist of instructions to add and delete particular assertions from the RAP memory database, but arbitrary LISP code is allowed so that changes to the memory can be based on arbitrary analysis of the current memory state. For example, when an object grasp is attempted but fails because the arm is already holding too much, it should be possible to infer a lower limit on the size of the object — the size of all of the items currently held by the arm might be summed and subtracted from the known capacity of the arm. An assertion could then be made to remember that the object being grasped is at least that large. The current interface to the simulator does not do the summation, but it does make different assertions depending on the what the arm is already holding. In particular, when the object cannot be grasped and the arm is empty, an assertion is made stating that the object is too big to ever grasp, but when the arm is holding other things,

an assertion is made to the effect that the arm is too full to pick up the object just now. All such assertions are retracted by the ungrasp handler whenever the arm puts something down.

The use of separate LISP procedures to process sensor notifications and primitive actions rather than more declarative representations like add/delete lists can be justified for several reasons. First, each robot will have a fixed number of primitive actions and sensor notifications available to it and these will not often change. While planning knowledge and hence the RAP library may change and grow as the robot is made more capable in the world, the interface handlers will usually remain fixed for a particular robot since they are defined by the hardware and the domain. Second, the memory changes required of a handler may depend on the RAP memory state in complex ways.[4] Therefore, whatever coding scheme is used, handlers will have to be quite complex. Rather than invent a new, complex representation, LISP code is used. Finally, the RAP execution system does not try to use the handlers for any purpose other than updating the memory in response to activities in the real world. The processing of actions and sensor data is encapsulated in the interface and its mechanics are of no interest to the RAP interpreter. The interpreter and RAPs only use the results of that processing as it appears in the RAP memory. Thus, handlers are effectively part of the robot (or simulator) hardware and may fruitfully be considered the primitive actions themselves.

The hardware interface is broken down into a control structure and two libraries of handlers: one for the primitive actions and another for the sensor handlers. The control structure is:

1. Accept an action request from the RAP interpreter.

2. Pass the action on to the hardware and wait for the resulting success or failure and sensor data.

3. Call the appropriate primitive action handler to alter memory in response to the success or failure returned.

[4] By complex I mean that handlers must include all the calls needed to find out the current state of memory, decide how it should be changed, and make the changes. The changes will include erasing and adding assertions and shuffling items around in expectation sets. See Figure 3.14 and Appendix A.

4. Loop through each sensor datum and call the appropriate sensor datum handler on each one.

5. Return success or the failure to the RAP interpreter.

### Two Examples

As an example consider the primitive action handler for `arm-grasp` shown in Figure 3.14. This code implements the memory update process required for an action request of the form `(arm-grasp arm object)`. The handler is a function of two arguments: the action request and the result of executing the action in the world. When an `arm-grasp` action is requested by the RAP interpreter, the interface passes it to the hardware and waits for the result. The handler function is then called with the request and result as arguments. If the result is success (or `okay`) the function changes the grasped item's location in memory to be held by the arm. It also erases the fact that the arm was positioned to grasp the item and any temporary assertions to the effect that the item's old container was too full to put other things inside. If the result is `arm-too-full`, the function makes an assertion that the arm is either temporarily too full, or that the arm is simply too small to pick the item up. If the arm is not positioned correctly, the function makes sure the memory does not say that it is, and any other failures (like `arm-dropped`) cause no change in memory at all. The result returned from this function is returned to the RAP interpreter. Note that the changes made to the RAP memory are context dependent for results `okay` and `arm-too-full`.

The handler for `eye-examine` is shown in Figure 3.15. It is much simpler than the `arm-grasp` handler because it does not change the world. If the action succeeds a notation is made so that gratuitous sensor operations can be avoided, and if the item being examined cannot be found, its location is changed to `unknown`. Almost all memory changes generated by an `eye-examine` primitive action result from processing the sensor notifications it generates.

Two examples of sensor handlers are shown in Figure 3.16. The first of these deals with notifications of item colors. Such notifications are usually generated by `eye-examine` actions and the appropriate memory update is simply to assert the color as a property of the item. The second example is more complex and is used to

```
(declare-primitive-action-handler 'arm-grasp
   (lambda ( request result ) ; request = (arm-grasp arm thing)
      (let ( (arm   (cadr request))
             (thing (caddr request)) )
         (cond ((eq result 'okay)
                (memory-erase *memory* `(arm-at ,arm))
                (let* ( (env  (create-binding-environment))
                        (okay (memory-query *memory* env
                                            `(location ,thing ?a)))
                        (place (binding-value env '?a)) )
                   (memory-erase *memory* `(too-full-for ,place))
                   (memory-erase *memory* `(holding ,place ,thing)))
                (memory-erase  *memory* `(location ,thing))
                (memory-assert *memory* `(arm-holding ,arm ,thing))
                (new-item-location *memory* thing arm)
                'okay)
               ((eq result 'arm-too-full)
                (let* ( (env  (create-binding-environment))
                        (any-extra
                         (memory-query *memory* env
                                       `(holding ,arm '?any true))) )
                (if any-extra
                 (memory-assert *memory* `(too-full-for ,arm ,thing))
                 (memory-assert *memory* `(too-small-for ,arm ,thing))))
                'arm-too-full)
               ((eq result 'arm-not-at)
                (memory-erase *memory* `(arm-at ,arm ,thing))
                'arm-not-at)
               (else
                result)))))
```

**Figure 3.14**: The `arm-grasp` Primitive Action Handler

update memory in response to `object-seen` notifications which usually result from `eye-scan` actions (see Appendix B). An `object-seen` notification means that an item of a particular class has been discovered at an accessible location and has been assigned a sensor name. The class and location of the item are asserted in the memory database, and the item is introduced into the memory's local model with the assigned sensor name. Introducing the item places it in the appropriate memory expectation set as discussed in Chapter 2.

```
(declare-primitive-action-handler 'eye-examine
   (lambda ( request result ) ; request = (eye-examine place thing)
      (let ( (thing (caddr request)) )
         (cond ((eq result 'okay)
                (memory-assert *memory* `(eye-examined ,thing))
                'okay)
               ((eq result 'eye-cant-find)
                (memory-erase *memory* `(location ,thing))
                (new-item-location thing 'unknown)
                'eye-cant-find)
               (else
                result)))))
```

**Figure 3.15**: The `eye-examine` Primitive Action Handler

```
(declare-sensor-datum-handler 'color
   (lambda (datum) ; datum = (color thing color)
      (let ( (thing  (cadr datum))
             (color (caddr datum)) )
         (memory-assert *memory* `(color ,thing ,color)))))

(declare-sensor-datum-handler 'object-seen
   (lambda (datum) ; datum = (object-seen place thing class)
      (let ( (place (cadr datum))
             (thing (caddr datum))
             (class (cadddr datum)) )
         (memory-assert *memory* `(class ,thing ,class true))
         (cond ((member place '(arm1 arm2))
                (memory-assert *memory* `(arm-holding ,place ,thing)))
               (else
                (memory-assert *memory* `(location ,thing ,place))))
         (introduce-new-item *memory* thing place class))))
```

**Figure 3.16**: Two Sensor Datum Handlers

### 3.3.3 The Hardware Interface and The RAP Interpreter

The primary reason for separating memory update from the RAP representation and algorithm is so that updates only occur when direct knowledge of the world make them appropriate. Since the memory only changes as the result of primitive actions, it will always reflect the best available picture of the real world state. RAPs are not allowed to alter the memory arbitrarily because they may make incorrect inferences or, should they be interrupted by some other process, may make inferences that are out of sync with the best known world knowledge. Dependency machinery could be set up to track the course of RAP based inference and memory changes so that assertions that lost support would be removed automatically, but it would be computationally very expensive and experience has shown it to be unnecessary for the robot activities attempted so far. No inference is allowed in the current model and all RAP interpreter queries to memory result only in unification of unbound variables.

An interesting experiment would be to add a new primitive actions to the system called `assert` which would make an assertion directly to the RAP memory. If this were done, tasks could be used to do complex inferences and update memory as a result. Inference would occur through task execution and would be interrupted and scheduled just like any other task. The effect would be to allow complex analysis of the data acquired via primitive actions to be done in the robot's "head". However, no dependencies would be set up and the tasks would have to be structured carefully to keep inferences from getting off track during interruptions or the execution of other, independent tasks. The interesting thing is that the RAP system itself does not have to change at all to support such an experiment.

## 3.4 Summary of The RAP Interpreter

There are three main concepts to take away from this discussion of the RAP interpreter: the way that a single task is structured and executed, the way that the execution of concurrent tasks is mediated, and the way that primitive actions change the world and update memory.

### 3.4.1 Executing a Single Task

Every goal that the execution system is assigned is combined with a RAP description from the library to form a task with the following components:

- The RAP code defining a success test and methods.
- A priority and possibly a deadline.
- Scheduling constraints.
- Preconditions added by task nets.
- Current variable bindings.
- A record of past method choices, their success or failure, and the variable bindings that went with them.

Each component is used by the RAP interpreter for a different purpose. The task priority, deadline and scheduling constraints are used to coordinate the task with other tasks that might also be on the task agenda. The preconditions are used to ensure that the method this task is a part of is proceeding as expected and the variable bindings are by the task to preserve internal state. Finally, the past execution record is used both to aid in method selection and in detecting and terminating futile loops.

Following a single task as it proceeds through the system shows it executing the algorithm:

- Check success clause: if satisfied succeed.
- Check preconditions: if not satisfied fail.
- Check method contexts to see which are applicable: if none are applicable fail.
- Choose an applicable method and record variable bindings.
- If the method chosen is part if a futile loop, fail.
- Install method task-net in agenda or execute primitive action request.
- Return to task agenda to await method completion.

- Record method success or failure.

The only way a task can finish successfully is to be selected from the agenda at a time when its succeed test is satisfied in RAP memory. In other situations the task loops, repeatedly trying methods in an attempt the satisfy its success condition. Other than succeeding, a task might also fail if preconditions imposed upon it are not true, if it has no applicable method, or if it is caught in a futile loop. A failed precondition means the method using the task has been interfered with and should be aborted. No applicable method means that the system doesn't know what to do, and a futile loop means that some unanticipated interaction with the world is foiling all attempts to achieve the task. The two important points are: a task only succeeds after it has confirmed that it has achieved its goal, and a task only fails when it knows no way of proceeding. In all other circumstances some method is selected and executed.

### 3.4.2 Mediating Task Selection

When the RAP execution system must deal with several concurrent tasks, it is faced with the problem of choosing which to select for execution next. Two mechanisms are used to control this choice: task constraints, and selection heuristics. Constraints are added to tasks as part of the initial sketchy plans or through RAP code annotations and unless all of a task's constraints are satisfied it is considered ineligible for execution and cannot be selected. Three types of constraint are supported by the RAP system:

- Explicit ordering constraints.
- Temporal delay constraints.
- Memory content constraints.

Explicit ordering constraints force tasks to be chosen in a specific order and the second task in an ordering will not run until the first either succeeds or fails and is removed from the agenda. Temporal delay constraints force a task to wait until a specific time has passed before it can execute and memory content constraints make a task ineligible to run until specific states are satisfied in the RAP memory.

After the consideration of constraints, there may still be many tasks eligible for execution so the RAP interpreter uses the following heuristics to choose between them:

- Prefer tasks with higher priorities.
- Prefer tasks with nearer deadlines.
- Prefer unfailed tasks over failed tasks.
- Prefer tasks from the same family as the last task chosen.
- If there are still several candidates choose one at random.

Priorities allow one task to "interrupt" another so that an emergency or opportunity can be dealt with in a timely fashion. Deadlines help order task selection so that tasks that should be done sooner are executed first. These two heuristics offer explicit task selection control while the others attempt to minimize task interference. Preferring unfailed tasks over those that have failed gives interfering circumstances a chance to go away — when a method isn't working, it doesn't hurt to do something else for a while. The heuristic of choosing a task from the same family as the last has the effect of focusing the system's attention on one high level task at a time. This prevents concurrent tasks from inadvertently interfering with each other in destructive ways.

### 3.4.3 Executing a Primitive Action

The execution system assumes that memory update occurs automatically during the execution of primitive actions and hence is independent of the RAP interpreter itself. The implementation of the system used for examples in this thesis includes a hardware interface between the interpreter and the simulated robot truck hardware to do the memory update. The interface employs a simple algorithm based on a library of primitive action and sensor datum handlers. Each primitive action is passed to the robot hardware and, when execution is complete, returns success or failure along with a set of sensor notifications. The interface calls the appropriate action handler on the the result and then loops through the sensor notifications calling the appropriate sensor datum handler on each one. Each handler is a piece of LISP code that alters the RAP memory so that it is consistent with the world information implied by the corresponding action result or sensor datum. Handlers are also responsible for generating the memory update operations required to keep expectation sets current

(see Appendix A). The important point is that memory update occurs only when the robot takes an action in the world and actually learns something new.

### 3.4.4 Summary

The implicit looping behavior of a single task is what gives the RAP execution system robust situation-driven execution behavior in the face of incorrect knowledge, primitive action failure, and emergency situations. Incorrect assertions in the RAP memory will generally manifest themselves by enabling methods that would not be applicable if the actual situation were known. Eventually this will result in the execution of a primitive action that fails but produces new information that corrects the memory assertions. Meanwhile, the tasks involved will not be satisfied and will pick new methods based on the new, correct information and try again. Emergency situations are dealt with by setting up high priority tasks constrained not to run until the situation arises. When the situation does arise, the task becomes eligible to run and its high priority lets it take control immediately.

While these properties of the execution algorithm allow robust behavior, the RAP code describing the tasks must be structured to take advantage of the robustness. Appropriate task constraints and methods must be coded into tasks so that they respond effectively to situation changes and primitive action failures. The RAP language supports the description of such tasks and includes facilities for specifying preconditions, constraints, priorities, deadlines, explicit loops and counting resources. The next chapter describes the RAP language in detail.

# Chapter 4

# The RAP Language

The final component of the RAP situation-driven execution system that must be described is the library of RAP descriptions. As new subgoals are generated by the instantiation of methods by the interpreter, each must be paired with RAP code to define its execution behavior. This code resides in the RAP library in the form of RAP definitions. Every task to be executed by the system must have a defining RAP in the library.

A RAP definition contains several different clauses that describe different aspects of its task's behavior at execution-time. The basic syntax, including each type of clause, is shown in Figure 4.1. The INDEX clause is mandatory and uniquely identifies each RAP type. The PRECONDITIONS and CONSTRAINTS describe situations in which the RAP can be used and the SUCCEED clause describes situations where the RAP has completed it's goal.

A RAP may contain many METHOD clauses describing different ways to accomplish its goal. Each method contains sub-clauses to define the context in which it is applicable, global restrictions required while it is executing, and the actual steps it embodies. The MONITOR-STATE, MONITOR-TIME and REPEAT-WHILE clauses allow the RAP to synchronize itself with conditions in the world, either by waiting for a situation to occur, or by looping until a situation changes. The RESOURCES clause defines temporary data-structures RAPs can use to track the allocation of resources.

This chapter begins with a discussion of the formulae that appear in most RAP clauses, their use as memory queries and the way variables are bound. This discussion

```
(DEFINE-RAP
   (INDEX index)
   (DURATION estimated execution time)

   (MONITOR-STATE formula)
   (MONITOR-TIME  reference delay deadline)
   (REPEAT-WHILE  formula)

   (SUCCEED       formula)
   (PRECONDITIONS formula)
   (CONSTRAINTS   formula)

   (RESOURCES internal resource definitions)

   (METHOD plan for carrying out RAP behavior)
   (METHOD plan for carrying out RAP behavior)
   ...)
```

**Figure 4.1**: The RAP Definition Syntax

is followed by a detailed description of each facet of the RAP representation language. A small portion of the language has already been described in Chapter 3 but it is included again here for completeness.

## 4.1 Queries to RAP Memory

Examination of the RAP definition outline above will show that almost every clause includes a formula to be unified with the RAP memory. A formula is meant to represent a state of the world and takes the form of a query to memory asking whether the state is believed true. Queries may contain unbound variables to be treated as existentially quantified when the memory query is made. When a query succeeds, the free variables in the formula become bound and the the bindings play an important role in RAP execution. Variable binding lifetimes are discussed at length in the sections to follow.

Formulae are constructed as RAP memory queries using the constructs discussed in Chapter 2 and Appendix A. Queries consist of propositions and functions, the special form `believe`, and the connectives `and`, `or` and `not`. The propositions and functions

in queries usually refer to assertions about the world state in memory and are subject to the default rules described in Section 2.2. However, tasks occasionally wish to alter their behavior based on their current execution history. In some situations, a method within a RAP may only need to be tried once or should be used only after another method has failed in a particular way. Special query functions are included to let RAP formulae query task execution history in these situations.

### 4.1.1 Basic Queries

The basic RAP query consists of a proposition corresponding to a single memory assertion. The actual assertions expected in memory depend on the domain and are completely independent of the RAP execution system. Queries involving memory assertions can bind free variables. Examples of basic queries are:

```
(color item-34 red)
(color ?item red)
```

The special form `believe`, introduced in Chapter 2, can also be used as a basic query:

(`believe` *proposition time-interval*)

where the proposition is to match an atomic formula in memory. When the time interval is bound, either by being specified as an integer or by being a variable bound to an integer, the `believe` succeeds if an assertion matching the proposition has been made to memory within the specified amount of time. For example:

```
(believe (color item-34 red) 45)
(believe (color item-34 red) ?time)
```

will behave identically assuming `?time` is bound to 45. It will succeed if `item-34`'s color has been asserted as red within the last 45 time units and it will fail if a contradicting assertion has been made within the time interval. If no assertion has been made within the interval, and the default value for the proposition matches the proposition, then the query will also succeed.

If the time interval in a believe form is a free variable, the form becomes a query asking when a matching assertion was made. If an assertion matching the proposition exists in the database, the `believe` will succeed with the variable bound to the time interval since the assertion was last made. For example, if the color of `item-34` was asserted to be blue 15 time units ago, the query:

```
(believe (color item-34 ?color) ?time)
```

will succeed with `?color` bound to `blue` and `?time` bound to 15.

### 4.1.2 Queries about Task Progress

In addition to queries about assertions in the RAP memory, RAPs can make queries about the current execution history of the instantiated task they describe. Three such queries are defined as the distinguished propositions:

(`method-tried` *method-name*)
(`last-result` *symbol*)
(`result-with-method` *method-name symbol*)

The first is satisfied if the method with the given name in the RAP has been attempted at least once since the task was instantiated. Method names are assigned within RAP method clauses and the query can only make reference to methods within the same RAP.

The `last-result` query is satisfied if the result of the last method tried by this task matches the symbol given. For example, when the succeed clause is left out of a RAP definition, it defaults to:

```
(SUCCEED (last-result OKAY))
```

which means that the RAP will finish whenever the last method it tried completed successfully. At least one method must have been tried before this query will succeed. Valid symbols to use in last-result queries depend on the domain and the type of failures that can be returned by the primitive action requests. A list of possible results in the delivery truck domain is included in Appendix B. In addition to these domain-dependent failures, the domain-dependent results `OKAY`, `NO-METHOD`, `INTERFERENCE`

and FUTILE-LOOP can also occur. The symbol OKAY means that all steps in the last method completed successfully. The symbol NO-METHOD means that the last method executed failed because some step did not have an applicable method and INTERFERENCE means that the preconditions for some step in the last method were not met. The symbol FUTILE-LOOP means that a step in the last method ended up in a futile loop. When a task fails, it returns the last domain-dependent failure it received if there is one. Otherwise it returns the domain-independent reason for its failure.

The query result-with-method is true if the last result of executing the specified method matches the given symbol.

### 4.1.3 Combining Queries

RAP queries may also contain predefined functions. Functions may contain free variables but they do not bind them. Thus, variables appearing in functions must be bound by plain proposition queries in the same formula. The predefined basic functions are:

```
+ - * / < > <= >= = N=
```

The first eight of these are defined on numbers only, and the last two are defined on symbols and strings as well. Example function uses are:

```
(+ 4 5)
(= ?color red)
```

Note that the second of these is true if ?color is bound to red but will not generate the binding. The first four mathematical functions can only appear as arguments in other functions or propositions. The six boolean functions can be combined with basic queries using and, or, and not.

Basic formula are tied together using and, or and not in the obvious way; except that query success and failure are treated as true and false when they appear as arguments. In a formula, and is true if each of its constituents is true or succeeds in unifying with memory, or is true if any one of its constituents is true or succeeds, and not is true if its constituent formula is false or fails to unify. The constituents of a connective can be any valid formula, including another connective. For example:

```
(and (color item-34 red)
     (size item-34 big)
     (contains item-34 item-56 true))
(or (color item-34 red)
    (color item-34 blue))
(and (color item-34 red)
     (not (size item-34 big)))
```

As discussed in Chapter 2, the RAP memory does not allow deduction during query processing; there are no backward chaining rules. Checking query satisfaction is simply a matter of unifying formula with RAP memory. This ensures that RAP decision procedures can be executed in a timely manner. If arbitrary deduction were allowed when checking a RAP formula, it might take the interpreter an arbitrary amount of time to make a method choice. Since the system cannot be interrupted during method choice, there would be no way to respond to changes in the external world during that time. By keeping the query process simple, the system will make method choices quickly and it will always be in a position to choose an appropriate task when the world changes. The trade-off is that complicated deductions cannot be done automatically. For example, when querying whether an arm is big enough to grasp an item, it might be useful to infer that it is not if the item and the tools it requires are together too big. However, this cannot currently be done. One way to lift this limitation is to extend the arm-grasp primitive action handler to make the inference and assert the result in memory when a grasp fails (see Section 3.3.2), but that just shuffles the computation into another module. If arbitrary deduction is required, the correct approach is to extend the RAP system to include deduction as a primitive process as suggested in Section 3.3.3

## 4.2 The INDEX Clause

The first clause in every RAP definition is the INDEX. The index takes the general form:

(INDEX (*name in-vars* => *out-vars*))

where the *name* is unique to the RAP being defined and is used as its index in the RAP library. A RAP definition may take both input and output variables, with the

input variables coming first, and the output variables following the special symbol `=>`. The input variables must have current bindings when the task described by the RAP is selected from the task agenda for execution. If any of the input variables are still unbound at task execution time, the task fails with the message `INTERFERENCE`. Output variables become bound during task execution and the bindings are exported to the invoking task. These mechanisms allow the output variables of early steps within a RAP method to generate bindings for use in later steps within the same method.

For example, consider the following possible RAP indices:

```
(INDEX (handle-low-fuel))
(INDEX (arm-pickup ?arm ?thing))
(INDEX (find-object ?object => ?place))
```

The first index might be used for a RAP designed to watch for the fuel level of the robot truck to get low. It doesn't need any arguments, assuming that "low" is defined within the RAP itself. The second example is the most typical form for a RAP description and might be used to index a RAP that can be used to pick up any object with any of the robot's arms. The third example might be used to describe a RAP that tries to discover the location of an object (maybe by looking, asking someone, or reading a book) and returns that location by binding it to the variable `?place` once found.

RAPs are always referred to via their indices in method task-nets. For example, to create a method that finds an object, goes to it and picks it up with `ARM1` the following task-net might be used:

```
(TASK-NET
   (t1 (find-object ?object => ?place) (for t2))
   (t2 (goto ?place) (for t3))
   (t3 (arm-pickup ARM1 ?object)))
```

This is not a very robust method, because there are many problems that might arise between steps `t2` and `t3`. However, it shows how an early task can be used to bind a variable, in this case `?place`, for use by a later task. The variable `?object` must be bound when the find-object task first executes and, given the semantics of the RAP interpreter, that means it must be bound when this task-net is instantiated.

## 4.3 The DURATION Clause

The second clause used in RAP definitions specifies the estimated execution time required for the RAP once instantiated. The DURATION clause is optional and, if present, appears in the RAP just after the index. Duration clauses take the form:

(DURATION *expression*)

where the expression is a numerical value or a simple expression that evaluates to such a value. Because the expression within a duration clause does not correspond to a memory query, all variables contained in the expression must be bound. The duration clause is evaluated only once, at instantiation time and if it is not included in a RAP, its value defaults to zero.

The duration clause is required to aid the RAP interpreter in satisfying task deadlines. As discussed in Section 3.2.4, one of the heuristics used when selecting a task from the agenda is to evaluate the latest time that the task can be started and still satisfy its deadline. The task that must be started earliest is selected (all other things being equal). However, if the duration of a task is not known, its latest start time cannot be properly calculated. The difficulty, of course, is that a task's estimated duration must be determined before it has chosen a method. Ideally, duration estimates would be put off until a task could be expanded down to primitive actions and the duration of each step determined. However, with a situation-driven execution system, a complete plan for a task cannot be known in advance. Task selection requires a duration estimate none the less, so duration clauses can be included in RAP descriptions to give such an estimate as soon as a task is instantiated.

Execution does not break down if duration information is incorrect or not available because the deadline satisfaction heuristic is only one task selection heuristic among many. The effect of poor duration information will be a tendency to finish tasks with tight deadlines late.

## 4.4 Appropriateness Conditions

There are three clauses within a RAP definition that describe conditions that must be true for the RAP as a whole to carry out its intended behavior. The SUCCEED clause

defines the actual intended behavior, and the PRECONDITIONS and CONSTRAINTS define states of the world in which the RAP can be expected to carry out that behavior. All three of these clauses have the same syntax:

(SUCCEED *formula*)
(PRECONDITIONS *formula*)
(CONSTRAINTS *formula*)

When a RAP is selected for execution, the interpreter processes these three clauses before anything else. The SUCCEED clause is processed first by using its formula as a query to RAP memory. If the query is successful, the RAP is finished and succeeds. If the success conditions are not met, the PRECONDITIONS clause is checked against memory and if it is satisfied execution continues. If the preconditions match fails, the RAP fails immediately with the message `INTERFERENCE`. After the preconditions are checked, the CONSTRAINTS are checked too. If satisfied the RAP continues, and if unsuccessful the RAP fails with `INTERFERENCE`. A RAP's constraints act like additional preconditions but they are also inherited by all subtasks that the RAP generates. Constraints constrain every subtask of a RAP while preconditions only constrain the RAP itself.

From a planner's point of view, the SUCCEED, PRECONDITIONS and CONSTRAINTS clauses define the RAP as a planning operator.[1] The SUCCEED clause is the state that the RAP makes true in the world and the PRECONDITIONS of the RAP are those states that must be true in the world before the RAP can be expected to satisfy its SUCCEED clause. One can imagine a traditional planner that solves novel problems by starting with RAPs that produce the desired state, as defined in their SUCCEED clause, and using their PRECONDITIONS clauses to define a new world state to generate. Chaining the RAPs together this way will eventually produce a plan that moves the current state to the desired state. Given such a planner, the CONSTRAINTS clause in each RAP are additional conditions that must be true when each RAP starts. With a more hierarchical problem solver, the CONSTRAINTS describe world states that must be

[1] This is the primary reason why preconditions and constraints appear at the RAP top level rather than within RAP methods. RAPs would sometimes be clearer if constraints were placed in a method along with the method context (see Figure 5.11) but the RAP then becomes more obscure as a planning operator. In planning, preconditions and constraints are used before method selection and, thus, must represent properties of the RAP as a whole. Further experience with planners that use RAPs will be necessary to determine if it is more useful to have preconditions and constraints inside or outside of methods.

kept true throughout the execution of the RAP in question: no hierarchical expansion of the RAP is acceptable if it violates any of the constraints.

### 4.4.1 The SUCCEED Clause

The SUCCEED clause includes a formula describing a state of the world and declares that the RAP cannot succeed until that state is satisfied. An executing RAP loops through its methods until its succeed clause is satisfied. If none of the methods satisfies the succeed clause, then the RAP eventually fails. However, when the RAP succeeds, the world state described in its succeed clause must be believed in memory.

The succeed clause in a RAP is not mandatory. It may be left out as a shorthand, and if so, the RAP is taken to be successful as soon as it executes any method that does not fail, *i.e.* the RAP will finish after picking any method that runs to completion. This is a way of specifying a task that is supposed to "do something" rather than bring about a state in the world. For example, "looking around to see what is nearby" is a task that needs to be completed successfully, but which does not bring about any specific state in the world. Many simple physical tasks, such as turning all the way around once, are also like this.

The formula in the succeed clause may contain free variables (*i.e.* variables not contained in the input variables of the RAP index). When the formula is successfully matched against the memory during execution, these variables will take on bindings that remain in effect. However, when the match is successful it means that the RAP is completed and it will finish immediately. Thus, the succeed clause can only be used to bind RAP index output variables and not variables to be used further within the RAP.

### 4.4.2 The PRECONDITIONS Clause

The PRECONDITIONS clause also includes a formula describing a state of the world but it declares that the RAP is not appropriate unless that state is believed true when the RAP begins to execute. Although a RAP is supposed to be applicable in a wide variety of situations, it is a rare RAP that makes sense in all situations. For example, the RAP for picking an item up with a robot arm may contain many different methods

to use for different items or different orientations, but no method may work if the arm is not empty to start with. A RAP's preconditions can be thought of as those states of the world that must be true before *any* of the RAP's methods can be used. The advantage of using preconditions for the RAP as a whole rather than including the preconditions in every method context is improved efficiency: the preconditions can be checked only once when deciding whether the RAP should continue. The trade-off is that if a new method were to be added to the RAP via some learning algorithm, it could not be applied outside those situations defined in the RAP's preconditions.

The formula in the preconditions clause can contain free variables like those in the succeed clause. Variables bound by the preconditions formula query are never kept so the preconditions clause can never be used to bind variables for use elsewhere in the RAP.

### 4.4.3 The CONSTRAINTS Clause

The CONSTRAINTS clause declares that the RAP will work only if the state described by its formula remains true in the world during the entire execution of the RAP and its methods. To ensure that constraints are obeyed throughout the execution of a RAP they must be inherited by all subtasks spawned by the RAP methods. For example, a RAP for moving an object out of the truck at a certain location might be constrained so that it only applies as long as the truck (and hence the arm) remains at that location. When such a task executes and chooses a method with steps like pick the object up, move the arm outside the truck, and put the object down, the steps inherit the constraint and only apply as long as the truck stays at the location. Without the constraint, the truck (with the arm) might move somewhere else between the pickup and the putdown, and the object will end up in the wrong place.

For a given task, constraints are treated exactly like preconditions: they must be true when the task comes up for execution otherwise the task will fail with message `INTERFERENCE`. However, because they are inherited, a subtask will have its parent's constraints but not its parent's preconditions. Free variables in constraints are treated like those in preconditions and do not remain bound after the clause has been considered.

### 4.4.4 Examples

A RAP's succeed clause, preconditions and constraints act to determine when it should continue and when it is finished. They can be used together to generate tasks with different behaviors. For example, consider the partial RAP definition below:

```
(DEFINE-RAP
   (INDEX    (arm-unload ?arm ?object ?location))
   (SUCCEED       (location ?object ?location))
   (PRECONDITIONS (status ?arm available))
   (CONSTRAINTS   (truck-location ?location))
   ...)
```

This RAP describes a task to unload an object from the truck with a particular arm. The arm, object and the object's location are input variables, and the task will finish when it is believed that the object is at the correct location. The task has the precondition that the specified arm be available for use, and the constraint that the truck start at, and remain at, the location where the object is to be deposited. The constraint gets inherited by subtasks so this RAP's methods are free to use generic pickup and putdown operations without fear of the object ending up in the wrong place because of an interruption. More stringent preconditions might also possible be considered (such as "the object must be inside the truck to start with") but if the RAP is interrupted and has to start again, such restrictions inhibit graceful recovery.

As another example, the following two pieces of code do *almost* the same thing:

```
(DEFINE-RAP
   (CONSTRAINTS (truck-location warehouse))
   (METHOD
      (TASK-NET
         (t1 (pickup-at  box-34 bay1) (for t2))
         (t2 (putdown-at box-34 external)))))

(DEFINE-RAP
   (PRECONDITIONS (truck-location warehouse))
   (METHOD
      (TASK-NET
         (t1 (pickup-at  box-34 bay1)
             ((truck-location warehouse) for t2))
         (t2 (putdown-at box-34 external)))))
```

The first defines a task that is only appropriate if the truck is at the warehouse and that any subtasks it generates must also take place at the warehouse. Therefore, the pickup and putdown tasks in the method will fail if the truck is not at the warehouse when they come up for execution. The second RAP also defines a task that is only appropriate if the truck is at the warehouse when it is choosing a method. The task-net also includes an explicit annotation to the effect that task `t2` should have the precondition that the truck be at the warehouse.

There are two differences between these two RAPs, however. One is that the constraint specified in the first is inherited by its subtasks so all steps required by the pickup and putdown method steps will also fail if the truck is not at the warehouse. The second RAP does not make this demand; it only requires the truck to be at the warehouse at method choice time and when the putdown subtask executes. The second difference is that the first RAP requires the pickup subtask to occur at the warehouse but the second does not. The precondition in the second RAP means that method choice must take place at the warehouse, and the annotation means that the putdown step also must take place at the warehouse. However, nothing in the second RAP prevents the truck from doing the first pickup task somewhere else if it moves between method instantiation and execution of the pickup subtask.

## 4.5 Specifying a RAP Method

Methods are represented as separate METHOD clauses within a RAP, and each contains a CONTEXT and a TASK-NET or PRIMITIVE. The context specifies those situations in which the method is appropriate and the task-net or primitive describes the actual sub-plan that makes up the method. The basic syntax for a method is:

```
(METHOD name
   (CONTEXT formula)
   (TASK-NET task network entries))
```

or

```
(METHOD name
   (CONTEXT formula)
   (PRIMITIVE primitive action request))
```

The name of the method is an optional symbol that distinguishes it from other methods in the same RAP. This name is the argument used in the task-execution-history queries described in Section 4.1 above.

### 4.5.1 Specifying a CONTEXT

The CONTEXT for a method is a formula that must be satisfied in the RAP memory before the method is appropriate. The formula is meant to describe those situations in which the the task-net in the method is a valid plan for carrying out the RAP's desired behavior. A method context need only be true when the method is chosen and not throughout its execution (*i.e.*, a context is like a precondition and not a constraint). During execution, the RAP interpreter looks at each method's context (after checking to make sure the RAP as a whole is still valid) and chooses one whose context is satisfied. For example, a method that is applicable when `arm1` is holding something might have a context like:

```
(CONTEXT (arm-holding arm1 ?thing))
```

which will only be satisfied when there exists some object such that `arm1` is holding it.

When the formula in the context is matched against memory, free variables will become bound and the bindings are kept for use in the contained task-net or primitive action request. In the example above, `?thing` is a free variable and after the match it will be bound to whatever object `arm1` is holding. The associated task-net can perform operations on that object by referring to `?thing`.

The context clause may be left out of a RAP method and results in a method that is always applicable.

### 4.5.2 Specifying a TASK-NET

The TASK-NET clause inside a method is used to specify a plan for carrying out the RAP's intended behavior. In general, a task-net contains a number of steps that need to be executed and each step is a separate entry in the net. Special annotations are

used to connect the various steps into a partially ordered network, specify preconditions one step sets up for another, and post metric temporal constraints on steps. The basic syntax for a task-net is:

```
(TASK-NET
   (step1-tag priority rap-index annotations)
   (step2-tag priority rap-index annotations)
   ...)
```

Each entry in the task-net has a tag, a priority, a RAP index specifying the subtask to create, and a list of annotations.

The tag in a task-net entry is a symbol used to uniquely identify the step in the task-net – each step must have a different tag. A step also has an optional *priority* consisting of a single integer value. This value is added to the priority of the enclosing RAP to give a new priority for the subtask instantiated by the step. The default priority value is zero which gives the task-net step the same priority as its parent. Priorities are discussed in detail in Section 3.2.4.

The RAP index in a step is a pattern that must match a RAP index exactly. The specification may contain variables, but they must be bound by the time the subtask spawned by the step is executed. Input variables in the second or later steps may be unbound at the time the task-net is spawned, but they must become bound, as output variables in earlier steps, before execution of the step begins.

Task-net steps are connected together to form a plan using *annotations*. It is the annotations that specify ordering constraints, protections and metric temporal constraints. Annotations are also used to specify some types of subtask effects (see Section 4.8 below). There are four basic types of temporal annotation that can be attached to a step:

```
(formula FOR tag)
(UNTIL-START tag)
(UNTIL-END tag)
(WINDOW tag start-time stop-time)
```

Any number of these may be combined in any order in any step. All tags however, must refer to other steps in the same task-net. FOR, UNTIL-START and UNTIL-END

clauses add ordering constraints to tasks while WINDOW clauses add temporal constraints and deadlines. The influence of these constraints and deadlines on processing of the task agenda is described in Section 3.2.4

### The FOR Annotation

The FOR annotation performs two functions: it specifies a task ordering constraint and it describes an execution-time "protection". The intended intuitive meaning is that the step being annotated is designed to generate the specified formula for the tagged step. For example, a task-net used in fueling up in the delivery truck domain is:

```
(task-net
   (t1 (select-fuel-drum external => ?drum)
       ((liquid-held ?drum fuel) for t2))
   (t2 (pickup ?drum)
       ((arm-holding ?arm ?drum) for t3))
   (t3 (pour-into ?drum fuel-bay) (for t4))
   (t4 (putdown-at ?drum external)))))
```

This task-net generates four separate tasks. The first looks around outside the truck until it finds a drum containing fuel. It binds the drum found to the variable `?drum` which is then used throughout the rest of the task-net. The second task picks the drum up using either arm, the third task dumps the contents of the drum into the truck's fuel tank, and the final task puts the empty drum down outside the truck.

The annotations on the tasks specify the order in which the four tasks should execute and also set up extra preconditions on the second and third tasks. Task `t1` ensures that the selected drum is holding fuel and adds a precondition to task `t2` that it still be holding fuel when `t2` begins to execute. Task `t2` then makes sure that some arm is holding the drum for task `t3`. If either of these preconditions is not satisfied when the appropriate task executes, the method will fail with message `INTERFERENCE`. Note that the formula in a FOR annotation is a RAP query and can bind variables for use by later tasks in the net. Thus, `?arm` will be bound to whichever arm is actually holding the drum and might have been used in the indices in tasks `t3` or `t4`.

### The UNTIL-START and UNTIL-END Annotations

The UNTIL-START and UNTIL-END annotations terminate the step they are attached to when the tagged step they point to either starts or finishes. For example, suppose two containers are to be filled at the same time but both filling operations are to stop as soon as one container is full. The following task-net fragment might be appropriate:

```
(TASK-NET
   (t1 (fill ?container1)
       (UNTIL-END t2))
   (t2 (fill ?container2)
       (UNTIL-END t1)))
```

These constructs are particularly useful when used in conjunction with monitor RAPs that start and finish when situations become true in the world (see Section 4.7 for more examples).

### The WINDOW Annotation

The WINDOW task-net annotation is used to specify metric temporal constraints between the relative execution time of two tasks. For example,

```
(TASK-NET
   (t1 (put-in ?cake oven))
   (t2 (take-out ?cake oven)
       (WINDOW t1 25 30)))
```

declares that the cake must be taken out of the oven at least 25, but not later than 30, time units after it was put in. That is, the take-out task must be executed during the temporal window 25 to 30 time units after the put-in task finishes. Instead of the name of another step in the task-net, the tag in a WINDOW annotation may also take on the special values `NOW` and `ABSOLUTE`. The first of these means that the reference point for the temporal window should be the instantiation time of the task-net rather than the end of a step within it, and the second means that the temporal window is in absolute time units rather than relative to an event.

As an example, consider the problem of reacting when enemy troops are detected in the delivery-truck domain. There are two ways to react to enemy troops: fight

or flee. Fighting has the advantage of clearing out the troops so that afterwards the truck can carry on with whatever it was doing before. However, if the enemy is not defeated in a reasonable time, they are likely to capture the truck and render it useless. Thus, an appropriate response is to fight for a while and if that isn't working, run away. Such behavior is encoded the method shown below:

```
(METHOD
   (CONTEXT (and (location ?thing external)
                 (class ?thing enemy-troops true)))
   (TASK-NET
      (t1 (shoot ?thing)
          (UNTIL-END t2))
      (t2 1 (run-away-from-enemy)
          (UNTIL-END t1)
          (WINDOW NOW 18 25))))
```

This method is applicable whenever there are enemy troops outside the truck, and contains two steps: shoot and run away. The shoot step can start anytime after the task-net is chosen, and it can continue until the flee step ends. The run step, on the other hand, is restricted to starting at least 18 time units after the method is chosen. This delay gives the shoot step a chance to do its work. If fighting is not working, 18 time units after the method was started the run step will become eligible to run and, since it has a higher priority than the fight step, will interrupt the fighting and take control. If either fighting or fleeing successfully eliminates enemy troops from outside the truck, both steps will finish together. A subtlety is that either the fight or the flee step must have a temporal window constraint to ensure prompt action in response to the enemy troops. Other tasks with the same priority may be competing for execution time and indirectly delay execution of both fighting and fleeing. By putting a deadline on fleeing (*i.e.* that it should finish before 25 time units have passed), this problem is overcome.

### 4.5.3 Specifying a PRIMITIVE

A method that contains a PRIMITIVE clause generates a primitive action request to carry out the task. Each primitive action clause describes a single action request to be passed to the hardware interface. For example, `arm-toggle` is a primitive action

for the simulated robot truck and the lowest level RAP to initiate an item activity (like shooting) contains the method:

```
(METHOD
   (CONTEXT (and (or (arm-at ?arm ?thing)
                     (arm-holding ?arm ?thing))
                 (know-sensor-name ?thing TRUE)))
   (PRIMITIVE (arm-toggle ?arm ?thing))))
```

This method is only applicable if the sensor name for the object to be toggled is known and if the arm is positioned appropriately. Note that the sensor name might not be known if the object is in an old expectation set and has not yet been matched to a currently accessible item. More sensing would be required in that situation.

## 4.6 The REPEAT-WHILE Clause

Another type of behavior that RAPs must be able to describe is looping. There are many tasks that require steps be repeated until some state is achieved in the world. For example, filling a gasoline tank from fuel drums involves emptying drums into the tank until the tank is full. Similarly, moving all of the boxes from the warehouse to the factory requires making trips until there are no more boxes in the warehouse. Many behaviors of this type can be encoded as *implicit* loops that rely on the RAP interpreter to generate the required repetition at execution time. For example, the following RAP definition might be used to fill a fuel tank:

```
(DEFINE-RAP
   (INDEX (fill-tank ?tank))
   (SUCCEED (full ?tank))
   (METHOD
      (CONTEXT (and (class ?drum fuel-drum true)
                    (full ?drum)))
      (TASK-NET
         (t1 (pour-into ?drum ?tank)))))
```

The RAP interpreter will continue to try the single method over and over until the succeed clause is satisfied. This has the effect of implicitly looping over "empty a fuel drum into the tank" until the the tank is full. However, in many situations it is

better to use an explicit construct that will inform a planner that a RAP is designed to have intentional looping behavior.

The RAP language uses the optional REPEAT-WHILE clause to explicitly declare looping RAPs. The repeat clause may be inserted at the RAP top level before method descriptions and has the syntax:

(REPEAT-WHILE *formula*)

Intuitively, this clause says that the RAP should continue to retry its methods as long as the *formula* is satisfied in memory. Technically, things are complicated by the fact that a repeat clause, in some sense, is competing with the succeed clause that appears in the same RAP. The succeed clause states that the RAP should finish when its formula becomes true, and the repeat clause states that the RAP should not finish while its formula remains true; if the two clauses conflict, there is a problem. This difficulty is resolved by giving the succeed clause precedence over the repeat clause. The repeat clause can then be characterized more precisely as "it is okay to retry the RAP methods until the RAP goal is achieved as long as the contained formula is true".

The reason why the repeat clause is repeat-while and not repeat-until is a result of the futile loop detection algorithm. It is possible for a RAP to get stuck looping forever when using a repeat clause if the RAP succeed clause is never satisfied. To avoid getting stuck in a futile explicit loop, the RAP interpreter must restrict the repeat clause in the same way that it restricts repeated method selection. Each time the formula in the repeat clause is checked, free variables are bound and the bindings are saved with those from the method chosen. If the same bindings repeatedly occur when the same methods are chosen, then the interpreter terminates the task. This restriction has two consequences. First, the formula in the repeat clause must be carefully written so that each time through the loop it will produce different variable bindings (*i.e.* the formula is not just a termination condition, but a measure of progress within the loop). Second, the repeat clause must be a repeat-while so that at each iteration the formula will be satisfied and variables bound.

Given the loop detection restrictions on the REPEAT-WHILE clause, it becomes simply a notational device like the PRECONDITIONS clause. Since the repeat clause must be satisfied for the task to continue in choosing a method, and its variable bindings are preserved with those from the method that gets chosen, it could be

placed in each method context and produce the same results. The difference between it and the preconditions clause is that variables bound during a repeat query are preserved until method instantiation is complete. The primary reason for having an explicit repeat clause is thus to notify a planner that the RAP is explicitly designed to loop. Implicit loops are very hard to reason about so making them explicit makes a planner's job easier.

Using an explicit loop, the tank filling RAP might be rewritten as:

```
(DEFINE-RAP
   (INDEX (fill-tank ?tank))
   (SUCCEED (full ?tank))
   (REPEAT-WHILE (and (class ?drum FUEL-DRUM)
                      (full ?drum)))
   (METHOD
      (TASK-NET
         (t1 (pour-into ?drum ?tank)))))
```

## 4.7 Monitors and Synchronizing with the World

The RAP language, as described so far, allows only the description of behaviors that depend on the order of step execution for their correctness. All changes to the world are produced by the methods themselves and are assumed to be under complete control of the robot. However, it is often the case that robot actions must be synchronized to changes in the world that are not under the robot's control. Such situations range from waiting for water to boil before adding the rice, to running away when enemy troops suddenly appear.

The concept of *monitoring* is used to synchronize RAP execution with changes in the world.[2] A task can monitor a state by suspending its execution until that state becomes true in the world. Monitoring is mediated through the mechanism of memory content constraints discussed in Section 3.2.4. A task waiting for some state of the world sits on the task agenda, and is ineligible to run, until the expected state becomes satisfied in the RAP memory.

[2]This terminology derives from a very similar concept developed McDermott [1977].

### 4.7.1 The MONITOR Clauses

The two constructs for setting up a monitor tasks are the RAP clauses:

(MONITOR-STATE *formula*)
(MONITOR-TIME *reference start-time end-time*)

The first of these causes execution of the RAP to suspend until the *formula* is satisfied in memory, and the second causes the RAP to suspend for a specified period of time. The time delay in a monitor-time clause takes the same form as the time window in a WINDOW task-net annotation. The reference can be either NOW, referring to task instantiation time, or ABSOLUTE. When the reference is NOW the start-time is the interval that the task must wait before continuing and the end-time is the interval to a deadline. When the reference is ABSOLUTE, the start-time and end-time are in absolute time units. Both monitor clauses are optional top-level forms that appear in a RAP before any method clauses. If both clauses appear in the same RAP, the RAP will remain suspended until both clauses become satisfied at the same time. Free variables in a MONITOR-STATE formula do not remain bound.

Monitor clauses change the way that RAPs are chosen from the execution agenda. Monitor-state clauses set up memory content constraints, and monitor-time clauses set up temporal constraints and task deadlines. Monitor clauses and task-net priorities work hand-in-hand to give RAPs powerful means for watching for emergencies, restoring important states, and looking for opportunities.

### 4.7.2 Active vs. Passive Monitors

Monitor-state clauses suspend a task until a formula becomes true in memory, but what happens when the formula being monitored requires active sensing to detect? For example, one might desire to set up a RAP to watch for fires in a box, and extinguish them should they appear. However, if detecting a fire in the box requires periodically opening the lid, a simple RAP monitoring "there is a fire in the box" is not sufficient. Such situations require active monitors to do repeated sensing as opposed passive monitors that can simply wait for a state to be detected in the normal course of events.

An active monitor must be represented with a RAP that carries out the required sensing, waits for some time to pass, and then senses again. Such behavior is easily generated using the MONITOR-TIME clause. Consider the following RAP for watching for a fire in a box:

```
(DEFINE-RAP
   (INDEX (watch-for-fire ?box))
   (SUCCEED nil)
   (MONITOR-TIME NOW 5 10)
   (METHOD
      (TASK-NET
         (t1 (look-inside ?box) FOR t2)
         (t2 (put-out-fire ?box)))))
```

This RAP describes a task that never finishes (because its succeed clause is never satisfied) and will "wake-up" every five time units, look into the box and put out any fires found there. We assume that if no fire is present the task to put the fire out will succeed immediately without taking any action. While active monitors like this are necessary in situations where the normal course of events does not generate the requisite sensory data, they are not responsive to sensory data generated by other tasks. For instance, if the box is opened to put something inside and a fire is seen 1 time unit after the above monitor task has just run, there will be no attempt to put the fire out until 4 more time units have passed and the monitor task wakes up again.

To respond to sensory data arriving from many sources, passive monitors defined with the MONITOR-STATE clause are more appropriate. Sensor information from any robot activity can satisfy the monitored state. The only problem occurs when no normal robot activity generates the appropriate information. The state being monitored would then come true in the world but never be noted in the RAP memory. In situations where the facts must be constantly monitored, but where other activities may generate the same information, the best solution is a combination of monitors. Considering the box/fire example again, one might use the following two RAPs together:

```
(DEFINE-RAP
   (INDEX (check-for-fire-in-box ?box))
   (SUCCEED nil)
   (MONITOR-TIME NOW 5 10)
```

```
      (METHOD
         (TASK-NET
            (t1 (look-inside ?box)))))

   (DEFINE-RAP
      (INDEX (monitor-inside-fire ?box))
      (SUCCEED nil)
      (MONITOR-STATE (fire-inside ?box))
      (METHOD
         (TASK-NET
            (t1 (put-out-fire ?box)))))
```

The first RAP repeatedly checks inside the box for a fire, and the second will put out a fire in the box as soon as it is noticed, no matter where the information comes from.

A final point about monitor tasks is their behavior with respect to futile loop detection. The very nature of such tasks is to continually watch for some situation to develop and then act on it. In most cases, monitor tasks are intended to loop forever as long as they continue to effectively deal with the situation they monitor each time it is encountered. Therefore, each time a monitor task actually blocks (*i.e.* becomes ineligible to run because the state it is monitoring is not true), the interpreter restarts its algorithm for futile loop detection for that task. This has the effect of allowing a monitor to loop forever providing its methods make the monitored state false each time it executes. Tasks including the monitor-time clause can never cause a futile loop because they will always block (assuming a start-time greater than zero).

### 4.7.3 Protections and Opportunities

Monitor RAPs, such as the fire-detecting example above, are obviously useful for dealing with sudden situation changes that threaten particular goals. Monitor RAPs, however, can be used for other purposes as well: the protection of world states, and the exploitation of opportunities.

#### Enforcing Protections

An execution time protection (without look-ahead) is somewhat different from the protections traditionally used during planning. Planning protections are set up to

prevent the addition of a plan step that alters a state produced by a previous plan step. At execution time such a protection is impossible to enforce because some other agent might do the altering. However, there are many different ways that such a protection might be approximated at run-time and several are discussed in Chapter 6. This section will discuss only one: restoring the required state whenever it is discovered to have changed. For example, suppose the truck picks up a fuel drum to deliver to the fuel depot. Along the way, the truck encounters a box it has been looking for and puts the fuel drum down to pick up the box. If the truck now continues on its way, the fuel drum will be left behind. The fact that the truck is holding the fuel drum should be protected and "repaired" before setting off for the depot.

```
(DEFINE-RAP
   (INDEX   (monitor-holding ?arm ?object))
   (MONITOR-STATE (not (arm-holding ?arm ?object)))
   (METHOD
      (TASK-NET
         (t1 (arm-pickup ?arm ?object)))))

(DEFINE-RAP
   (INDEX   (carry-object ?object ?location))
   (SUCCEED (location ?object ?location))
   (METHOD
      (CONTEXT (arm-holding ?arm ?object))
      (TASK-NET
         (t1 1 (monitor-holding ?arm ?object)
               (UNTIL-END t2))
         (t2 (go-to ?location)))))
```

**Figure 4.2**: An Example of a Protection RAP

State-restoration protections are set up using tasks monitoring the complement of the state to be protected. When the monitored formula becomes satisfied, the protected fact has ceased being true and the RAP takes some action to restore it. Consider a monitor designed to protect an arm-holding fact like that in Figure 4.2. When the carry task is used to move a fuel drum to the fuel depot, it sets up a monitor to for the fuel drum being set down. Should the fuel drum ever be set down, the monitor responds by picking it up again. This example is somewhat simple

minded and may cause loops where the truck puts the fuel drum down in service of one goal, picks it up again, sets it down again, *ad infinitum*.[3]

### Detecting Opportunities

Another use for monitor tasks is exploiting execution time opportunities. An opportunity arises when a method is selected for the pursuit of a goal and new information arrives later that suggests the use of a different, "better" method. To detect opportunities, a task can be set up to watch for the information that would enable selection of the preferred method. Should that information be detected, the monitoring task will interrupt execution of the current method and allow method reselection.

An example of opportunism required in the delivery truck domain arises when sending the truck to fetch an object. Suppose the truck is requested to get a fuel drum and take it to the warehouse. The closest fuel drum known is across town so the truck sets off to retrieve that one. The truck then runs across a fuel drum along the way that will do the job. Assuming that the method chosen for fetching the fuel drum breaks down into steps like `(goto across-town)`, `(pickup fuel-drum)`, `(goto warehouse)`, the new fuel drum will be discovered during the first goto. However, since the RAP describing that task says nothing about fuel drums, the truck will simply pass it by. To take advantage of the opportunity, a different method for going across town must be used that includes a monitor for the appearance of a new fuel drum as shown in Figure 4.3.

If the move-object RAP is used to move the fuel drum to the warehouse, it will break the task down into the same three steps: go to the fuel drum, pick it up, and go to the warehouse. However, a monitor is also set up in parallel with the first steps to watch for the appearance of a fuel drum along the way. Should a fuel drum be discovered, the monitor RAP will wake up, pickup the drum, succeed and the method will skip ahead to step `t4`.

[3]Schoppers discusses infinite loops caused by this type of execution time protection strategy in Schoppers [1987a].

```
(DEFINE-RAP
   (INDEX   (monitor-appearance ?type))
   (MONITOR-STATE (and (location ?object external)
                       (class ?object ?type)))
   (METHOD
     (CONTEXT (and (location ?object external)
                   (class ?object ?type true)))
     (TASK-NET
        (t1 (pickup ?object external)))))

(DEFINE-RAP
  (INDEX   (move-object ?type ?location))
  (SUCCEED (and (location ?object ?location)
                (class ?object ?type true)))
  (METHOD
     (CONTEXT (and (location ?obj1 ?loc1)
                   (class ?obj1 ?type)
                   (not (location ?obj2 external)
                        (class ?obj2 ?type))))
     (TASK-NET
        (t1 1 (monitor-appearance ?type)
              (UNTIL-END t3))
        (t2 0 (go-to ?loc1)
              (UNTIL-END t1)
              ((truck-location ?loc1) FOR t3))
        (t3 0 (pickup ?obj1 OUTSIDE)
              (UNTIL-END t1)
              ((arm-holding ?arm ?obj1) FOR t4))
        (t4 0 (go-to ?location))))
  (METHOD
     (CONTEXT (and (location ?object external)
                   (class ?object ?type true)))
     (TASK-NET
        (t1 0 (pickup ?object external)
              ((arm-holding ?arm ?object) FOR t2))
        (t2 0 (go-to ?location)))))
```

**Figure 4.3**: Noticing an Opportunity with RAPs

## 4.8 Short-Term Memory and Internal Resources

Some behaviors that one would like to express in the RAP language require the use of short-term memory. An underlying principle of situation-driven execution is that the world is its own best model: RAPs should not try and keep track of the changes they make to the world, but rather should continuously sense the world and see the actual changes that have taken place. This idea works well for a great many tasks but some behaviors require keeping track of abstract properties, like sets, from one step to another. Consider two different tasks: the first requires keeping track of the amount of fuel in a fuel drum, and the second requires moving all of the fuel drums from the warehouse to the garage. The first can be achieved without any short-term memory because the fuel drum can be examined directly whenever the amount of fuel inside needs to be checked. The second, however, requires keeping track of which fuel drums inside the truck are from the warehouse and which are being used for other purposes.

The usual way for RAP methods to pass information from one step to another is through the use of index variables. Each RAP can take input variables (and bind output variables) and act differently based on the bindings of those variables. The need for short-term memory arises when the number of index variables needed cannot be determined in advance (*i.e.* how many members a set will have) or when the value of a variable binding must be changed inside the RAP (*i.e.* when counting). The RAP language handles these situations by allowing the creation of set and counter resources that can be bound to index variables and shared between RAPs. In this way, a counter can be defined to hold the number of fuel drums to be moved from the warehouse to the garage and it can be shared by both the load and unload subtasks. The load can increment the counter each time it places a fuel drum in the truck and the unload can decrement it each time it takes one out. That way the same number of fuel drums gets unloaded at the garage as were loaded at the warehouse.

### 4.8.1 The RESOURCES Clause

Counters and sets are called *resources*. Resources are defined at the RAP top level, before any methods, using the syntax:

```
(RESOURCES (SET ?name value)
           (COUNTER ?name value)
           ...)
```

A resource clause is executed only once, when the task defined by the RAP is first instantiated, and may contain any number of SET and COUNTER declarations. A set declaration causes a new set resource to be created and bound to the variable name it is paired with. The optional second argument is a previously defined set to be copied. If no argument is given, the set starts out empty. A counter declaration causes the creation of a new counter resource with the initial value given as either a number or another counter to be copied.

Resources are altered using special annotations in task-net step specifications. These annotations are called *effects* and at execution-time they change the values of the specified resources. There are two different effects for counters and two for sets:

| | | |
|---|---|---|
| (SET-ADD ?set *thing*) | - | *add something to a set resource* |
| (SET-SUB ?set *thing*) | - | *remove something from a set resource* |
| (COUNTER-ADD ?counter *amount*) | - | *increment a counter resource* |
| (COUNTER-SUB ?counter *amount*) | - | *decrement a counter resource* |

Each effect either adds or removes an object from a set, or increments or decrements a counter. The effects annotations of a RAP step are only executed after the step itself succeeds. Thus, resources are only altered on the successful completion of a task.

To make reference to resource values, the following functions can be placed in any RAP formula used as a memory query. Each function has the obvious semantics:

| | | |
|---|---|---|
| (MEMBER ?set *thing*) | - | *does* ?set *contain* thing |
| (EMPTY ?set) | - | *is* ?set *empty* |
| (SIZE-OF ?set) | - | *the number of members in* ?set |
| (VALUE ?counter) | - | *the value of* ?counter |

Using internal resources, the problem of moving fuel drums from one place to another without disturbing other drums in the cargo bays can be solved with the three RAPs shown in Figure 4.4. The first RAP describes the overall delivery task and consists of the obvious four steps: go to where the items are to leave from, pick them all up, go to their destination, and put them all down. The important feature

of this RAP is the definition of the counter resource `?number` to be shared between the load and unload subtasks. The RAP for the load task picks up every item of the appropriate type and places them in the truck's cargo bay. Furthermore, each time that an item is successfully placed in the cargo bay, the task-net COUNTER-ADD annotation increments `?number`. Thus, at the completion of the load task, the `?number` counter will hold the number of items of the requested type actually found and picked up. The unload task then puts down as many items of the appropriate type as indicated by the counter. If other tasks have placed identical items in the cargo bay for other reasons, they will not be unloaded by mistake.

Notice that defining the counter in the delivery RAP means that the value of the counter survives any failures that might occur during the go to, load, or unload subtasks. For example, suppose the truck is given the task of moving all of the medical boxes from location A to location D. A deliver-all task is instantiated and the truck moves to location A and finds 6 boxes there. However, after picking up only 3, enemy troops show up and the truck is forced to flee. Later, the pickup subtask comes up for execution again and fails because the truck is not at location A. The deliver-all task now runs again and the truck sets off to A once more. The problem is that 3 boxes were already loaded into the truck and they will not be found at A. This situation is dealt with properly because the counter is defined at the deliver level and does not get redefined by the failure. Thus, the second time the method is used, the counter will start out saying three boxes are already in the truck. At A the truck will load 3 more boxes bringing the counter up to 6 and then go to location D. At D the unload task will unload all 6 boxes because that is the number the counter says are in the cargo bay.

### 4.8.2 Resources vs. Execution Strategies

In some cases, RAPs can be written to use features of the real world rather than internal resources to keep track of item allocations. For example, all the fuel drums to be delivered to the factory could be placed at the front of the cargo bay and all of those for the warehouse could be placed at the back. That way, even though the drums are identical, they will not get mixed up. The essence of such a strategy is to use some property of the world to make indistinguishable items distinguishable.

```
(DEFINE-RAP
   (INDEX (deliver-all ?type ?from ?to))
   (RESOURCES (COUNTER ?number 0))
   (METHOD
      (TASK-NET
         (t1 (go-to ?from)
             ((truck-location ?from) (FOR t2)))
         (t2 (load ?type ?number ?from) (FOR t3))
         (t3 (go-to ?to)
             ((truck-location ?to) (FOR t4)))
         (t4 (unload ?type ?number ?to)))))

(DEFINE-RAP
   (INDEX (load ?type ?number ?place))
   (SUCCEED (or (not (and (location ?thing external)
                          (class ?thing ?type true)))
                (too-full-for cargo-bay ?type)))
   (CONSTRAINT (truck-location ?place))
   (REPEAT-WHILE (and (location ?thing external)
                      (class ?thing ?type true)))
   (METHOD
      (TASK-NET
         (t1 (pickup-at ?thing external)
             ((arm-holding ?arm ?thing) FOR t2))
         (t2 (putdown-at ?thing cargo-bay)
             (COUNTER-ADD ?number 1)))))

(DEFINE-RAP
   (INDEX (unload ?type ?number ?place))
   (SUCCEED (or (not (and (location ?thing cargo-bay)
                          (class ?thing ?type true)))
                (<= (VALUE ?number) 0)))
   (CONSTRAINT (truck-location ?place))
   (REPEAT-WHILE (and (location ?thing cargo-bay)
                      (class ?thing ?type true)))
   (METHOD
      (TASK-NET
         (t1 (pickup-at ?thing cargo-bay)
             ((arm-holding ?arm ?thing) (FOR t2)))
         (t2 (putdown-at ?thing external)
             (COUNTER-DEL ?number 1)))))
```

**Figure 4.4**: Delivering a Load of Items

Location, orientation, and inclusion in a spatial group, are all physical properties that can be used to keep identical items apart in the real world and people use them as memory aids all the time. More intrusive strategies are also used when confusion can cause problems. Labelling, color coding, and packaging are all ways to make it easier to tell which items in the world are being used for which tasks. My mittens had my name sewn in them when I was in first grade so that it was easy to distinguish them from the other childrens' even when they looked the same. The members of my rock climbing group mark their gear with colored stripes so that it easy to sort out after a day on the rock. Even the president has a red phone for making important calls (well, he does in the movies). All of these labels could be avoided with more memory use: my mittens were usually with my coat, the rock climbing gear doesn't really all look the same, and the red phone is probably the one on the left. However, people seem to run into a memory capacity limit in such situations that makes it much more reliable to use the real world to do the work.

An interesting experiment would be to limit the number of counter and set resources that are available for use by RAPs. Currently, there is no limit and it is as easy for the execution system to "remember" a hundred different sets of items as it is to remember one. If resources were limited, however, the system would have to be allocate them more carefully and make more use of physical strategies as memory aids instead. People do seem to have such limits and are forced into physical strategies all of the time. Fortunately, there is no shortage of reality.

## 4.9 Protections and Policies

The last construct in the RAP definition language implements temporary global policies. Suppose that a task requires absolute quiet when using a particular method. Given the language so far, the steps themselves can be chosen to be quiet ones, either by explicitly specifying RAPs that use quiet methods, or by passing in an index value that signals the spawned RAPs to be quiet. However, there is no way to signal *all* RAPs, including those in service of other tasks, that they should choose quiet methods.

### 4.9.1 The NOTATIONS Clause

Such signals are implemented using the NOTATIONS clause. Notations clauses appear only within method specifications between the context and task-net clauses and have the syntax:

```
(METHOD
   (CONTEXT ...)
   (NOTATIONS assertion assertion ...)
   (TASK-NET ...))
```

Each assertion in the clause must be a simple proposition containing only bound variables which can be asserted to the memory database. When the assertions are made, they are are scoped over the method so that when the method finishes, either normally or because of a step failure, they are retracted again. For the life-time of the task-net, the notations become part of the memory for any other tasks to see.

Using notations, the need to be quiet might be coded as:

```
(DEFINE-RAP
   ...
   (METHOD
      (CONTEXT ...)
      (NOTATIONS (be-quiet true))
      (TASK-NET ...)))
```

During the execution of the steps in the task-net, the assertion `(be-quiet true)` will be in memory, and any other RAPs that care to check will be able to choose quiet methods. This use of a notation implements some aspects of the idea of planning *policies* discussed in [McDermott, 1978] and [Charniak and McDermott, 1985]. Monitor tasks may also be awakened by the assertion of a notation.

## 4.10 Summary of the RAP Language

In summary, the RAP language can be loosely divided into five sections which exist for separate reasons:

- The core language — INDEX, SUCCEED, METHOD

- Efficiency checks — CONSTRAINTS, the FOR annotation
- Synchronization — MONITOR-STATE, MONITOR-TIME
- Inter-task communication — RESOURCES, NOTATIONS
- Planning annotations — PRECONDITIONS, REPEAT-WHILE

The core language supports basic situation driven execution behavior as described in Chapter 1, but it must be augmented to overcome specific problems. The constraints clause and FOR task-net annotation set up conditions on method subtasks to detect situations when a method is not behaving as expected. Early detection of such situations allows the method to be aborted and prevents inappropriate subtasks from executing pointlessly. Monitor clauses allow tasks to synchronize their execution with situations out in the world, and resource clauses allow the subtasks within a method to make and share short-term records of resource usage. Finally, the preconditions and repeat-while clauses allow RAPs to be coded in a more perspicuous fashion, easing the job of both the initial RAP writers and planners that wish to examine RAP internals.

The core language supports all of the basic notions required in a situation-driven execution system. The succeed clause specifies the world state that must be achieved before the defined task can be considered complete. Even if a selected method runs to completion without failure, the task spawning it will not finish unless the desired success conditions have been met. The method clauses specify all of the ways known to carry out a task. Each method describes a network of subtasks to be executed in service of the task and can be thought of as a hierarchical expansion of the task. Different methods will be applicable in different circumstances and methods are repeatedly tried until the task succeeds. Along with the RAP interpreter algorithms for task selection and futile loop detection, the core language alone achieves situation-driven execution behavior.

One problem with an execution system based on only the core RAP language and interpreter, however, is its inclination to execute method subtasks even after it becomes clear that continuing on with the method is pointless. For example, consider the following method:

```
(METHOD
```

```
(CONTEXT (location ?object external))
(TASK-NET
   (t1 (arm-move ARM1 ?object))
   (t2 (arm-grasp ARM1 ?object))))
```

The first subtask is designed specifically to position the arm so that it can grasp the desired object. If all goes well, there will be no problem and the second subtask will succeed in picking the object up. However, if there is an interruption between the two subtasks, ARM1 may get used for something else and moved away from the object to be grasped. When the interruption passes, the second subtask will come up for execution and the arm grasp primitive will be executed even though it is clear that it cannot work because the arm is not in the right place. This problem arises whenever methods are interrupted and is a result of the reliance on task expansion as the mechanism for achieving a goal. Instantiating methods and waiting for them to complete is rather like calling a set of subroutines that must run to completion before control gets returned to the calling task.To get around this problem, constraints clauses and FOR annotations are used to set up "check points" in the method so that subtasks will stop executing when the situation becomes inappropriate. Such check points prevent execution of a method from continuing when it becomes clear it isn't working out.

Another issue not addressed by the core RAP language is that of synchronizing task execution to uncontrolled states in the world. The two monitor clauses allow RAPs to define tasks that are constrained to wait until some state is satisfied in the world. Careful use of these features enables the description of tasks that "wakeup" in response to emergencies or other situations that should be dealt with immediately. Similarly, tasks can be set up to specifically look for opportunities to improve overall behavior by becoming active when new, helpful information becomes available. Without monitor RAPs, the execution system would be unable to deal effectively with changing situations.

The core language, in concert with monitor clauses, give tasks the ability to conform their behavior to the current state of the world. Method contexts specify world states in which particular methods apply, and monitor clauses let tasks wait until particular states in the world become known. However, when RAP defined behaviors must be altered by abstract decisions made by other RAPs, resource and notation

clauses must be used as well. Resources allow subtasks in a method to share the members sets and the values of counters. Notations allow RAPs to notify each other about more global restrictions on execution by making temporary additions to the RAP memory. Resources allow tasks within a method to communicate while notations allow completely independent tasks to communicate as well.

Finally, the preconditions and repeat-while clauses are used to annotate a RAP so that its internal workings are more accessible to examination by a planner. Preconditions gather up those facets of the world state that must be satisfied before the RAP can be applied at all. Such preconditions can be pushed into method contexts, but it is often clearer to make them more explicit. Similarly, the repeat-while clause in a RAP makes it plain that the described task is expected to loop as long as the given state remains true in the world. Again, the same information can be pushed into method contexts and generate identical behavior. However, is often useful for a planner to know that a particular RAP will loop without having to try and derive it from implicit information.

### 4.10.1 RAPs and Sketchy Plans

This chapter, along with Chapters 2 and 3, finishes the description of the three main divisions in the RAP execution system: the RAP memory, the RAP interpreter, and the language used to define the RAP library. The final issue to be dealt with is the way a top-level sketchy plan should be defined. The answer is as a RAP — the best way for a planner to generate instructions for the RAP system is to build a RAP to be executed. The method (or methods) inside the RAP would consist of a task-network build by the planner, and the RAP succeed and monitor clauses would describe when the plan should execute and when it has achieved its goals. Along with the RAP to be instantiated, the planner need only supply an appropriate priority and deadline for the resulting task. The notion of sketchy plan and RAP are identical.

There also remain several interesting issues that fall outside of the description of the RAP system itself. The first of these are issues in representation. Does the execution system and language as defined allow a resonable number of realistic execution behaviors to be encoded as RAP tasks? Chapter 5 explores techniques for representing different types of RAP behavior, giving many examples of execution time protections,

opportunity recognition, and short-term memory strategies in particular.

Another interesting issue is that of testing the RAP system to show that it actually works. Since a real robot was not readily available, experiments were done using a simulated robot delivery truck in a complex, changing domain. The results of these experiments are presented in Chapter 6. Execution traces are also provided to illustrate the generation of different types of behavior. The results show that the RAP system, and the paradigm of situation-driven execution, actually do effectively deal with uncertain knowledge of the world, effector failures, and interruptions.

# Chapter 5

# Issues in RAP Representation

The RAP situation-driven execution system described in the preceding chapters is designed to cope with the problems of acting in a complex, dynamic world. However, a robust execution system and expressive action description language are only part of the solution. There remains the problem of building effective action descriptions for the RAP library when confronted with uncertain sensory information and autonomous agents. Most planning research done in the past has assumed the planner is the only actor in a completely predictable world. In such situations, a library of relatively straightforward task descriptions will generate correct behaviors [Sacerdoti, 1975, Vere, 1983, Wilkins, 1985, Dean *et al.*, 1987]. For example, a typical plan for taking a drum of fuel to the factory is:

```
goto depot -> pickup a drum -> goto factory -> putdown drum
```

Intuitively, this simple plan captures all of the necessary actions — you go find a drum of fuel, pick it up, carry it to the factory, and then put it down. Unfortunately, the intuitive effectiveness of this plan rests heavily on the assumption that the world is completely under the planner's control. When confronted with a more realistic world, complexities begin to creep into such a plan and it rapidly loses its intuitive feel.

There are four main difficulties that arise when writing adaptive action packages for the RAP library:

- Making sensing strategies explicit

- Enabling opportunity recognition
- Preventing undesirable states of the world
- Coordinating the reactions to various situation changes

These difficulties do not so much result in problems that need solution as introduce subtleties into behavior description that must be understood and dealt with. This chapter is intended primarily to point out some of these subtleties and show how they affect the representation of many apparently straightforward tasks. Along the way, many example RAPs are presented and some general techniques for representing behaviors are described.

## 5.1 Sensing Strategies

The first consideration when constructing a task representation is that explicit sensing subtasks must be incorporated into the task's methods. Tasks must make sure that enough sensing requests are generated to keep relevant parts of RAP memory up-to-date. In particular, sensing steps must be included to:

- Correct uncertain and missing knowledge in memory.
- Enable item identification across expectation sets.
- Monitor dynamic situations

These problems arise, in one form or another, when interacting with any complex, dynamic domain.

Uncertain and missing knowledge is unavoidable in the RAP memory because the system cannot know the entire state of the world. Therefore, explicit sensing tasks are incorporated into task descriptions to gather direct knowledge of the world before it is needed to make method choices or succeed clause checks. For instance, in the delivery truck domain, the truck looks around to see what new items have become accessible when it changes location. The truck may also look inside boxes to see if they are empty or look at the indicator lights on a machine to see if it is working. Such routine sensing tasks fill in missing world knowledge.

As discussed in Chapter 2, the RAP memory forms expectation sets to hold the descriptions of items it expects to encounter in a particular situation. When the situation is encountered, the memory automatically tries to match these expected descriptions to the actual items present in the world. Often, unambiguous matches cannot be made on the basis of available information and the memory ends up holding many independent descriptions for a single real item: the description most recently attained through direct sensing along with descriptions from past expectation sets. Multiple item descriptions cause confusion because all are available to match queries, but only *one* is real. Sensing tasks must be used to prevent such confusion by gathering enough information to ensure description matching across expectation sets.

The problem of situation monitoring arises when other agents in the world can change that part of the world accessible to the system. Sensing operations to fill in knowledge gaps and enable item identification help the system build a detailed picture of the directly accessible situation, but when that situation can change on its own, additional sensing is necessary to ensure that significant changes are noticed. Checking that a box is empty does not mean that it will stay empty, or even stay accessible, when other agents are around. The sensing required while pursuing a single task does not generally track every important feature of the immediate situation so such features must be monitored separately.

### 5.1.1 Routine Memory Maintenance

Uncertain and incomplete world knowledge is the driving force behind situation-driven execution. The items actually at a location, the contents of a box, and the current weather are all facts that must be sensed before they can be known for certain. Since RAP succeed clauses and method contexts depend on this information being in memory, the RAPs must contain sensing tasks to gather it. The routine memory-maintenance problem is deciding what sensing tasks are required for a particular behavior.

Several authors have explored methods for automatically adding sensing tasks to more traditional plans. Brooks [1982] considers the idea of predicting actuator positioning uncertainties and adding sensing tasks to measure the actual position when the uncertainty exceeds acceptable limits. A model developed in Gini *et al.* [1985]

uses planner intent to add sensor tasks designed to ensure the world is unfolding as planned and Doyle [1986] reports a system that does the same thing using an analysis of volatile preconditions. These methods make the assumption that an initial plan can be built, and that sensing is required only to verify that execution is proceeding as planned. The difficulty is in choosing appropriate sensing tasks from an operator library in advance.

The Bumpers scheduler developed in Miller [1985] takes a different approach and assumes the operators used to generate the initial plans already include appropriate sensing tasks. Previous approaches attempt to produce minimal sensing strategies, but Miller's operators are idiosyncratic in that they use sensing wherever it seems required. The justification for such *ad-hoc* operators is the observation that the sensing strategies needed to support a given task are as task- and domain-dependent as the task's actions. The RAP system takes essentially the same approach — routine sensing tasks are built into RAP descriptions whenever they might be necessary.

Even though routine sensing tasks are spread through RAPs in an *ad-hoc* manner, they are used in three basic ways:

- To always check a specific fact for a specific method.
- To check critical facts only when they become too uncertain.
- To perform routine sensing as a service to the whole system.

The specific color sensing task used when loading rocks into the truck is shown in Figure 5.1. The RAP shown is designed to load as many rocks of a particular color as possible into a cargo bay. A repeat-while loop is used, and the method in the RAP is retried as long as there is a rock outside the truck that might be the correct color. The first step in the method is to check the color, and a `FOR` annotation is used to reject the rock if it is the wrong color. This routine check is cheap and doing it all the time minimizes the effect of memory errors. Note that a rock will only be examined once because the examination places its color into the RAP memory. The actual rocks loaded are also recorded in a set resource for use in an unloading task later. The point is that the eye-examine sensing task is simply part of the behavior needed to load colored rocks into the truck.

```
(DEFINE-RAP
   (INDEX     (truck-load-rocks-here ?here ?color ?bay ?rocks))
   (SUCCEED   (or (not (and (location ?thing external)
                            (class ?thing rock true)
                            (or (color ?thing ?color)
                                (color ?thing unknown))))
                  (>= (size-of ?rocks) 8)))
   (CONSTRAINTS (truck-location ?here))
   (REPEAT-WHILE (and  (location ?thing external))
                       (class ?thing rock true)
                       (or (color ?thing ?color)
                           (color ?thing unknown)))
   (METHOD
      (CONTEXT (and (location ?thing external))
                    (class ?thing rock true)
                    (or (color ?thing ?color)
                        (color ?thing unknown)))
      (TASK-NET
         (t1 (eye-examine ?thing)
             ((color ?thing ?color) for t2))
         (t2 (make-room-in-bay ?bay ?rocks)
             ((< (number-in-bay ?bay) 8) for t3))
         (t3 (move-thing-to ?thing ?bay)
             (set-add ?rocks ?thing)))))
```

**Figure 5.1**: Routine Sensing to Check a Specific Fact

Sensing tasks can also be used more sparingly when some uncertainty can be tolerated. The RAP shown in Figure 5.2 describes a task for dumping a fuel drum into the truck. If a fuel drum is known to be empty it is ignored. If a drum has been checked within the last 10 time units and it contained fuel, then it is used without checking again. If the drum was last checked more than 10 time units ago, or has never been checked, then it is examined before being picked up. This behavior assumes that a drum that recently held fuel probably still does and checking it again is not strictly necessary.

As a final routine sensing task, consider the problem of determining the objects that are currently accessible to the delivery truck. The need to know which items surround the truck is so pervasive that it is reasonable to check them whenever the truck changes location. Thus, the RAPs that move the delivery truck from place to

```
(DEFINE-RAP
   (INDEX   (dump-in-one-fuel-drum))
   (METHOD
      (CONTEXT (and (location ?drum external)
                    (liquid-held ?drum fuel)
                     (believe (quantity-held ?drum ?amount) 10)
                     (and (not (= ?amount unknown))
                          (> ?amount 0))))
      (TASK-NET
         (t1 (pickup ?drum)
             ((arm-holding ?arm ?drum) for t2))
         (t2 (pour-into ?drum fuel-bay) (for t3))
         (t3 (putdown-at ?drum external))))
   (METHOD
      (CONTEXT (and (location ?drum external)
                    (liquid-held ?drum FUEL)
                    (or (quantity-held ?drum unknown)
                        (not (believe (quantity-held ?drum ?amount) 10)))))
      (TASK-NET
         (t0 (eye-examine ?drum)
             ((and (believe (quantity-held ?drum ?amount) 10)
                    (not (= ?amount unknown))
                    (> ?amount 0)) for t1))
         (t1 (pickup ?drum)
             ((arm-holding ?arm ?drum) for t2))
         (t2 (pour-into ?drum fuel-bay) (for t3))
         (t3 (putdown-at ?drum external)))))
```

**Figure 5.2**: Routine Sensing to Confirm an Uncertain Fact

place always generate eye-scan primitive action requests at new locations. The RAP in Figure 5.3 shows part of one such movement task.

### 5.1.2 Item Identification

Consider a robot that requires a full fuel drum and notes that `item-34` in memory is known to be a full fuel drum located at the depot. Upon actually moving to the depot, the robot encounters two fuel drums and the hardware interface assigns them sensor-names `drum-1` and `drum-2`. The drums can now be directly manipulated via the names `drum-1` and `drum-2` but not yet via the name `item-34`. They can be distinguished by their sensor-names because those names are intended to reflect

```
(DEFINE-RAP
   (INDEX   (truck-move-in-direction ?direction ?to-place))
   (SUCCEED (truck-location ?to-place))
   (PRECONDITIONS (and (my-fuel ?level)
                       (> ?level 0)))
   (METHOD
      (CONTEXT (direction external ?road ?direction))
      (TASK-NET
         (t0 (pull-in-arm arm1) (for t2))
         (t1 (pull-in-arm arm2) (for t2))
         (t2 (truck-set-speed   ?road) (for t4))
         (t3 (truck-set-heading ?direction)  (for t4))
         (t4 (truck-move-p) (for t5))
         (t5 (eye-scan-p external)))))
```

**Figure 5.3**: Routine Sensing to Stay in Touch with the World

detailed position (*i.e.*, one might be on the left and one on the right). On the other hand, `item-34` is "the full fuel drum", and it is not yet known which drum is full. Before "the full fuel drum", *i.e.* `item-34`, can be picked up, it must be identified with either `drum-1` or `drum2`. To force the identification, a RAP must issue instructions to look inside the drums to see which is full. While the details of this problem depend on the structure of the RAP memory, the general problem of identifying past item descriptions with objects in the real world will arise for *any* execution system working with realistic sensors.

There has been a lot of philosophical work done on the problem of determining which object an internal representation denotes in the real world. That work has a long and interesting history that can be explored starting from Charniak and McDermott [1985] or Genesereth and Nilsson [1987]. However, the item identification problem faced by the RAP system is really deciding which representations denote the same real object when all are consistent with it. Any solution to this problem must inevitably include the generation of additional sensing tasks to discriminate among the possible candidates in memory. There has been a good deal of work done on this problem in computer vision research [Brooks, 1984, Lowe, 1987], but it assumes that enough information will be generated automatically. An exception, is a system described in Allen and Bajcsy [1985] that generates specific

sensory requests to confirm a tentative match. In the RAP system, the sensory system does not always deliver enough information to match descriptions with real objects without additional sensing.

In the example above, there is no way to tell before arriving at the depot whether `item-34` will require special sensing to identify it. The robot must wait and see what other fuel drums are encountered and how the memory system matches them up. Recall from Chapter 2 that the memory supplies the required information through the special query form:

(`know-sensor-name` *item status*)

where item is the memory name to be checked and status can be `TRUE` meaning that the item has been identified with a real object or `FALSE` meaning it hasn't. RAPs that must refer to actual objects (such as those making primitive action requests) will have to spawn methods that include extra sensing tasks if this assertion is not satisfied for the item in question.

As an example, consider the RAPs from the robot delivery truck domain shown in Figures 5.4 and 5.5. These RAPs encode the task of turning an item on or off in the world using the arm-toggle primitive action. The task designed to be used by other RAPs is described by the arm-toggle-p RAP which contains two methods, one to be used when the item to be toggled has been identified with a real object and the other to used when it has not. The first method simply spawns a task to issue the arm toggle primitive action request. The second method cannot make a primitive action request directly because there is no way to refer to the appropriate object in the real world. Instead, two tasks are spawned: one to force identification of the item to be toggled with a real object, and the other to request the toggling action. The RAP describing the force identification operation is not all shown because it is quite complex, containing different methods for different classes of items. For example, boxes and drums must have their contents checked while rocks must have their size and color examined. The force-identification RAP loops over each item in memory that has a sensor-name and is of the right class. The properties relevant to the calls in question are sensed during each iteration of the loop and, at some point, the memory will receive the piece of information it needs. At that point the required

```
(DEFINE-RAP
   (INDEX   (arm-toggle-p ?arm ?thing))
   (PRECONDITIONS (or (arm-at ?arm ?thing)
                      (arm-holding ?arm ?thing)))
   (METHOD
      (CONTEXT (KNOW-SENSOR-NAME ?thing true))
      (TASK-NET
         (t1 (real-arm-toggle-p ?arm ?thing))))
   (METHOD
      (CONTEXT (not (KNOW-SENSOR-NAME ?thing true)))
      (TASK-NET
         (t1 (force-identification ?thing)
             ((KNOW-SENSOR-NAME ?thing true) for t2))
         (t2 (real-arm-toggle-p ?arm ?thing)))))

(DEFINE-RAP
   (INDEX   (real-arm-toggle-p ?arm ?thing))
   (METHOD
      (CONTEXT (and (or (arm-at ?arm ?thing)
                        (arm-holding ?arm ?thing))
                    (KNOW-SENSOR-NAME ?thing true)))
      (PRIMITIVE (arm-toggle ?arm ?thing))))
```

**Figure 5.4**: Encoding the Primitive Toggle Operation

`know-sensor-name` assertion will become true in memory and the force-identification task will succeed.

### A Complex Example

The problem of item identification shows up clearly in the delivery truck RAP shown in Figure 5.6. The purpose of this RAP is quite simple. A cargo bay in the delivery truck can hold only eight items so tasks must sometimes make room for a new item. The RAP shown is designed to take items out of the bay until there is room, but without removing items in the set `?objects-to-stay`. This RAP is used when loading a number of rocks into a cargo bay and when more room is needed, rocks just put in should not be taken out. The load-rocks RAP keeps track of rocks it puts in the bay and uses them as the set of objects not to be removed.

```
(DEFINE-RAP
   (INDEX   (force-identification ?thing))
   (SUCCEED (KNOW-SENSOR-NAME ?thing TRUE))
   (REPEAT-WHILE (and (KNOW-SENSOR-NAME ?object true)
                      (not (examined-since-move ?object))
                      (class ?thing  ?class true)
                      (class ?object ?class true)))
   (METHOD
      (CONTEXT (and (arm-holding ?arm ?object)
                    (KNOW-SENSOR-NAME ?object true)
                    (not (examined-since-move ?object))
                    (class ?thing  ?class true)
                    (class ?object ?class true)
                    (not (class ?object container true))))
      (TASK-NET
         (t1 (eye-examine ?object)))) ; Get external characteristics
   (METHOD
      (CONTEXT (and (arm-holding ?arm ?object)
                    (KNOW-SENSOR-NAME ?object true)
                    (not (examined-since-move ?object))
                    (class ?thing  ?class true)
                    (class ?object ?class true)
                    (class ?object container true)))
      (TASK-NET
         (t1 (eye-examine ?object))     ; Get external characteristics
         (t2 (arm-examine ?object))))) ; See what is inside
   (METHOD -other object types and locations-)
```

**Figure 5.5**: Forcing Item Identification in Memory with a RAP

The first method in the RAP applies when the cargo bay contains a removable item without a known sensor-name. When this method is instantiated, it generates tasks to determine the item's sensor-name and then move it out of the bay. There is a subtle but important reason for the force-identification task to appear explicitly in this method. If the force-identification is not done at this level, it must eventually occur as a subtask of the move task so the item can be picked up. The problem is that when the identification does happen, the item might end up matched with an object that is not to be removed. To prevent such a match from causing problems, the force-identification is done explicitly and an annotation is used to stop the method if the match is undesirable.

```
(DEFINE-RAP
   (INDEX   (make-room-in-bay ?bay ?object-to-stay))
   (SUCCEED (or (<  (number-in-bay ?bay) 8)
                (>= (size-of ?objects-to-stay) 8)))
   (REPEAT-WHILE (and (>= (number-in-bay ?bay) 8)
                      (location ?object ?bay)
                      (not (MEMBER ?objects-to-stay ?object))))
   (METHOD
      (CONTEXT (and (location ?thing ?bay)
                    (not (MEMBER ?objects-to-stay ?thing))
                    (not (KNOW-SENSOR-NAME ?thing))))
      (TASK-NET
         (t1 (force-identification ?thing)
             ((not (MEMBER ?objects-to-stay ?thing)) for t2))
         (t2 (move-thing-to ?thing external))))
   (METHOD
      (CONTEXT (and (location ?thing ?bay)
                    (not (MEMBER ?objects-to-stay ?thing))
                    (KNOW-SENSOR-NAME ?thing)))
      (TASk-NET
         (t1 (eye-examine ?thing)
             ((not (MEMBER ?objects-to-stay ?thing)) for t2))
         (t2 (move-thing-to ?thing external)))))
```

**Figure 5.6**: A RAP Made Bizarre as a Result of Sensing Problems

The second method seems as if it shouldn't suffer from this problem because it refers to items with known sensor-names. However, the set of items to stay in the bay may also contain memory descriptions that have not yet been matched to real objects in the bay — causing the same problem in reverse. The most conservative strategy would be to force identification of every item not to be removed, but that would often result in a lot of unnecessary sensing. To strike a compromise, this method includes a sensing task to find out as much about each item to be removed as it easily can (such as size, color, *etc.*) If a problem identification takes place, the annotation in the task-net stops the method before a mistake is made. If no identification takes place, it is assumed that the item can safely be removed. Mistakes can happen if the eye-examine task does not generate enough information to make all appropriate identifications in memory and, in practice, the truck does occasionally get confused and takes incorrect items out of the cargo bay.

### 5.1.3 Situation Monitoring

The situation monitoring problem arises in dynamic domains when accessible items come and go unpredictably. Such comings and goings can cause the system to believe that items are present when they aren't and to be unaware of items when they are present. Incorrectly believing items to be present can result in failed primitive action requests while unnoticed items can result in missed opportunities and possible disaster. In general, the situation-driven execution model handles primitive action failure appropriately by updating the local memory model and choosing new methods for the failed task. Detecting the presence of newly arrived items is more troublesome, however. There is no general mechanism built into the RAP system for keeping track of the local situation if the changes are not directly relevant to current task activities.

For example, watching for a full fuel drum to be discovered while travelling to the depot to get one implies that some process will be looking around and registering the discovery of fuel drums. The routine eye-scan tasks intrinsic to movement tasks in the delivery truck domain do some of this, but they will not detect fuel drums that arrive after the truck has looked around. To maintain a reasonable routine level of awareness in the deliver truck domain, a routine active monitor must exist to "wake up" every now and then and update the local memory model. The RAP describing that task is shown in Figure 5.7. This routine scanning task is placed on the task agenda and given a very high priority so it will track the comings and goings of items in the world regardless of other tasks that are present. When the world changes too quickly between these routine scans, the execution system gets quite confused, missing opportunities and emergencies, and spending a great deal of its time trying to manipulate objects that are no longer there.

Previous work in situation monitoring has taken the same approach of using active monitors as the RAP system. Important features of the world are identified in advance, and a two tiered structure is used to keep track of them. First, some method of occasionally (or continually) scanning the world is assumed to generate a picture of the current situation, and to identify agents that might cause change. A process is then instantiated to watch for each agent to change the situation. Each process is assumed to run concurrently with others and produce timely notifications when important features of the world change. Handler and Sanborn [1987] and Arkin [1987]

```
(DEFINE-RAP
   (INDEX     (monitor-current-location))
   (SUCCEED   (not T))
   (MONITOR-STATE (not (believe (eye-scanned external) 15)))
   (METHOD
      (TASK-NET
         (t1 (eye-scan-p EXTERNAL)))))
```

**Figure 5.7**: A RAP to Check the Local Situation Every 15 Time Units

describe such two tiered systems while Slack and Miller [1987] assumes the existence of such a system. The object-following robot described in Horswill and Brooks [1988] was conceived with a very different model in mind, but still contains a process to scan for an object to follow and another that watches for the object to change position.

### 5.1.4 Identification Errors

A final subtle sensing problem that arises in the delivery truck domain results from an interaction between the need to force memory-description identification and the fact that items are often moved around without the system's knowledge. The problem is that item descriptions in memory are sometimes identified with the wrong real world object. A typical example is that of looking for a fuel drum with an "X" on the bottom. If a single fuel drum was seen at the factory last time it was visited and that drum had an "X" on the bottom, the memory will automatically identify a new drum seen there with the old description. This can cause problems if fuel drums have been switched. Suppose that `item-65` in memory is believed to be a drum with an "X" residing at the factory. Upon arrival, the truck does a routine eye-scan, sees only one drum called `drum-1`, and memory identifies it with `item-65`, the one expected fuel drum. The assertion:

```
(know-sensor-name item-65 true)
```

is now satisfied in memory — as is the assertion brought forward from `item-65`:[1]

```
(has-x-on-bottom item-65 true)
```

[1] Hopefully something less bogus than `has-x-on-bottom` would actually appear in memory.

The truck believes that `drum-1` and `item-65` describe the same fuel drum and that drum has an "X" on the bottom. However, another agent may have changed drums so that `drum-1` is not the drum referred denoted by `item-65`. If the truck believes its memory, it will be wrong.

Item identification errors are unavoidable when dealing with any dynamic world that requires the use of memory. Unfortunately, of all the sensing problems that arise in dealing with the world, this is the least amenable to general solution. The only answer is to make sure that when an item with a special property is required, that property is specifically checked. Special RAPs have to be written to carry out the required additional sensing. These RAPs will have to judge for themselves whether the added cost of additional sensing is better than accepting whatever risk is involved in being wrong. If the drum at the factory is needed only because it is a drum, it makes no difference whether it is the one with the "X" on the bottom of not. On the other hand, if the drum with the "X" holds something of great importance, it will be best to check its bottom.

### 5.1.5 Sensory Issues

There are two main points to make with respect to sensory issues and the RAP system. The first is that task descriptions must explicitly deal with three complications:

1. Routine sensing is required to ensure that the parts of RAP memory relevant to method choices are up-to-date.

2. When other agents can alter the current situation, additional sensing is required to monitor the current situation and prevent mistakes.

3. The memory system will sometimes make errors in item identification.

Planning systems usually finesse these complications by assuming a completely predictable future, but in a complex, dynamic world they cannot be avoided.

The second point is that AI research has little experience with the sensing strategies required to support reasonable behavior in dynamic domains. While introspection often supplies a good starting point for writing down the actions needed to perform a

task, it supplies very little help with the sensing that must go along with the actions. When I think about moving a box from the floor to the table, it is clear that I must pick it up, move it up to the table, and put it down again. However, the sensing I do in concert with the actions is completely situation-driven and performed without advanced thought. This is almost certainly because sensing and acting are tightly coupled and our descriptions need only speak about the actions. When we actually execute the actions, the sensing required takes care of itself. Thus, when a task must be spelled out, only actions come to mind. This phenomenon supports our contention that sensing is situation-driven just like detailed action choice, but it sheds little light on the problem of developing sensing strategies for the RAP library. As a result, the sensing issues discussed above were discovered more by trial and error than by design after the system had been implemented and RAPs were being written to cope with the delivery truck domain.

## 5.2 Opportunism

An important aspect of intelligent action is the ability to exploit opportunities when they arise. Seizing an opportunity might loosely be defined as the use of previously unknown information to change one's behavior in an advantageous manner. For example, picking up a twenty dollar bill from the sidewalk, or using a newly discovered fuel drum rather than going to the depot are both examples of opportunistic execution-time behavior. These examples, however, represent two quite different ways for opportunities to relate to active execution goals. Most people will pick money up off the ground even when they have no immediate plans for its use. Money is deemed an asset of sufficient value to actively acquire whenever an honest opportunity arises. On the other hand, it only makes sense to stop and use a newly discovered fuel drum if one is already looking for one. Thus, opportunities arise both global utility considerations that are not connected to a particular goal and from temporary situations generated in the pursuit of a definite goal. In fact, the ability to seize temporarily defined opportunities is so important that any execution system must have it.

Consider the following scenario. The delivery truck needs to get a specific tool so that it can perform a task. The last place that it saw the tool was in the warehouse so it sets off to get it. Along the way to the warehouse, the truck discovers the tool in the

road. If the truck does not immediately abandon its plan of going to the warehouse and pick the tool up instead, it cannot be said to be demonstrating effective behavior. This scenario arises over and over again in dynamic domains. A specific state of the world is required and the execution system adopts the best plan it can to achieve that state. However, in the midst of executing the plan, the state becomes achievable in a much better way. The ubiquity of such *plan-revision opportunities* in the delivery truck domain makes it odd that the literature is almost silent on the issue.

Detecting and acting on opportunities at execution time should be clearly distinguished from detecting and acting on opportunities discovered at planning time. While planning, a system often generates new information about its expected future by projecting its intended actions. The opportunities that such derived information makes available for plan revision are discussed at length in Hayes-Roth and Hayes-Roth [1979]. The problem of changing ones plan in response to opportunities that arise at execution time is a quite different problem however. Birnbaum [1986] discusses the differences between execution-time and plan-time opportunism at length.

The problem of acting on *utility-based opportunities*, (*i.e.*, opportunities that allow the overall state-of-affairs to be improved, like picking money up off the ground) is somewhat more widely discussed because it is the same problem as acting in response to an emergency situation like a fire. In particular, Kaelbling [1986a] suggests the use of concurrent tasks with high priorities to detect and deal with emergencies, as does Georgeff [1986]. The RAP system uses this same approach to deal with emergency situations and with other utility-based opportunities as well. An important consequence of using concurrent tasks to recognize opportunities is that the system cannot react to situations that have not been anticipated in some way — if no task has been set up in advance, there can be no reaction. While the particular mechanisms used by the RAP system to set up such tasks are system specific, Birnbaum [1986] argues, as would I, that the need to anticipate opportunities is not an accident of system architecture, but an unavoidable consequence of the way information must be processed while interacting with the world. It is impossible for an execution system to automatically notice every aspect of the world that may be relevant to any of its current goals. Each goal must look for (Birnbaum), or set up tasks to look for (RAPs), its own opportunities.

Thus, the central characteristic of RAP system opportunism is that it is not au-

tomatic. RAPs set up their own mechanisms for recognizing, and being notified of, useful opportunities when they arise. Mechanisms for setting up notifications for both utility based and plan revision opportunities are discussed below.

### 5.2.1 Plan-Revision Opportunism

A plan revision opportunity occurs when a task is executing one method and encounters a situation that enables another, more effective method to be chosen instead. To get the appropriate behavior, a task must split into two tracks: one describing the method to use in most situations, and another to watch for the opportunity to use the more efficient method. The track watching for the opportunity must always take the form of a high-priority monitor task waiting for the opportunity is detected (see Section 4.7). These requirements define the idiom:[2]

```
(DEFINE-RAP
   (SUCCEED -goal-)
   (METHOD
      (CONTEXT -usual context-)
      (TASK-NET
         (t0 1 (monitor-for-opportunity)
             (UNTIL-END t1))
         (t1 (do-default-method)
             (UNTIL-END t0))))
   (METHOD
      (CONTEXT -context holding opportunity-)
      (TASK-NET
         (t0 (do-better-method)))))
```

When the task comes up for execution and the opportunistic situation does not exist, the first method will be chosen. That method instantiates a default plan for the task with a concurrent monitor to watch for a opportunity to arise. The opportunity monitor is generated at a higher priority than the default method so that it will interrupt should the opportunity arise. If the opportunity does arise, the monitor can either succeed or fail immediately, giving the second method a chance to execute, or it can instantiate a better method itself.

[2]The monitor task can be either active or passive within this coding idiom. However, if a strictly passive monitor is used, similar behavior can be achieved by placing constraints on the subtasks of the default method.

For example, a task to get donuts might be coded in the following manner:

```
(DEFINE-RAP
   (INDEX (get-donuts))
   (SUCCEED -have donuts-)
   (METHOD
      (CONTEXT -no donut shop in sight-)
      (TASK-NET
         (t0 1 (watch-for-donut-shop)
            (UNTIL-END t1))
         (t1 (goto supermarket)
            (UNTIL-END t0)
            (( -at supermarket-) FOR t2))
         (t2 (buy-at donuts supermarket)
             (UNTIL-END t0))))
   (METHOD
      (CONTEXT -donut shop in sight-)
      (TASK-NET
         (t1 (buy-at donuts donut-shop)))))
```

When a task described by this RAP come up for execution, the current situation is checked for a local donut shop. Should a donut shop be present, then the second method is chosen and the donuts are purchased. If a donut shop is not present, then the first method is chosen. The first method spawns a trip to the supermarket to buy inferior donuts, but also sets up a task at a higher priority to keep watch for a donut shop. If a donut shop is encountered before getting to the supermarket, the watching task wakes up, takes control because of its higher priority, buys donuts, and finishes. This completes the method and the trip to the supermarket is abandoned.

### 5.2.2 Utility-Based Opportunism

Utility-based opportunism is distinguished from plan-revision opportunism because no actions are carried out unless the opportunity actually arises. A $20.00 bill laying on the ground is an example of such an opportunity. Picking up the money is not a better method for doing something the system is already committed to do. Rather, picking up money is an example of a behavior that is good for the system to exhibit in any situation. Emergencies like encountering a fire or enemy troops are also examples of such reactions.

Utility-based opportunism is also encoded in RAPs using a single idiom. A task is placed on the agenda constrained to run only when the situation containing the opportunity arises. The task is assigned a high priority, and when the opportunity is encountered, it becomes eligible to run, interrupts whatever is being done, and performs the appropriate action. For example, the money opportunity routine might be encoded as:

```
(DEFINE-RAP
   (INDEX (pickup-money))
   (MONITOR-STATE  (-money on ground-))
   (SUCCEED (not true))
   (METHOD
      (CONTEXT  (-money on ground-))
      (TASK-NET
         (t0 (pick-up-money)))))
```

An utility-based opportunity monitor can usually be recognized by the fact that it contains a monitor-state clause triggered by the opportunity and a succeed clause that can never be satisfied. The monitor-state clause allows the task to run only when appropriate and the unsatisfiable succeed clause prevents the task from leaving the agenda. More examples of this behavior are discussed in Sections 4.7 and 5.2.4.

### 5.2.3 Examples of Opportunistic RAPs

Tasks with opportunistic behavior are used to deal with many features of the delivery truck domain. The principal use of plan-revision opportunism, however, is to take advantage of the discovery of new items when travelling from one place to another. A special case of this is embodied in the two RAPs shown in Figure 5.8. These RAPs together describe a task for going to the location of a specific item. The go-to-object RAP includes a single method appropriate when the location of the object being sought is known. The heart of the method is a task to travel to the object's location, just as one would expect. However, the method spawns two other tasks as well: one to watch for the object and another to scan the object's location upon arrival. At first glance, it might not be clear why they are required.

The monitoring task is required because the object being sought may have moved. In particular, the object may have moved away from its last location but to a new

```
(DEFINE-RAP
   (INDEX   (monitor-for-object ?object))
   (SUCCEED (believe (LOCATION ?object external) 15))
   (MONITOR-STATE (believe (LOCATION ?object external) 15)))

(DEFINE-RAP
   (INDEX   (go-to-object ?object))
   (SUCCEED (location ?object external))
   (METHOD
      (CONTEXT (location ?object ?place))
      (TASK-NET
         (t0 1 (monitor-for-object ?object)
               (until-end t1) (for t3))
         (t1 (truck-travel-to ?place)
             (until-end t0) (for t3))
         (t3 (eye-scan-p external)))))
```

**Figure 5.8**: Finding a Specific Item with RAPs

location on the way there. Without the monitor task to watch for the object in this situation, the truck will drive right past it and look in its old location. It seems to make more sense to stop when the object is encountered. The eye scan task is needed to perform the routine scanning of the object's location when the truck arrives. Normally that task would be performed by the truck-travel task, but in this case, the monitor task may interrupt the travel task and prevent it from making the scan. Together these tasks describe the opportunistic behavior required to find an object in the world — a simple form of opportunism to be sure, but a very important one.

Another example of opportunism in the delivery truck domain is illustrated by the RAPs in Figure 5.9. The two RAPs shown implement part of the machinery for keeping fuel in the truck as it moves around in the world. The handle-low-fuel RAP is the top-level task that would routinely be instantiated at a high priority on the truck's task agenda. It spawns two tasks that exist concurrently and act independently. The first of these tasks monitors the truck's fuel level and implements actions to increase it when it gets low. The actions taken usually consist of finding full fuel-drums and dumping them into the truck's fuel tank. The second task spawned has the job of

keeping a full fuel drum in the truck's cargo bay in case of emergency. If the truck is far away from any full fuel drums when its fuel level gets low, it may run out on the way to get more. As a last resort in such situations, the fix-low-fuel task will use any fuel the truck is carrying in its cargo bays to get it to the nearest fuel depot. The replenish task makes sure there is always at least one fuel drum in the cargo bay to use.

```
(DEFINE-RAP
   (INDEX   (handle-low-fuel))
   (METHOD
      (TASK-NET
         (t0 (fix-low-fuel))
         (t1 (replenish-fuel-reserve)))))

(DEFINE-RAP
   (INDEX    (replenish-fuel-reserve))
   (SUCCEED  (not true))
   (MONITOR-STATE (and (not (and (location ?drum bay1)
                                 (class ?drum fuel-drum true)))
                       (and (location ?drum external)
                            (class ?drum fuel-drum true)
                            (or (quantity-held ?drum unknown)
                                (and (quantity-held ?drum ?amount)
                                     (> ?amount 0))))))
   (METHOD
      (CONTEXT (and (location ?thing external)
                    (class ?thing fuel-drum true)
                    (quantity-held ?thing ?amount)
                    (> ?amount 0)))
      (TASK-NET
         (t2 (move-thing-to ?thing bay1))))
   (METHOD
      (CONTEXT (and (location ?thing external)
                    (class ?thing fuel-drum true)
                    (quantity-held ?thing ?amount)
                    (= ?amount unknown)))
      (TASK-NET
         (t1 (eye-examine ?thing)
             ((and (quantity-held ?thing ?some)
                   (> ?some 0)) for t2))
         (t2 (move-thing-to ?thing bay1)))))
```

**Figure 5.9**: Delivery Truck Fuel Monitoring RAPs

The replenish-fuel-reserve task is a utility-based opportunity monitor structured to pick up full fuel drums discovered in the normal course of events when there are none in the cargo bay. Note that the second method includes a task to examine the candidate drums to make sure they hold some fuel. Whenever the truck holds no reserve fuel drum, this task will begin checking each drum encountered until a full one is found and moved into the cargo bay. As long as that (or any other) fuel drum remains in reserve, the task will remain ineligible to run on the task agenda. However, as soon as the drum is used, this task will again wakeup and begin looking for a new reserve.

### 5.2.4 Opportunistic RAPs

Plan-revision opportunities arise whenever new information suggests abandoning a currently active plan in favor of a better one. In the RAP system that means stopping the execution of one task-net and instantiating another. A system that must stick to a given course of action once selected will display odd behavior in complex, dynamic domains because initial choices will often be based on inaccurate information. Utility-based opportunities arise when new information suggests taking some action to increase the robots overall well-being even though the opportunity is not directly relevant to an active plan. The RAP system implements such opportunities by placing a special task on the task agenda with no purpose beyond simply exploiting the opportunity. Utility-based opportunism is a special case of reacting to the world when it suddenly changes. That problem is discussed further in Section 5.4.1.

One type of opportunistic behavior that the RAP system cannot represent is that of choosing a low priority task over a high priority task because it suddenly becomes very easy to achieve. Consider the problem of stopping for donuts on the way to the fuel depot. In the delivery truck domain, getting fuel when the truck is low has a high priority task because running out is often fatal. Thus, when fuel gets low, the truck stops what it is doing and sets off to find some before it is too late. Suppose the truck has also been assigned the task of getting donuts. Even though such a task is obviously of lower priority than getting fuel, it does not require extra fuel to stop if a donut store is passed on the way. This type of behavior is not supported in the RAP system because it requires analysis of expected task resource use. Getting

donuts is okay because it does not compromise the goal of getting fuel. However, if the robot passes a donut store while running from enemy troops, stopping would be a terrible mistake. There is no way to distinguish between these two situations, however, without projecting the effects of stopping and comparing them with the effects of not stopping. Such an analysis is precisely what the RAP system is designed to do without. Task priorities are a heuristic designed to avoid the problem of making complex trade-off decisions and, thus, opportunities that can only be properly detected by making such trade-offs cannot be exploited.

## 5.3 Putting Constraints on the World

"Don't put the glass down", "keep the candy machine full", and "keep rats out of the factory" are all examples of goals that ask for states of the world to be maintained or prevented. It is very natural to use such constraint goals when describing many types of behavior, and an execution system will often be called on to represent them.

The most common application of behavioral constraints in the literature is as planning protections. Sussman [1975] introduced the term "protection" to describe the need for a planner to enforce the implicit constraints contained in many planning operators. For instance, the simple plan "pickup the glass", "goto the kitchen", "putdown the glass" assumes that the glass is held during the whole trip to the kitchen. Such assumptions must be protected while planning if the final plan is to behave as expected. Many planners have been built using the idea of explicit protections [Tate, 1977, Vere, 1983, Dean *et al.*, 1987], and [Charniak and McDermott, 1985] and [Joslin *et al.*, 1986] contain good reviews. The TWEAK planner described by Chapman [1985] is, in some sense, the definitive protection based planner. Protections like "don't put the glass down" arise frequently as they are implicit in virtually every multi-step method.

"Keep rats out of the factory" or "stay away from fires" cannot be treated as strict protections because they cannot generally be satisfied completely. Rats cannot always be prevented from getting into the factory on their own, but plans that give them few opportunities are better then plans that leave the door open all of the time. Similarly, fires cannot always be prevented and sometimes it is necessary get close

to one. "Keep rats out of the factory" and "stay away from fires" are more like preferences than constraints. Plans that keep one farther away from fires are better then those that don't. McDermott [1978] calls such goals policies. Their influence on planning behavior is discussed further in McDermott [1985] and Hogge [1988].

At plan time, constraints usually take the form of notations that prohibit the planner from adding steps to change the constrained facts. This method of a enforcing constraints relies on the planner being in complete control of the world. When there are no autonomous agents, a state can be maintained or prevented by the planner simply by not acting to change it. In a dynamic domain, however, many states are not under control of the system and cannot be protected simply through inaction. Making sure that a candy machine never gets empty is easy if only the planner is buying candy, but when there are other customers, the candy will be consumed unpredictably. Keeping rats out of a factory is hard because the rats will be hiding most of the time. Even those states completely under the planner's control, like holding a glass, cannot be maintained in all situations. A planner might decide to take a glass to the kitchen, but the protection that prevents it from being put down along the way can probably be ignored if a fire is discovered and both hands are needed for the extinguisher.

### 5.3.1 Constraint Goals at Execution Time

The notion of a planning constraint, as discussed above, is too strong to carry over directly into the RAP execution system. A strict constraint on a state is usually not appropriate because changing the state temporarily when the situation demands is often quite reasonable. Furthermore, a strict constraint on the world is impossible because other agents can always make changes without asking. At execution-time, planning constraints must translate from strict prohibition into a more precise description of their intent. For example, "keep rats out of the factory", might actually mean to not bring rats in, or to take rats out when they are found, or to set rat traps, or a combination of all three. Similarly, "don't put the glass down" might actually mean not to put the glass down *unless* it is picked up again before leaving. At execution time, these strategies give the system much more room to respond to other goals that might arise.

As an example, consider the problem of keeping chocolate bars in the candy machine. It will generally be impossible to guarantee that the machine always contains chocolate bars short of preventing people from using it (which might be a good strategy in some situations). Thus, the task of keeping chocolate bars in the machine must really be one of periodically checking the machine, and refilling it if necessary. There are an almost endless variety of ways to encode such a behavior depending on how critical it is that the machine never be empty:

- The machine can be checked frequently so that it is almost never empty.
- The machine can be checked infrequently, thus using less time but increasing the risk of running out of chocolate bars.
- The machine can be checked only when passing by in pursuit of another task.
- The machine can be filled only when someone complains that it is empty.
- The machine might be made to sound an alarm when the last chocolate bar is removed.

All of these strategies can be represented as RAPs once the desired behavior has been made explicit.

Unlike opportunistic task descriptions, however, there is no RAP coding idiom that captures the notion of a constraint. Constraints and `FOR` annotations can be used to stop methods when a fact that should be protected is changed through some sort of interference, but this represents only a rudimentary notion of prevention. Usually, different states to be maintained in the world require different behavioral strategies. The resources required to maintain a state must be balanced against the cost of having it change. Making sure that a bomb does not get jostled may be a crucial protection which justifies dedicated vigilance and the exclusion of other tasks, while protecting the holding of a glass is a suggestion that can be ignored with little cost in most situations. Maintaining some protections may even require actively altering the current situation. For example, preventing a cake from being crushed before a party at home may mean simply letting it sit on the table, while preventing it from being crushed during a picnic may require putting it in a special container.

While there is no coding idiom for enforcing constraints within the RAP system, the fact that other agents can always change a state means that protection strategies usually fall into two catagories: tasks to restore a state when it changes, and code distributed throughout the RAP library to prevent the state change in the first place. Which strategy is chosen depends on how much control the system has over the constraint, and the cost associated with failing to maintain it. Special code is appropriate when the system has control over the state and the cost of failure is high, and restoration tasks are necessary when the state cannot be controlled. A combination of the two is the usual course of events.

### 5.3.2 An Example in the Delivery Domain

Consider a policy of not putting ammunition boxes and fuel drums in the same delivery-truck cargo bay at the same time. We will assume that the danger of catastrophic explosion is great so the situation should never be allowed to happen. We will also assume that agents other than the truck itself can move objects in and out of the truck's bays. The first assumption means that the truck (*i.e.*, the RAP system) must structure its behavior so it never puts fuel and ammunition together under any circumstances. The second assumption means that the contents of the cargo bays can never be taken for granted. The truck might adopt general preventive strategies like putting a sign in the cargo bay warning against putting fuel and ammunition together, but in general such strategies can only decrease the likelihood of error. No guarantees can be made about another agent's behavior.

Thus, this policy requires two quite different behaviors be implemented. First, it is impossible to guarantee that fuel and ammunition will not appear in a cargo bay together so the truck should periodically check its cargo bays to make sure that no explosions are imminent. This behavior does not enforce the policy of not putting fuel and ammunition together, but it does attempt to move them apart before it is too late. Second, all RAPs that move things in and out of cargo bays must ensure that they do not put ammunition and fuel together. The next two sections go into these behaviors in some detail.

### 5.3.3 Repairing a Constraint Violation

The ammunition/fuel policy requires that no cargo bay ever contain both fuel drums and ammunition boxes at the same time. Since other agents may bring about such a state without the truck knowing about it, the truck must maintain a backup strategy of separating the two should they ever be discovered together. Such a strategy consists of two components: a passive monitor to fix the problem whenever it is discovered, and an active monitor to check the cargo bays periodically to see if the problem has arisen. One way of representing these behaviors is shown in Figure 5.10. The first RAP in the figure passively waits for fuel drums and ammunition boxes to appear in the same cargo bay, and the second wakes up every 10-15 time units and looks in a bay. If the second RAP every sees fuel and ammunition together, the first RAP will move them apart.

The first RAP has two methods. The first method moves all fuel drums out of the cargo bay, and the second moves all of the ammunition boxes out. The choice of method in any particular situation will be random, in accordance with the algorithm outlined in Section 3.2.2. Both methods assume that the remove-all-of-type task will not place the removed items in another cargo bay where the same situation will arise. A task defined with this RAP must be instantiated on the task agenda whenever fuel/ammunition interactions might arise. The task must also have a high enough priority to interrupt whatever else is happening and deal with the problem immediately.

The second RAP simply waits on the task agenda for 10-15 time units, wakes up, and looks around. A task defined by this RAP must be instantiated on the agenda for each cargo bay to be watched and these tasks must also have high enough priorities to do their job in a timely manner. An observation task for each cargo bay, working in concert with a single task to separate the problem items, enables the truck to actively recognize and react to unintentional violations of the fuel/ammunition constraint — whether the constraint was violated by another agent or by the truck itself. Such a backup strategy is often useful even when the protected state is entirely under the execution system's control.

```
(DEFINE-RAP
   (INDEX (deal-with-fuel-ammo-interaction))
   (SUCCEED (not true))
   (MONITOR-STATE (and (location ?thing1 ?bay)
                       (class ?thing1 fuel-drum)
                       (location ?thing2 ?bay)
                       (class ?thing2 ammo-box)))
   (METHOD
      (CONTEXT (and (location ?thing1 ?bay)
                    (class ?thing1 fuel-drum)
                    (location ?thing2 ?bay)
                    (class ?thing2 ammo-box)))
      (TASK-NET
         (t1 (remove-all-of-type ?bay fuel-drum))))
   (METHOD
      (CONTEXT (and (location ?thing1 ?bay)
                    (class ?thing1 fuel-drum)
                    (location ?thing2 ?bay)
                    (class ?thing2 ammo-box)))
      (TASK-NET
         (t1 (remove-all-of-type ?bay ammo-box)))))

(DEFINE-RAP
   (INDEX (monitor-cargo-bay ?bay))
   (SUCCEED (not true))
   (MONITOR-TIME (now 10 15))
   (METHOD
      (TASK-NET
         (t1 (eye-scan-p ?bay)))))
```

**Figure 5.10**: Watching for an Ammunition/Fuel Interaction

### 5.3.4 Preventing a Constraint Violation

The backup strategy of separating fuel drums from ammunition boxes outlined above helps undo mistakes made by others, but the truck should be able to prevent itself from putting them together in the first place. The best way to prevent the truck from mixing fuel and ammunition is to prevent the lowest level arm-ungrasp RAP from putting one down in the presence of the other. This strategy is effective no matter how higher level behaviors are encoded because there is no way to get an item into a cargo bay without ungrasping it. There are many different ways to implement

this strategy and the two most general ones are described below.

### Using Constraint Clauses

The first strategy is the simplest and comes closest to the spirit of a strict plan-time constraint. The load RAPs used to put fuel drums and ammunition boxes into cargo bays can include explicit constraints to cause their subtasks to fail if a problematic item is already in the bay. The code to implement such a strategy is shown in Figure 5.11. Two RAPs are used, one to dispatch on the type of item to be loaded into the bay, and another to set up the appropriate constraint. When a task to load a fuel drum is executed, the first method in the load RAP is instantiated. This generates a task to load one fuel drum into the bay and constrains it to require that the cargo bay not hold an ammunition box already. When ammunition boxes are to be loaded, the second method produces a similar constraint against fuel drums. At any point in the execution of these subtasks, the constraint will cause a failure if fuel drums and ammunition boxes are about to be placed together.

This strategy is effective at preventing load tasks from placing fuel and ammunition together as long as there is enough routine sensing. However, there are many other RAPs that might also place a fuel drum or an ammunition box into a cargo bay. For instance, a RAP might need to make room in one cargo bay by moving a fuel drum into another. Encoding appropriate constraints into every RAP that might handle a fuel drum is tedious, complex, and makes the writing of new RAPs very difficult.

### Using Explicit Low-Level Sensing

Rather than try and control the assignment of fuel drums and ammunition boxes to cargo bays through the use of constraints, it is easier and more effective to add explicit checks into the lowest-level ungrasp RAPs. Since an inherited constraint is just an execution time check on a memory state, moving the check down to where it matters produces the same results without having to specify it in many different places. The most effective way to write a low level ungrasp RAP that includes such a check is shown in Figure 5.12. The RAP includes three methods: one to make sure there are no ammunition boxes where a fuel drum is to go, one to check there are no fuel drums where an ammunition box is to go, and one for ungrasping other things.

```
(DEFINE-RAP
   (INDEX (load ?type ?bay ?number))
   (RESOURCES (counter ?count ?number))
   (SUCCEED (<= (value ?count) 0))
   (REPEAT-WHILE (> (value ?count) 0))
   (METHOD
      (CONTEXT (= ?type fuel-drum))
      (TASK-NET
         (t1 (load-one-constrained fuel-drum ?bay ammunition-box)
             (COUNTER-SUB ?counter 1))))
   (METHOD
      (CONTEXT (= ?type ammunition-box))
      (TASK-NET
         (t1 (load-one-constrained ammunition-box ?bay fuel-drum)
             (COUNTER-SUB ?counter 1))))
   (METHOD
      (CONTEXT (and (not (= ?type fuel-drum))
                    (not (= ?type ammunition-box))))
      (TASK-NET
         (t1 (load-one ?type ?bay)
             (COUNTER-SUB ?counter 1)))))

(DEFINE-RAP
   (INDEX (load-one-constrained ?type ?bay ?constraint))
   (CONSTRAINTS (not (and (location ?thing ?bay)
                          (class ?thing ?constraint true))))
   (METHOD
      (TASK-NET
         (t1 (load-one ?type ?bay)))))
```

**Figure 5.11**: Enforcing a Constraint with Constraints

Notice the way that the first two methods are written in the ungrasp RAP. An explicit eye-scan is included to ensure that the ungrasp primitive step occurs in a situation free of problems. The eye-scan is done explicitly to ensure that ammunition or fuel have not appeared in the cargo bay since the last time it was scanned. The eye-scan takes time and might not be justified if some mistakes can be tolerated. In this case, however, all errors must be avoided so explicit sensing is indicated. Another subtlety in the coding of this RAP is the inclusion of the `know-sensor-name` check in the context of each method. Like the make-room-in-bay RAP discussed in Section 5.1.2, there is a potential for mix-up here if the item involved is referred to

```
(DEFINE-RAP
   (INDEX (arm-ungrasp-p ?arm ?thing))
   (SUCCEED (not (arm-holding ?arm ?thing)))
   (METHOD
      (CONTEXT (and (arm-place ?arm ?place)
                    (class ?thing fuel-drum true)
                    (KNOW-SENSOR-NAME ?thing)))
      (TASK-NET
         (t1 (eye-scan-p ?place)
             ((not (and (location ?item ?place)
                        (class ?item ammunition-box))) FOR t2))
         (t2 (real-arm-ungrasp-p ?arm ?thing))))
   (METHOD
      (CONTEXT (and (arm-place ?arm ?place)
                    (class ?thing ammunition-box true)
                    (KNOW-SENSOR-NAME ?thing)))
      (TASK-NET
         (t1 (eye-scan-p ?place)
             ((not (and (location ?item ?place)
                        (class ?item fuel-drum))) FOR t2))
         (t2 (real-arm-ungrasp-p ?arm ?thing))))
   (METHOD
      (CONTEXT (and (not (or (class ?thing ammunition-box true)
                             (class ?thing fuel-drum true)))
                    (KNOW-SENSOR-NAME ?thing)))
      (TASK-NET
         (t1 (real-arm-ungrasp-p ?arm ?thing)))))
```

**Figure 5.12**: Enforcing a Constraint with Explicit Sensing

via a long-term memory description. If `?thing` is bound to a description with an unknown or incorrect class, it will not appear to be either a fuel drum or ammunition box and the third method will put it down without further checking. However, in the delivery truck domain, whenever an item's sensor name is known, its class is known as well. An actual arm-ungrasp RAP has to be slightly more complex to force identification in those cases where the item's sensor name is not known.

A final comment on this strategy for preventing fuel drums and ammunition boxes from coming together is that it might not work. There will always be a time lag between gathering the sensor data necessary to assess the situation and issuing the primitive action requests to make things happen. During that time interval some

other agent can sneak in and mess things up.

### 5.3.5 Constraint Issues

While the RAP system can encode many types of constraint-based behavior, each specific constraint usually requires an idiosyncratic representation. There is no general procedure for translating a planning constraint into its active, execution time equivalent. Such strategies usually trade off resource expenditure against the cost of having the constraint violated. Different situations will demand different trade offs and, hence, different strategies. The issue is further complicated by the lack of look-ahead in the RAP system. When specific situations are to be prevented, all actions that might cause problems must anticipate the situations and try to avoid them. Thus, a planning system that wishes to put constraints on sketchy plans must realize that such constraints will often translate into complex active strategies.

## 5.4 Coordinating Reaction Tasks

A final consideration when constructing a RAP library is that the agent being controlled must not only be adaptive, but reactive as well. For example, a mail delivery robot might be considered adaptive as it travels down a hallway and moves around people, boxes and other obstructions. As it moves it must adapt its actions to cope with the changing situation it encounters along the way. However, if the robot encountered a fire and went around it as if it were simply another obstruction, it couldn't be said to be reacting appropriately. Discovering a fire should change the task the robot chooses to work on from delivering mail to sounding the alarm.

Reactive behavior implies the existence of concurrent monitor tasks on the RAP execution agenda. Each monitor task waits to react to some situation when it arises. To make the mail robot reactive, it must hold both a task to deliver the mail and a task to sound the alarm in case of fire. In general, there will be reactive tasks on the agenda for utility-based opportunities, for repairing constraint violations, and to monitor the currently accessible situation. All of these tasks are designed to interrupt tasks of lesser importance so they can react in a timely manner. To prevent confusion,

and produce reasonable behavior when several monitors wake up at the same time, the priority levels assigned to particular monitor tasks must be carefully considered.

Task priorities ultimately derive from a consideration of what will happen when a situation is not dealt with quickly. When ignoring a state change will lead to disaster, the task to deal with it should have a high priority so that it can quickly take control. The RAP system does not look into the future, so the assessment of the severity of ignoring various state changes must be done in advance. This results in the assignment of a static priority to each initial reaction task on the system's agenda. When the assessment of a task suggests significantly different priorities in different situations, it can be split into separate tasks with appropriate priorities for each situation. The RAP system has no facility for dynamically changing a task's priority because priority assessment must include looking into the future.

Properly coordinating a system's reactions is almost universally discussed in terms of concurrent tasks with different priorities. Schank and Abelson [1977] present a taxonomy of goals and subgoals in and suggest that some classes of goal give rise to higher priority tasks than others. Goals giving rise to preservation tasks will usually be of higher priority than goals of enjoyment for example. McCalla *et al.* [1982] suggests adding concurrent, high-priority control tasks to a route plan before execution so that appropriate reactions will be exhibited. Kaelbling [1988] also discusses priority assignment in the GAPP system and gives a notation for their specification. In Wilensky [1983] it is argued that priorities should not be static but dynamically determined and the PRS system [Georgeff *et al.*, 1986] uses explicit meta-planning information to actually do some dynamic assessment.

### 5.4.1 The Delivery Truck Domain

To illustrate some of the considerations that must go into structuring a RAP-based execution system, let us return to the delivery truck domain. In the experiments described in Chapter 6, three basic reaction tasks run concurrently with delivery tasks assigned to the truck. As shown in Figure 5.13, these tasks are: a situation monitoring task, a task to deal with enemy troops, and a task to prevent the truck from running out of fuel.[3] These tasks are assigned the priorities 8, 6, and 4 respectively, and regular

[3]These tasks are also discussed in Chapter 3

delivery tasks run at priority 2. A separation of 2 between each level is required.

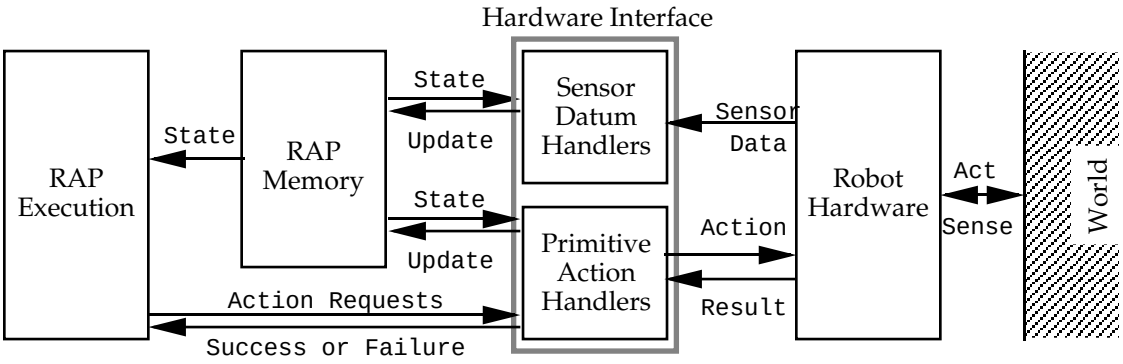



**Figure 5.13**: Delivery Domain Reaction Tasks

### Monitoring the Current Situation

The situation monitoring sensory task is assigned the highest priority because it is the most important. The purpose of the situation monitoring task is to produce an eye-scan primitive action request whenever the local situation has not been checked for 20 time units.[4] An eye-scan returns all of the accessible items surrounding the truck so periodically scanning ensures that the local situation description remains up-to-date in memory. The importance, and high priority, of this task stem from the fact that the enemy troops might arrive at any time. Should they arrive while the truck is busy on some task that does not require an eye-scan operation, their presence might remain undetected for an arbitrarily long time. The situation monitoring task prevents enemy troops from going undetected for more than 20 time units.

### Dealing with Enemy Troops

The presence of enemy troops signals an emergency situation that must be dealt with immediately to prevent the truck from being captured or disabled. The method used to deal with enemy troops is shown in Figure 5.14 and consists of two methods: one to fight and one to flee.[5] There are two things in particular to notice about this RAP.

[4]The code for this task is shown in Figure 5.7 and is discussed on page 159.

[5]The details of the methods in this RAP are discussed in more detail on page 126.

First, the task-net in the fight method describes a behavior that involves fighting for a while and then fleeing if the fight is going badly. To support that behavior, a run-away task is spawned with priority one higher than the shooting task. This added priority enables the run-away task to take control promptly when fighting time is up. However, the added priority puts the run-away task at a higher priority than this task (*i.e.*, 7 instead of 6) and care must be taken to avoid having it pre-empt situation monitoring. To keep situation monitoring from being interfered with, it is assigned a priority two higher than enemy troop handling.

```
(DEFINE-RAP
   (INDEX   (handle-enemy-troops))
   (SUCCEED (not true))
   (MONITOR-STATE (and (location ?enemy external)
                       (class ?enemy enemy-unit true)))
   (METHOD
      (CONTEXT (and (location ?thing weapon-bay)
                    (class ?thing weapon true)
                    (quantity-held ?thing ?amount)
                    (> ?amount 0)))
      (TASK-NET
         (t1 (shoot-enemy-troops ?thing)
             (UNTIL-END t2))
         (t2 1 (run-away-from-enemy)
             (UNTIL-END t1)
             (WINDOW NOW 18 25))))
   (METHOD
      (TASK-NET
         (t1 (run-away-from-enemy)))))
```

**Figure 5.14**: The RAP for Dealing with Enemy Troops

The second point to notice is that the enemy-troops task is a passive monitor. It relies on the situation monitoring task to gather information, and thus, the two work together to give the system appropriate reactions. As a passive monitor, however, the enemy-troops task will also respond to information gathered by other tasks. This gives the system an immediate response no matter how the truck finds out about enemy troops.

### Maintaining a Reasonable Fuel Level

The RAP descriptions for the fuel monitoring task is shown in Figures 5.9 and 5.16. The fuel monitoring task actually splits into two concurrent tasks, as shown in Figure 5.15, which work in concert to produce a complex behavior. The fix-low-fuel task waits on the task agenda until the fuel level gets below ten units. The fuel level does not need to be actively monitored because the truck fuel gauge is accessible without sensing and is updated by the hardware interface whenever a primitive action is executed. When the fuel level goes below ten units the task becomes eligible to run and will interrupt any delivery tasks and refuel the truck. There are three refueling methods and which is selected depends on the situation. If there are fuel drums outside the truck, they are always used first. If there are no fuel drums outside the truck, and there is fuel available for the trip, the truck goes in search of full fuel drums. Finally, if fuel is very short, the reserve stored in cargo bay one is used to give the truck a chance to find more.

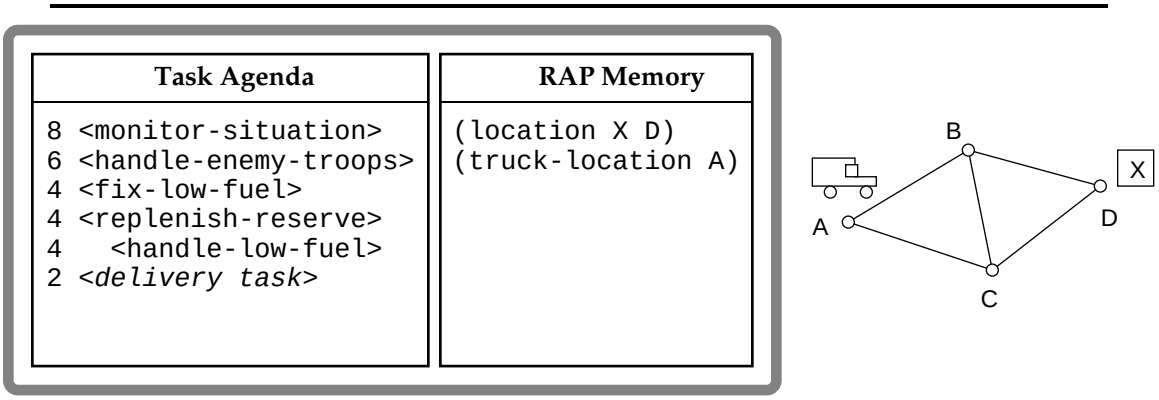



**Figure 5.15**: Two Fuel Monitoring Tasks

To support a fuel reserve, a utility-based opportunity monitor is used. The replenish-reserve task waits on the execution agenda until there is both a need for a new fuel reserve, and an accessible fuel drum. At that point, the task becomes active and puts the fuel drum into cargo bay one for use in an emergency situation. The task priority on the replenish-reserve task must be high, however, so that fuel drums are grabbed while they are accessible.

The replenish-reserve task works in conjunction with a physical memory strategy

```
(DEFINE-RAP
   (INDEX   (fix-low-fuel))
   (SUCCEED (not true))
   (MONITOR-STATE (and (truck-fuel ?level)
                       (< ?level 10)))
   (METHOD 1 ; use local fuel if possible
      (CONTEXT (and (location ?thing external)
                    (class ?thing fuel-drum true)
                    (quantity-held ?thing ?amount)
                    (or (= ?amount unknown)
                        (> ?amount 0))))
      (TASK-NET
         (t1 (fuel-up-locally))))
   (METHOD 2 ; if no local fuel try finding some
      (CONTEXT (and (truck-fuel ?level)
                    (> ?level 5)))
      (TASK-NET
         (t0 (fuel-up-elsewhere))))
   (METHOD 2 ; if fuel is dangerously low, use reserve
      (CONTEXT (and (truck-fuel ?level)
                    (<= ?level 5)))
      (TASK-NET
         (t0 (dump-in-fuel-reserves)))))
```

**Figure 5.16**: The Refueling RAP

shared by other tasks in the system.[6] The essence of the strategy is to reserve cargo bay for use by the truck alone, with all cargo being shipped using bay two. This separation prevents sensing and memory confusion from causing critical tools to be removed and left behind. A fuel reserve is no good if it might be removed from the truck by mistake. Rather than try and construct a complex series of monitors to make sure that tools and fuel do not leave the truck, they are placed only in cargo bay one. Loading and unloading RAPs then only need to stay away from cargo bay one to prevent errors. RAP representation and execution is often considerably better when the world can be used to avoid confusion in this way. However, such strategies affect the structure of all relevant RAPs in the library.

Finally, notice that the fuel monitoring tasks are placed two priority levels away

[6]Physical memory strategies are discussed in Section 4.8.2.

from the enemy troop tasks. Enemy-troop tasks are given priority because refueling takes time and enemy troops must be dealt with immediately. Two levels of separation are required because the subtasks that look for fuel drums set up plan-revision monitors to keep an opportunistic eye out for drums along the way (see Section 4.7.3). The separation between refueling and troop handling tasks can lead to problems when the truck is fleeing from enemy troops, however. Low fuel-levels will not be dealt with while fleeing because the fleeing tasks will have a higher priority. One cure to this problem would be to write the refueling and troop tasks so that each contains special methods to use when both are applicable. For example, the troop task might have a method to just fight if fuel is critically low, or to always flee towards location holding fuel.

### 5.4.2 Reaction Task Issues

There are three important points to consider when embedding an execution system in a domain:

- An appropriate set of reaction tasks
- A priority structure for the reaction tasks
- Physical strategies for improving RAP execution behavior

Reaction tasks are required to monitor critical aspects of the world, deal with emergency situations and take advantage of utility-based opportunities. A priority structure is needed to coordinate the reaction tasks so they interrupt each other appropriately. Finally, physical strategies can be used to simplify RAP code and task behavior in complex domains.

The decisions made in dealing with these three points have a major effect on the RAP representations in the library. Reaction tasks, priorities and physical strategies all work in concert with RAP methods to produce reasonable behavior and they must take each other into account. When a system does not react appropriately, it cannot be effective.

## 5.5 Summary of Representation Issues

In summary, there are four major issues that arise in various guises when creating task descriptions for a RAP based execution system. These issues are:

- Designing sensing strategies to keep the memory up-to-date
- The need for plan-revision opportunism
- Difficulties with maintaining or preventing situations in the world
- Defining and coordinating reaction tasks

Furthermore, these issues must be confronted by any execution system acting in a complex, dynamic world. As soon as a system cannot know everything about its future, these problems, which all involve tracking and reacting to changes in the world, become unavoidable.

RAPs must include sensing strategies to support three different memory processes:

- Filling in missing or uncertain information
- Item identification
- Situation monitoring

Information will routinely be missing or incomplete in the RAP memory and RAP methods must include sensing tasks to fill it in when it becomes relevant. Similarly, sensor limitations will sometimes leave the memory with multiple representations for the same item and RAPs will have to generate sensing tasks to gather the specific information needed to merge those representations together when necessary. The dynamic nature of the world also means that important features must be checked, not once, but repeatedly to ensure that changes will be noticed in a timely manner. Subtle interactions between these processes often lead to RAPs that are much more complex than introspection would suggest.

Plan-revision opportunities arise whenever method choice must be made on the basis of incomplete or incorrect information and new information that becomes available later suggests the use of a "better" method. An intelligent system must recognize

such situations and abandon its initial method in favor of the better one. The RAP system captures such behavior in a task-net idiom that instantiates the initial method concurrently with a higher priority task to watching for the opportunity. Plan-revision opportunities arise very commonly.

When moving from predictable planning domains to dynamic execution domains, plan protections and policies must become active strategies to enforce a constraint on the world. For example, the simple idea of protecting some state can be translated into any number of different strategies ranging from active defense against other agent tampering, to simply not changing it oneself. In fact, undesirable states will often be unavoidable and the best that can be done is to repair them when they occur. Planning constraints typically become combinations of both prevention and repair strategies.

An execution system that is embedded in its domain must also react appropriately to utility-based opportunities and emergency situations. In the RAP system, such reactions are generated through the use of concurrent tasks. When a reaction is indicated, the appropriate task interrupts the currently active task to deal with the situation. To coordinate interruptions and sort out interference between multiple reactions, priorities based on the domain and the RAP library must be carefully assessed and assigned to each task.

The next chapter presents the results of actually running the RAP system in the simulated delivery truck domain. The results show that intelligent behavior relies heavily on the decisions made in dealing with these four execution issues. In fact, it was in the process of struggling to construct a working RAP library for the delivery truck that many of the subtleties discussed above were first encountered.

# Chapter 6

# Examples and Experiments

The RAP situation-driven execution system has been described in some detail in the previous chapters and, at this point, it is time to show that an actual implementation of the RAP system will indeed cope effectively with the real world. Ideally, one would like to show proof that the RAP approach to task execution can solve a problem that other systems cannot. Or that the system can handle the same problems as other systems but with a more useful representation. Or, at the very least, that the system actually behaves as claimed.

Unfortunately, it is very difficult to prove of any of these things because it is unclear what form such proof would take. By definition, coping is "doing something reasonable in most situations". However, the terms "reasonable" and "in most situations" are too vague to give that definition any rigor. Without a rigorous definition to prove things about, evaluation of the success of a situation-driven execution system must lie in actual performance.

To study the RAP system's performance on realistic problems, it was implemented to control a simulated delivery-truck. The simulator used to represent the domain is described in Firby and Hanks [1987b], and the domain itself is described in Appendix B. The implementation of the RAP memory and interpreter follow the descriptions in Chapters 2, 3, and 4 and are domain-independent up to the ability of an assertional database to represent the world. The domain-dependent aspects of the system are all included in the hardware interface and the RAP library as both must be tailored to the characteristics of the truck. To keep things honest, the RAP system

and the simulator are completely isolated from one another except at the hardware interface. The system can only get information about the world by issuing primitive action requests to the truck and getting information back from its sensors. The simulator itself is not used by the execution system in any way; the truck is treated exactly as it would be if it were real.[1]

As evidence that the RAP system can cope effectively with the simulated delivery-truck domain, this chapter offers two types of example: detailed traces of truck behavior on small problems, and aggregate measures of truck behavior on large problems. Detailed examples take the form of annotated execution traces to show the way truck actions adapt to the complexities of the world. While such examples are very illustrative, they are useful in describing only short behaviors, because large scale tasks typically require thousands of primitive actions. To show the aggregate performance over large scale tasks, graphs summarizing specific aspects of system behavior are presented. The aspects summarized are chosen to highlight the ability of the overall system to carry out complex tasks in the face of a dynamic world.

The evidence presented shows that the RAP system does indeed behave as suggested in the preceding chapters. To the extent that the system behaves as suggested, and the behavior suggested covers a wide range of interesting problems, as argued in Chapters 4 and 5, the RAP approach to situation-driven execution has proven very successful.

## 6.1 The Delivery Truck Domain

Through the use of experiments in the delivery-truck domain, we would like to show that a RAP-based situation-driven execution system can indeed cope with a complex, dynamic world. As discussed in Section 1.4.2 and Appendix B, the delivery domain is a simulated world in which a truck is controlled by the execution system. The world consists a set of locations connected together by roads and the truck can move along the roads from location to location. At each location, the truck may encounter

[1] As mentioned in Chapter 1, there are those who would argue that useful statements about the way a system will "really" behave cannot be obtained without using real robot hardware in the real world. While we have much sympathy for such a position, we argue in [Firby and Hanks, 1987b] that reasonable progress can still be made if realistic simulators are used. Real hardware is also still difficult to get one's hands on.

any number of items that can be manipulated in various ways. The domain is more complex than the usual planning research "blocks world" domains because of the large number and varied types of item manipulation required. Furthermore, many aspects of the simulator operate in a probabilistic fashion. For example, the amount of fuel used in traversing a road, the chance of a primitive action failing for no reason, and the effectiveness of shooting at enemy troops are all vary probabilistically. The domain is also dynamic, containing other agents that move objects around and enemy troops which may appear at any time.

### 6.1.1 The Basic Domain

A map of the delivery-truck domain is shown in Figure 6.1. Each of the boxes in the figure represents a location and the lines connecting them represent roads. The three quarries on the right produce rocks of different colors and the three factories on the left consume them. The warehouse at the top of the map contains tools, weapons and ammunition and the depot at the bottom of the map contains fuel drums. The quarries, warehouse, and depot are active locations that periodically produce new rocks, ammunition, and fuel. Enemy troops can appear anywhere in the world except the warehouse and depot, and objects in the world move from place to place on their own to simulate the actions of unspecified, autonomous agents. To gain knowledge about the world, the truck must use sensing tasks to scan the items at its current location and examine each in detail.

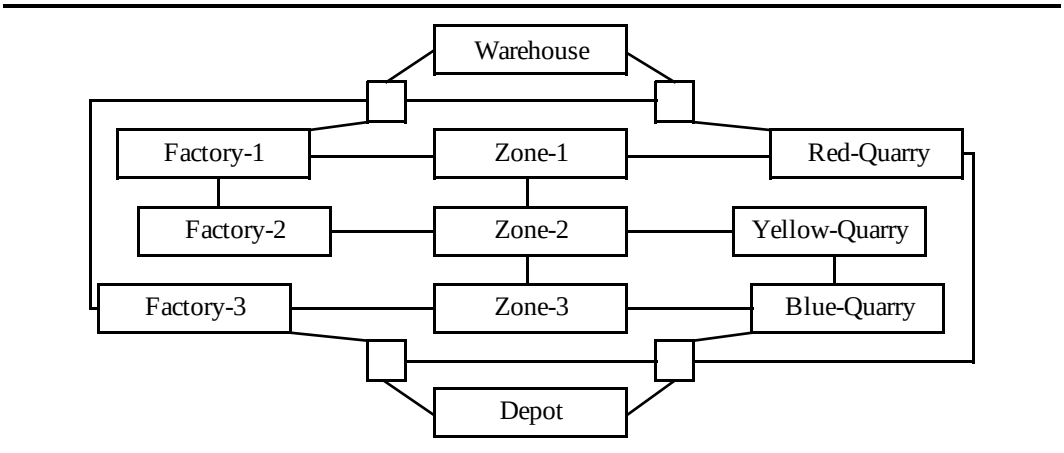



**Figure 6.1**: The Delivery Truck Domain

The key features that make this domain a good test of the RAP system are:

- Complete knowledge about objects in the world is impossible
- Other agents can dynamically change the current situation
- The truck's primitive actions can fail

Complete knowledge of the domain is impossible to attain because the truck's sensing abilities are limited to its current location. For example, when the truck is at the depot it can see all of the objects there, but it cannot find out anything about the rest of the world. Furthermore, in keeping with the spirit of the experiments, the truck does not start out with an initial picture of the world other than knowing the connectivity of the map. In fact, when the system is first connected to the truck in the world, the special RAP shown in Figure 6.2 is instantiated to look around and develop a picture of the initial accessible state.

```
(DEFINE-RAP
   (INDEX  (wakeup))
   (METHOD
      (TASK-NET
         (t1 (eye-scan-p external))
         (t2 (eye-scan-p bay1))
         (t3 (eye-scan-p bay2))
         (t4 (arm-move-to-folded arm1))
         (t5 (arm-move-to-folded arm2)))))
```

**Figure 6.2**: The RAP To Initially Wake the Truck Up

To simulate the actions of other agents, the simulator runs a demon at regular intervals to randomly shuffle objects between locations in the world. The frequency of this demon's shuffles, and the number of objects that get moved each time, are variables in the experiments. The shorter the interval between shuffles, the more often the truck has to scan its location to maintain an up-to-date picture of the world in memory. The more objects that get shuffled each time, the more out-of-date the memory will get between scans. Independently from the shuffling demon,

enemy troops can also appear throughout the world. After enemy troops arrive at the truck's location they will stay there until they are shot or the truck leaves. While enemy troops are present, there is an increasing probability that the truck will become captured. However, when the truck leaves a location, the enemy troops do not follow.

Along with dynamic agents in the form of the shuffling demon and enemy troops, the truck must also cope with primitive action failures. All primitive actions will fail when the world is not in an appropriate state. For example, a grasp will fail if the arm is not positioned at a graspable object and the truck will not move if it is not pointed toward a road. Similarly, a grasp may fail after the arm has been positioned properly because the shuffling demon has wakened and moved the object away. All primitive actions will also fail if the objects they involve are no longer accessible. In addition, there is a small probability that the grasp primitive will drop the object it is trying to pick up. Appendix B shows all primitive actions the truck can execute and the way each might fail.

### 6.1.2 Objects in the Domain

As implied by the wakeup RAP discussed above, the truck being controlled by the system has two arms and two cargo bays. The arms are identical except that `arm2` can grasp bigger items. The two cargo bays are also the same, but `bay1` is set aside for tools and a fuel reserve as discussed in Section 5.4.1, leaving only `bay2` for carrying cargo. The truck also has a fuel-bay with a capacity of 30 units of fuel. Fuel is consumed each time the truck traverses a road and it typically has to refuel two or three times during one of the large experiments discussed below. Since actual fuel consumption varies probabilistically around a nominal value, the precise consumption, and the number of refueling tasks required for an experimental run, cannot be determined in advance.

To support the truck in its travels, the domain contains fuel drums that can contain from 0-5 units of fuel. The depot is an active location that periodically removes empty fuel drums from the world and generates new full ones. When the truck needs fuel, it must find fuel drums and pour them into its fuel bay. Enemy troops may also appear as described above, so the world contains a weapon and ammunition that the truck can use to defend itself. The weapon can be moved on and off of the truck through

the use of a special gun tool. To shoot enemy troops, the weapon must contain ammunition and it must be toggled by a truck arm. Ammunition is produced at the warehouse in ammunition boxes that may hold from 0-15 rounds. The truck must pour ammunition boxes into the gun to keep its 15 round magazine loaded. Shooting consumes ammunition so occasional reloading is required.

The basic task assigned to the delivery truck is to move rocks from one place in the world to another. Rocks come in different colors and sizes and each quarry generates rocks of one color. Quarries only generate rocks periodically so they cannot be counted on to have rocks on hand all of the time. However, for the experiments, the period was tuned so the truck only had to wait occasionally. As rocks are generated at the quarries, the shuffling demon slowly disperses them throughout the world. To keep this process from over-populating the world with rocks (or fuel or ammunition), the shuffling demon will not place new items at a location that already holds ten or more. For the same reason, the quarries and other item producers do not generate new items if more than ten already exist at their location.

### 6.1.3 The Basic Delivery Task

For the examples in this chapter, the basic task is to delivery some number of rocks of a given color to one of the factories. The truck can use rocks from anywhere in the world to fill an order and an order may require the truck to make an arbitrary number of trips. While busy delivering rocks, the truck might run into enemy troops or run out of fuel. Therefore, the RAP system's task agenda starts out with the three basic reaction tasks and priorities discussed in Section 5.4.1: a fuel monitoring task, an enemy monitoring task, and a situation monitoring task. When new delivery tasks are given to the truck, they are added to the task agenda along with these reaction tasks. No real sketchy plan is necessary because each delivery task can be instantiated with a single RAP and executed concurrently with any others. To prevent accidentally completing a task for the truck, the shuffling demon does not appear at the factories.

The truck uses an order filling strategy with the following steps:

1. Go to a location believed to hold rocks of the correct color.

2. If no rocks are known, or if rocks are not found using step 1, go to the appropriate quarry.

3. Pick up enough rocks at the current location to fill the order or the truck.

4. Go to the correct factory.

5. Unload the rocks picked up in step 3.

6. If the order is not yet complete, go to step 1.

This strategy results in the truck trying to find rocks it knows about, and if it doesn't know of any, going to the appropriate quarry and waiting for some new rocks to be produced.

### 6.1.4 The RAP Library

To support the delivery truck in its domain, the RAP library holds 96 RAPs defining 195 separate methods. The most complex feature of the domain is controlling the truck's two arms. When moving an arm from one place to another there are many special cases that can arise depending on where it starts and where it is going. There are also many specific recovery strategies needed to deal with different types of arm failure. 33 RAPs in the library smooth out these complexities and define more generally applicable arm-move, arm-pickup, and arm-putdown operations. Tasks to move the truck from place to place are also reasonably complex and require 19 RAPs. Basic sensing tasks take up 10 RAPs that are referenced by almost every other RAP, 7 RAPs are required to generate appropriate refueling behavior, 9 RAPs define tasks for handling enemy troops, and one RAP is used to monitor the current situation. Building on this structure, only 6 RAPs are needed to define the tasks that implement the rock delivery strategy described above.

Only a few RAPs are required to actually implement the delivery tasks because most of the routine truck control tasks, and their associated failures, are independent of the high-level tasks that are built on them. The problems involved in moving things in and out of the truck are almost independent of the items themselves, just as they are in the real world. Specific strategies are necessary for items with different

shapes, sizes, and weights, but very seldom does anything else about an item matter. This ability of RAPs to "smooth the world out" by encapsulating the special purpose strategies needed to carry out abstract tasks is crucial if planning is to be possible. The idea of a sketchy plan only makes sense when useful high-level tasks can be defined for the planner to use as building blocks. The RAP system was proposed as a representation and interpreter for such tasks and it is significant that the library demonstrates the structure it does. Many RAPs are required to deal with the low-level details in the world, but only a few are required to define high-level tasks once the details are smoothed out.

## 6.2 Detailed Examples

One way to illustrate the effectiveness of the RAP system is through the use of detailed execution traces. This section includes two traces: a long one that shows the creation, execution and completion of all the RAPs required to execute a simple adaptive behavior, and a short one that shows the way priorities and monitors interact to generate useful reactions. The traces are intended to show the RAP system adapting to cope with uncertain information, primitive action failure and dynamic situation changes. Traces also give insight into the way the RAP interpreter functions.

The first trace follows the process of mounting a weapon in the truck's weapon-bay. At an abstract level, this task is very simple and consists of going to the weapon, picking it up, and putting it in the bay. In reality however, the trace is much more complicated. The truck first picks up a special tool needed to pick up the gun and, after doing that, finds out that the gun is too large for the arm being used. The truck then switches the tool from one arm to the other and, after the switch, it must cope with dropping the tool a couple of times. This sort of low-level manipulation of objects while information is gained (the size of the gun) and failures are dealt with (dropping things) is exactly what the RAP system is designed for.

The second trace follows the truck through an interactions with enemy troops. The truck is moving from a quarry to a factory and runs into enemy troops in one of the intermediate zones. The enemy-troop monitor immediately wakes up and the truck tries shooting at the troops. After a few shots, the enemy troops have not been

vanquished so the truck turns and runs away. This type of reaction is the basis of the RAP system's ability to cope with dynamic situations.

Both traces are actual output taken from experiments in the delivery-truck domain. The first trace includes all of the tasks required to instantiate the mount-weapon behavior while the second trace includes only the primitive action requests.

### 6.2.1 Basic Coping Behavior

The behavior traced in this example is that of mounting a weapon in the weapon-bay. The truck starts out at the fuel depot and is assigned the task of getting a weapon ready for use should enemy troops appear. The truck must move to the warehouse to find a gun and then lift it into the weapon-bay. A total of 25 primitive actions are generated in the course of picking the gun up. A mistake is made early on because the truck does not know how large the gun is and some juggling is required to sort things out. Almost all of the actions performed are in service of low-level arm movement and grasping tasks and correspond intuitively to the kinds of manipulation people do quite unconciously. One might pick up a wrench, discover that it won't work in your left hand, and switch it to your right without a second thought.

The trace included is complete, except where indicated, so that it also serves as a good example of the inner workings of the RAP interpreter. Detailed examples are difficult to include because the context is as important as the RAPs themselves in controlling the system's behavior. Therefore, in this example, the situation is kept very tame and I have tried to include all of the state description required to follow along.

#### Trace 1: Mounting a Weapon

The truck can only shoot at enemy troops when it is carrying a loaded weapon in its weapon bay. Therefore, before the reaction task for handling enemy troops can be added to the task agenda, the RAP shown in Figure 6.3 must be executed. This RAP describes a task that mounts a weapon in the weapon bay, and then loads it with ammunition.

```
(DEFINE-RAP
   (INDEX   (ready-weapon => ?weapon))
   (SUCCEED (and (location ?weapon weapon-bay)
                 (class ?weapon weapon true)
                 (quantity-held ?weapon ?amount)
                 (> ?amount 0)))
   (METHOD
      (TASK-NET
         (t1 (mount-weapon => ?weapon)
             ((location ?weapon weapon-bay) for t2))
         (t2 (load-weapon ?weapon)))))
```

**Figure 6.3**: Getting a Weapon Ready

We will start following the execution trace at the point when a ready-weapon task comes up for execution the first time. The truck is near the beginning of an experimental run, and the low-fuel and situation-monitor reaction tasks are already on the agenda. Lines starting with "-->" indicate that the interpreter has just selected the indicated RAP to execute and lines beginning with "..." show actions taken by the interpreter while executing that RAP. The first part of the trace is:

```
--> Considering task <READY-WEAPON => ?WEAPON> [8]
    ... Creating task: MOUNT-WEAPON            [9]
    ... Creating task: LOAD-WEAPON             [10]
    ... Suspending task: READY-WEAPON          [8]
```

This output covers one execution of the ready-weapon task. The numbers after each line are assigned when the task is created so that it can be easily distinguished throughout the trace (however, when a task is completed its number gets recycled). The method selected for the ready-weapon task generates two subtasks: mount-weapon and load-weapon. After placing these subtasks on the agenda, the RAP interpreter suspends the ready-weapon task by placing it back on the agenda and constraining it to wait for both the mount-weapon and load-weapon tasks to complete. Therefore, the agenda now holds tasks [9], [10] and [8], as well as any other tasks that might be running concurrently.

After executing the ready-weapon task, the interpreter must select a new task to execute and, since nothing more important is eligible to run, it selects the next task

```
(DEFINE-RAP
   (INDEX   (mount-weapon => ?weapon))
   (SUCCEED (and (location ?weapon weapon-bay)
                 (class ?weapon weapon true)))
   (METHOD - methods for other situations -)
   (METHOD
      (CONTEXT (not (truck-location warehouse)))
      (TASK-NET
         (t1 (truck-travel-to warehouse)
             ((truck-location warehouse) for t2)
             ((truck-location warehouse) for t3))
         (t2 (look-for-class weapon => ?thing)
             ((class ?thing weapon true) for t3) (for t4))
         (t3 (move-thing-to ?thing weapon-bay))
         (t4 (arm-examine ?thing)))))
```

**Figure 6.4**: RAP for Mounting a Weapon on the Truck

in the ready-weapon family. This is the mount-weapon task shown in Figure 6.4. The mount-weapon task contains several methods for use when different weapon locations are known but, in this example, the truck does not know about any weapons. When no weapon locations are known, the RAP directs the truck to go to the warehouse as a default. The trace continues below and shows that choice.

```
--> Considering task <MOUNT-WEAPON => ?WEAPON> [9]
    ... Creating task: TRUCK-TRAVEL-TO         [11]
    ... Creating task: LOOK-FOR-CLASS          [12]
    ... Creating task: MOVE-THING-TO           [13]
    ... Suspending task: MOUNT-WEAPON          [9]
```

The method chosen includes three subtasks. The truck-travel-to task moves the truck to the warehouse, the look-for-class task looks around there for a weapon, and the move-thing-to task moves the weapon found into the weapon-bay. After adding these three tasks to the agenda, the interpreter suspends the mount-weapon task and selects the truck-travel-to task to run next.

```
--> Considering task: <TRUCK-TRAVEL-TO WAREHOUSE>      [11]
     - trip edited out -
--> Considering task: <TRUCK-TRAVEL-TO WAREHOUSE>      [11]
    ... Task accomplished: <TRUCK-TRAVEL-TO WAREHOUSE> [11]
    ... Removing task: [11]
```

The truck-travel-to task generates a good many subtasks that move the truck from its current location to the warehouse. These subtasks have been taken out of the trace to make it shorter but when the last of them completes, the truck-travel-to task comes up for execution again. At this point, the truck is at the warehouse, its succeed clause is satisfied and it is removed from the task agenda. Removal of the travel task enables execution of the look-for-class task. The look-for-class task is a loop to check items that are currently accessible and see if any of them is a weapon. The items around the truck will have just been scanned during the last step in the truck-travel-to task. Therefore, all accessible items will be known along with their primary class. However, `weapon` is a secondary characteristic of objects and can only be determined by examining an item closely. The first iteration of the loop to do the examination produces the next trace fragment.

```
--> Considering task <LOOK-FOR-CLASS WEAPON => ?FOUND> [12]
    ... Creating task: EYE-EXAMINE                     [14]
    ... Suspending task: LOOK-FOR-CLASS                [12]
--> Considering task <EYE-EXAMINE ITEM-15> [14]
    ... Creating task: EYE-EXAMINE-P       [11]
    ... Suspending task: EYE-EXAMINE       [14]
--> Considering task <EYE-EXAMINE-P EXTERNAL ITEM-15> [11]
    ... Creating task: REAL-EYE-EXAMINE-P              [19]
    ... Suspending task: EYE-EXAMINE-P                 [11]
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL ITEM-15> [19]
    ... Primitive operation (EYE-EXAMINE EXTERNAL OBJ-7) - OKAY
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL ITEM-15> [19]
    ... Task accomplished: <REAL-EYE-EXAMINE-P EXTERNAL ITEM-15> [19]
    ... Removing task: [19]
    ... Task accomplished: <EYE-EXAMINE-P EXTERNAL ITEM-15>      [11]
    ... Removing task: [11]
    ... Task accomplished: <EYE-EXAMINE ITEM-15>                 [14]
    ... Removing task: [14]
```

The look-for-class task chooses an unexamined accessible item with internal name `item-15` and generates an eye-examine task to check it out. The eye-examine task considers the location of the item and generates an eye-examine-p with the location filled in. The eye-examine-p task would then force identification of the item if its sensor name were not known. However, since its sensor name is known, a real-eye-examine-p task is generated immediately and that task makes a primitive action request to the hardware interface. Notice that at the time of the primitive

action request, internal names are translated into sensor names (*i.e.*, `item-15` becomes `obj-7`).[2] The primitive action succeeds, and the hardware interface updates the memory with the data returned by the examination. In this instance, `item-15` is an ammunition box. With the completion of the primitive action, all of the eye-examine related tasks are also complete, and are removed from the agenda. Now, the look-for-class task comes up for execution again, and the same sequence of tasks is generated for item `tool-2`.

```
--> Considering task <LOOK-FOR-CLASS WAREHOUSE WEAPON => ?FOUND> [12]
    ... Creating task: EYE-EXAMINE                                  [14]
    ... Suspending task: LOOK-FOR-CLASS                             [12]
--> Considering task <EYE-EXAMINE TOOL-2> [14]
    ... Creating task: EYE-EXAMINE-P      [11]
    ... Suspending task: EYE-EXAMINE      [14]
--> Considering task <EYE-EXAMINE-P EXTERNAL TOOL-2> [11]
    ... Creating task: REAL-EYE-EXAMINE-P           [19]
    ... Suspending task: EYE-EXAMINE-P              [11]
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL TOOL-2> [19]
    ... Primitive operation (EYE-EXAMINE EXTERNAL TOOL-2) - OKAY
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL TOOL-2> [19]
    ... Task accomplished: <REAL-EYE-EXAMINE-P EXTERNAL TOOL-2> [19]
    ... Removing task: [19]
    ... Task accomplished: <EYE-EXAMINE-P EXTERNAL TOOL-2>      [11]
    ... Removing task: [11]
    ... Task accomplished: <EYE-EXAMINE TOOL-2>                 [14]
    ... Removing task: [14]
```

In this piece of trace the internal and sensor name for `tool-2` have been replaced with a single name for easier reading. The extra "`Considering task`" lines that come right before the "`Task accomplished`" lines have also been removed.

```
--> Considering task <LOOK-FOR-CLASS WAREHOUSE WEAPON => ?FOUND> [12]
    ... Creating task: EYE-EXAMINE                                  [14]
    ... Suspending task: LOOK-FOR-CLASS                             [12]
--> Considering task <EYE-EXAMINE GUN-1> [14]
    ... Creating task: EYE-EXAMINE-P     [11]
    ... Suspending task: EYE-EXAMINE     [14]
--> Considering task <EYE-EXAMINE-P EXTERNAL GUN-1> [11]
    ... Creating task: REAL-EYE-EXAMINE-P           [19]
    ... Suspending task: EYE-EXAMINE-P              [11]
```

[2]This translation can only occur if the sensor name has already been determined with a previous sensing operation. In this case, the eye-scan done by the truck-travel-to task upon arrival at the warehouse generated the sensor names.

```
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL GUN-1> [19]
    ... Primitive operation (EYE-EXAMINE EXTERNAL GUN-1) - OKAY
--> Considering task <REAL-EYE-EXAMINE-P EXTERNAL GUN-1> [19]
    ... Task accomplished: <REAL-EYE-EXAMINE-P EXTERNAL GUN-1> [19]
    ... Removing task: [19]
    ... Task accomplished: <EYE-EXAMINE-P EXTERNAL GUN-1>      [11]
    ... Removing task: [11]
    ... Task accomplished: <EYE-EXAMINE GUN-1>                 [14]
    ... Removing task: [14]
```

After examining the tool, the look-for-class task loops again and examines `gun-1` The eye-examine primitive action request returns data saying that this item is a weapon and the hardware interface makes a note to that effect in memory. At that point, the succeed clause in the look-for-class task becomes satisfied as `gun-1` is an accessible item known to be a weapon. Notice that the output variable for the look-for-class task is bound after the succeed clause is checked.

```
--> Considering task <LOOK-FOR-CLASS WEAPON => ?FOUND>      [12]
    ... Task accomplished: <LOOK-FOR-CLASS WEAPON => GUN-1> [12]
    ... Removing task: [12]
--> Considering task <MOVE-THING-TO GUN-1 WEAPON-BAY> [13]
    ... Creating task: PICKUP-HERE                     [12]
    ... Creating task: PUTDOWN-AT                      [14]
    ... Suspending task: MOVE-THING-TO                 [13]
```

Immediately after the look-for-class task completes, the move-thing-to task comes up for execution and begins the process of moving `gun-1` into the weapon-bay. It spawns two subtasks: the first to pick the gun up and the second to put it down in the weapon-bay. The RAPs for the pickup task are shown in Figure 6.5. The pickup-here RAP, which describes the first task generated to move the gun, uses the more general pickup RAP, but first constrains all of its subtasks to occur at the current location. This stops the pickup from continuing if it is interrupted and the truck moves away before the gun can be picked up. The putdown-at does not require such a constraint because the gun is being put down inside the truck (which moves when the truck moves).

The pickup RAP must choose which arm to use to pick the gun up. Currently, the only criterion used to make this choice is whether or not the arm is known to be too small. On the truck used in the experiments, `arm1` is not as strong as `arm2` and when

```
(DEFINE-RAP
   (INDEX   (pickup-here ?here ?thing))
   (CONSTRAINTS (truck-location ?here))
   (SUCCEED (arm-holding ?some-arm ?thing))
   (METHOD
      (TASK-NET
         (t1 (pickup ?thing)))))

(DEFINE-RAP
   (INDEX   (pickup ?thing))
   (PRECONDITIONS (or (not (too-small-for arm1 ?thing))
                      (not (too-small-for arm2 ?thing))))
   (SUCCEED (arm-holding ?some-arm ?thing))
   (METHOD
      (CONTEXT (not (too-small-for arm1 ?thing)))
      (TASK-NET
         (t1 (arm-empty  arm1)
             ((not (arm-holding arm1 ?anything)) for t2))
         (t2 (arm-pickup arm1 ?thing))))
   (METHOD
      (CONTEXT (not (too-small-for arm2 ?thing)))
      (TASK-NET
         (t1 (arm-empty  arm2)
             ((not (arm-holding arm2 ?anything)) for t2))
         (t2 (arm-pickup arm2 ?thing)))))
```

**Figure 6.5**: RAPs for Picking Things Up

items are known to be too big for `arm1`, `arm2` will always be chosen. However, when the item is not known to be too large for an arm, both methods will be applicable and an arm will be selected at random. Thus, the trace continues and `arm1` is chosen.

```
--> Considering task <PICKUP-HERE WAREHOUSE GUN-1> [12]
    ... Creating task: PICKUP                      [11]
    ... Suspending task: PICKUP-HERE               [12]
--> Considering task <PICKUP GUN-1> [11]
    ... Creating task: ARM-EMPTY    [19]
    ... Creating task: ARM-PICKUP   [18]
    ... Suspending task: PICKUP     [11]
--> Considering task <ARM-EMPTY ARM1> [19]
    ... Task accomplished: <ARM-EMPTY ARM1> [19]
    ... Removing task: [19]
```

The arm-empty task generated by the pickup task is used to get rid of any extra items the arm might be holding. In this case there aren't any and the arm-empty task is trivially satisfied. Next, the arm-pickup task is chosen by the interpreter.

```
--> Considering task <ARM-PICKUP ARM1 GUN-1> [18]
    ... Creating task: ARM-PICKUP            [19]
    ... Creating task: ARM-MOVE-TO           [15]
    ... Creating task: ARM-GRASP-THING       [23]
    ... Suspending task: ARM-PICKUP          [18]
--> Considering task <ARM-PICKUP ARM1 TOOL-2> [19]
    ... Creating task: ARM-MOVE-TO            [22]
    ... Creating task: ARM-GRASP-THING        [26]
    ... Suspending task: ARM-PICKUP           [19]
```

The arm-pickup task generates three subtasks: first pick up a necessary tool, then move the arm to the gun, and finally grasp the gun. The arm-pickup RAP is written with special methods for items that require tools, and items of class weapon are known to require the use of a tool like `tool-2`. Therefore, to pickup `gun-1`, `arm1` must first be holding `tool-2`. The arm does not need a tool for picking up `tool-2`, however, so only two tasks are generated for it: one to position the arm, and one to grasp the tool.

The arm is currently folded, so executing the arm-move-to task to reach `tool-2` requires two steps. First the arm must be moved outside the truck, and then it must be moved to be at `tool-2`. Executing the first step produces the following trace. The arm-move-to-thing and arm-move-to-external tasks generate only single step plans, but they contain many different methods for different arm motion scenarios.

```
--> Considering task <ARM-MOVE-TO ARM1 TOOL-2> [22]
    ... Creating task: ARM-MOVE-TO-THING       [29]
    ... Suspending task: ARM-MOVE-TO           [22]
--> Considering task <ARM-MOVE-TO-THING ARM1 TOOL-2> [29]
    ... Creating task: ARM-MOVE-TO                  [24]
    ... Suspending task: ARM-MOVE-TO-THING          [29]
--> Considering task <ARM-MOVE-TO ARM1 EXTERNAL> [24]
    ... Creating task: ARM-MOVE-TO-EXTERNAL      [20]
    ... Suspending task: ARM-MOVE-TO             [24]
--> Considering task <ARM-MOVE-TO-EXTERNAL ARM1> [20]
    ... Creating task: ARM-MOVE-P                [21]
    ... Suspending task: ARM-MOVE-TO-EXTERNAL    [20]
--> Considering task <ARM-MOVE-P ARM1 EXTERNAL> [21]
    ... Creating task: REAL-ARM-MOVE-P          [30]
```

```
    ... Suspending task: ARM-MOVE-P             [21]
--> Considering task <REAL-ARM-MOVE-P ARM1 EXTERNAL> [30]
    ... Primitive operation (ARM-MOVE ARM1 EXTERNAL) - OKAY
--> Considering task <REAL-ARM-MOVE-P ARM1 EXTERNAL>       [30]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM1 EXTERNAL> [30]
    ... Task accomplished: <ARM-MOVE-P ARM1 EXTERNAL>      [21]
    ... Task accomplished: <ARM-MOVE-TO-EXTERNAL ARM1>     [20]
    ... Task accomplished: <ARM-MOVE-TO ARM1 EXTERNAL>     [24]
```

From now on, the "Removing task" lines will be edited out of the trace when they come after a "Task accomplished" line. Accomplished tasks are always removed from the agenda. The next step is to move the arm to tool-2.

```
--> Considering task <ARM-MOVE-TO-THING ARM1 TOOL-2> [29]
    ... Creating task: ARM-MOVE-P                   [24]
    ... Suspending task: ARM-MOVE-TO-THING          [29]
--> Considering task <ARM-MOVE-P ARM1 TOOL-2> [24]
    ... Creating task: REAL-ARM-MOVE-P        [20]
    ... Suspending task: ARM-MOVE-P           [24]
--> Considering task <REAL-ARM-MOVE-P ARM1 TOOL-2> [20]
    ... Primitive operation (ARM-MOVE ARM1 TOOL-2) - OKAY
--> Considering task <REAL-ARM-MOVE-P ARM1 TOOL-2>         [20]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM1 TOOL-2>   [20]
    ... Task accomplished: <ARM-MOVE-P ARM1 TOOL-2>        [24]
    ... Task accomplished: <ARM-MOVE-TO-THING ARM1 TOOL-2> [29]
    ... Task accomplished: <ARM-MOVE-TO ARM1 TOOL-2>       [22]
```

Finally, the arm-grasp-thing task executes and picks the tool up. Again, during these steps the arm-grasp-p would have had to force item identification if the sensor-name for tool-2 wasn't already known.

```
--> Considering task <ARM-GRASP-THING ARM1 TOOL-2> [26]
    ... Creating task: ARM-GRASP-P                  [22]
    ... Suspending task: ARM-GRASP-THING            [26]
--> Considering task <ARM-GRASP-P ARM1 TOOL-2> [22]
    ... Creating task: REAL-ARM-GRASP-P        [20]
    ... Suspending task: ARM-GRASP-P           [22]
--> Considering task <REAL-ARM-GRASP-P ARM1 TOOL-2> [20]
    ... Primitive operation (ARM-GRASP ARM1 TOOL-2) - OKAY
--> Considering task <REAL-ARM-GRASP-P ARM1 TOOL-2> [20]
    ... Task accomplished: <REAL-ARM-GRASP-P ARM1 TOOL-2> [20]
    ... Task accomplished: <ARM-GRASP-P ARM1 TOOL-2>      [22]
    ... Task accomplished: <ARM-GRASP-THING ARM1 TOOL-2>  [26]
    ... Task accomplished: <ARM-PICKUP ARM1 TOOL-2>       [19]
```

The arm-pickup `tool-2` task spawned by the arm-pickup `gun-1` task is now complete and the RAP interpreter goes on to execute the arm-move-to task to move `arm1` so that is can pickup the gun. This time, moving the arm only takes one step because the arm is already outside of the truck.

```
--> Considering task <ARM-MOVE-TO ARM1 GUN-1> [15]
    ... Creating task: ARM-MOVE-TO-THING      [26]
    ... Suspending task: ARM-MOVE-TO          [15]
--> Considering task <ARM-MOVE-TO-THING ARM1 GUN-1> [26]
    ... Creating task: ARM-MOVE-P =>                [19]
    ... Suspending task: ARM-MOVE-TO-THING          [26]
--> Considering task <ARM-MOVE-P ARM1 GUN-1> [19]
    ... Creating task: REAL-ARM-MOVE-P       [22]
    ... Suspending task: ARM-MOVE-P          [19]
--> Considering task <REAL-ARM-MOVE-P ARM1 GUN-1> [22]
    ... Primitive operation (ARM-MOVE ARM1 GUN-1) - OKAY
--> New rock order ORDER-106: 4 RED rocks for FACTORY-1
    ... Creating task: DELIVER-ROCKS [20]
--> Considering task <REAL-ARM-MOVE-P ARM1 GUN-1>         [22]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM1 GUN-1>   [22]
    ... Task accomplished: <ARM-MOVE-P ARM1 GUN-1>        [19]
    ... Task accomplished: <ARM-MOVE-TO-THING ARM1 GUN-1> [26]
    ... Task accomplished: <ARM-MOVE-TO ARM1 GUN-1>       [15]
```

A rock order has slipped in here. Rock orders are generated at intervals during the experiments and one happened to appear here. Rock orders have a lower priority than getting the truck's weapons ready, though, so the interpreter continues on picking up the gun. However, a snag quickly develops because `arm1` is too small to hold both the tool and the gun.

```
--> Considering task <ARM-GRASP-THING ARM1 GUN-1> [23]
    ... Creating task: ARM-GRASP-P                [15]
    ... Suspending task: ARM-GRASP-THING          [23]
--> Considering task <ARM-GRASP-P ARM1 GUN-1> [15]
    ... Creating task: REAL-ARM-GRASP-P       [26]
    ... Suspending task: ARM-GRASP-P          [15]
--> Considering task <REAL-ARM-GRASP-P ARM1 GUN-1> [26]
    ... Primitive operation (ARM-GRASP ARM1 GUN-1) - ARM-TOO-FULL
--> Considering task <REAL-ARM-GRASP-P ARM1 GUN-1> [26]
    ... Goal failed (INTERFERENCE) - ARM-TOO-FULL: [26]
    ... Removing task: [26]
--> Considering task <ARM-GRASP-P ARM1 GUN-1>      [15]
    ... Goal failed (INTERFERENCE) - ARM-TOO-FULL: [15]
    ... Removing task: [15]
```

The primitive action request to pick up the gun fails because the arm is too full. After the failure, the real-arm-grasp-p task executes again and it would try the primitive action again. However, the real-arm-grasp-p contains a precondition saying that it is only valid if the last method tried did not result in an ARM-TOO-FULL failure. Thus, it fails immediately and the arm-grasp-p task starts up. The arm-grasp-p task is structured the same way and also fails immediately. This use of failure-based preconditions at a very low-level keeps the implicit looping behavior of the RAP interpreter from trying primitive actions over and over when it is clear there is no point. When the arm is just too small, it makes no sense to try the grasp again. The arm-grasp-thing task does not contain such a precondition, however, because there are strategies for picking up some items that fix the problem at this level. Unfortunately, the gun is not such an item so the arm-grasp-thing task tries the same method over again.

```
--> Considering task <ARM-GRASP-THING ARM1 GUN-1> [23]
    ... Creating task: ARM-GRASP-P                [15]
    ... Suspending task: ARM-GRASP-THING          [23]
--> Considering task <ARM-GRASP-P ARM1 GUN-1> [15]
    ... Creating task: REAL-ARM-GRASP-P       [26]
    ... Suspending task: ARM-GRASP-P          [15]
--> Considering task <REAL-ARM-GRASP-P ARM1 GUN-1> [26]
    ... Primitive operation (ARM-GRASP ARM1 GUN-1) - ARM-TOO-FULL
--> Considering task <REAL-ARM-GRASP-P ARM1 GUN-1> [26]
    ... Goal failed (INTERFERENCE) - ARM-TOO-FULL: [26]
    ... Removing task: [26]
--> Considering task <ARM-GRASP-P ARM1 GUN-1>      [15]
    ... Goal failed (INTERFERENCE) - ARM-TOO-FULL: [15]
    ... Removing task: [15]
--> Considering task <ARM-GRASP-THING ARM1 GUN-1>    [23]
    ... Goal failed (INFINITE-LOOP) - ARM-TOO-FULL : [23]
    ... Removing task: [23]
--> Considering task <ARM-PICKUP ARM1 GUN-1>       [18]
    ... Goal failed (INTERFERENCE) - ARM-TOO-FULL: [18]
    ... Removing task: [18]
```

Retrying the same method, results in the same set of failures, and the futile-loop detector prevents the arm-grasp-thing task from trying a third time. The resulting failure causes the first arm-pickup task executed on the gun to come up for execution again. This task immediately fails for a rather obscure reason. Looking back at the RAP code for the pickup task in Figure 6.5 will show that the task-net chosen

includes the arm-empty task to make sure that arm1 is not holding anything when the arm-pickup begins. The first time the arm-pickup is executed, the arm is not holding anything but, this time, the arm is holding tool-2. The arm-pickup fails immediately because the implied precondition generated by the FOR clause in the task-net is not satisfied in this case. The final result is that the original pickup task comes up for execution once again.

```
--> Considering task <PICKUP GUN-1> [11]
    ... Creating task: ARM-EMPTY    [18]
    ... Creating task: ARM-PICKUP   [23]
    ... Suspending task: PICKUP     [11]
--> Considering task <ARM-EMPTY ARM2>       [18]
    ... Task accomplished: <ARM-EMPTY ARM2> [18]
    ... Removing task:                      [18]
```

Notice that this time the pickup task has chosen to use arm2 rather than arm1 because the method using arm2 has not yet been tried and the method using arm1 has failed. In simpler circumstances, the system might learn to never try arm1 on this item again. The arm-grasp action handler in the hardware interface (see Figure 3.14) will assert that the arm being used to grasp an item is too small if it fails with ARM-TOO-FULL and the arm is empty. The pickup RAP will then never consider the method using that arm again. When RAPs are coded appropriately, this type of low-level world structure can be learned quite naturally.[3] However, in this case, arm1 cannot pick the gun up but it is also holding tool-2. Thus, the action handler does not assert that arm1 is too small and arm1 may be tried again in the future.

The next task executed is to get arm2 to pick up the tool needed for the gun. This task is more complex than before, because the tool is already being held by arm1. The complexity creeps in at the subtask to move the arm to the tool.

```
--> Considering task <ARM-PICKUP ARM2 GUN-1> [23]
    ... Creating task: ARM-PICKUP            [18]
    ... Creating task: ARM-MOVE-TO           [15]
    ... Creating task: ARM-GRASP-THING       [26]
    ... Suspending task: ARM-PICKUP          [23]
--> Considering task <ARM-PICKUP ARM2 TOOL-2> [18]
```

[3] In no sense is this a general type of learning. The RAPs must be coded in advance to alter their behavior if the information is available — all that has to be learned is the information. A more interesting learning theory would cover the construction of new RAPs and RAP methods.

```
    ... Creating task: ARM-MOVE-TO            [19]
    ... Creating task: ARM-GRASP-THING        [22]
    ... Suspending task: ARM-PICKUP           [18]
--> Considering task <ARM-MOVE-TO ARM2 TOOL-2> [19]
    ... Creating task: ARM-MOVE-TO-THING       [21]
    ... Suspending task: ARM-MOVE-TO           [19]
--> Considering task <ARM-MOVE-TO-THING ARM2 TOOL-2> [21]
    ... Creating task: ARM-PUTDOWN                    [24]
    ... Creating task: ARM-MOVE-TO                    [29]
    ... Suspending task: ARM-MOVE-TO-THING            [21]
```

The arm-move-to-thing task has chosen a quite different method this time than when the tool was outside the truck. When the tool was outside, arm-move-to-thing just generated a task to move the arm directly to the tool. Now, it has generated two tasks: one to put the tool down first and then one to move the arm. This putdown task is required because the truck cannot switch items from one arm to another; it must put the item down first. The arm-putdown task uses `bay1` for the exchange since it has been set aside for such uses. Thus, the first step in moving `arm2` becomes moving `arm1` to `bay1`. Notice that the arm-move-to tasks have chosen different methods for moving the arm than they did when moving it outside.

```
--> Considering task <ARM-PUTDOWN ARM1 TOOL-2> [24]
    ... Creating task: ARM-PUTDOWN-AT          [30]
    ... Suspending task: ARM-PUTDOWN           [24]
--> Considering task <ARM-PUTDOWN-AT ARM1 TOOL-2 BAY1> [30]
    ... Creating task: ARM-MOVE-TO                    [31]
    ... Creating task: ARM-UNGRASP-P                  [32]
    ... Suspending task: ARM-PUTDOWN-AT               [30]
--> Considering task <ARM-MOVE-TO ARM1 BAY1> [31]
    ... Creating task: ARM-MOVE-TO-BASIC     [34]
    ... Suspending task: ARM-MOVE-TO         [31]
--> Considering task <ARM-MOVE-TO-BASIC ARM1 BAY1> [34]
    ... Creating task: ARM-MOVE-P                   [33]
    ... Suspending task: ARM-MOVE-TO-BASIC          [34]
--> Considering task <ARM-MOVE-P ARM1 BAY1> [33]
    ... Creating task: REAL-ARM-MOVE-P      [35]
    ... Suspending task: ARM-MOVE-P         [33]
--> Considering task <REAL-ARM-MOVE-P ARM1 BAY1> [35]
    ... Primitive operation (ARM-MOVE ARM1 BAY1) - OKAY
--> Considering task <REAL-ARM-MOVE-P ARM1 BAY1>          [35]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM1 BAY1>    [35]
    ... Task accomplished: <ARM-MOVE-P ARM1 BAY1>         [33]
    ... Task accomplished: <ARM-MOVE-TO-BASIC ARM1 BAY1> [34]
    ... Task accomplished: <ARM-MOVE-TO ARM1 BAY1>        [31]
```

The next step is to ungrasp the tool and deposit it in bay1.

```
--> Considering task <ARM-UNGRASP-P ARM1 TOOL-2> [32]
    ... Creating task: REAL-ARM-UNGRASP-P        [33]
    ... Suspending task: ARM-UNGRASP-P           [32]
--> Considering task <REAL-ARM-UNGRASP-P ARM1 TOOL-2> [33]
    ... Primitive operation (ARM-UNGRASP ARM1 TOOL-2) - OKAY
--> Considering task <REAL-ARM-UNGRASP-P ARM1 TOOL-2> [33]
    ... Task accomplished: <REAL-ARM-UNGRASP-P ARM1 TOOL-2>  [33]
    ... Task accomplished: <ARM-UNGRASP-P ARM1 TOOL-2>       [32]
    ... Task accomplished: <ARM-PUTDOWN-AT ARM1 TOOL-2 BAY1> [30]
    ... Task accomplished: <ARM-PUTDOWN ARM1 TOOL-2>         [24]
```

Now that the tool has been put down, arm2 must be positioned to pick it up. Moving the arm requires first moving it into the bay and then to the tool. The trace for the first step shows only the highlights of this because it is just like the arm movements above.

```
--> Considering task <ARM-MOVE-TO ARM2 TOOL-2> [29]
    ... Creating task: ARM-MOVE-TO-THING       [30]
    ... Suspending task: ARM-MOVE-TO           [29]
--> Considering task <ARM-MOVE-TO-THING ARM2 TOOL-2> [30]
    ... Creating task: ARM-MOVE-TO                   [24]
    ... Suspending task: ARM-MOVE-TO-THING           [30]
--> Considering task <ARM-MOVE-TO ARM2 BAY1>  [24]
    - lines missing -
    ... Primitive operation (ARM-MOVE ARM2 BAY1) - OKAY
    ... Task accomplished: <ARM-MOVE-TO ARM2 BAY1> [24]
```

After moving the arm into the bay, it must be moved to the tool.

```
--> Considering task <ARM-MOVE-TO-THING ARM2 TOOL-2> [30]
    ... Creating task: ARM-MOVE-P                    [24]
    ... Suspending task: ARM-MOVE-TO-THING           [30]
--> Considering task <ARM-MOVE-P ARM2 TOOL-2> [24]
    ... Creating task: REAL-ARM-MOVE-P        [33]
    ... Suspending task: ARM-MOVE-P           [24]
--> Considering task <REAL-ARM-MOVE-P ARM2 TOOL-2> [33]
    ... Primitive operation (ARM-MOVE ARM2 TOOL-2) - OKAY
```

At this point, the arm movement task is almost complete, but the situation monitor task wakes up. The situation monitor is one of the reaction tasks always present on the task agenda and it becomes eligible to run whenever the current location has

not been scanned recently. Since the task of mounting the weapon has not required a scan since the truck arrived, the situation monitor wakes up to do one. The situation monitor runs immediately because it has a high priority.

```
--> Switching family <MONITOR-SITUATION>   [4]
    ... Creating task: EYE-SCAN-P          [32]
    ... Suspending task: MONITOR-SITUATION [4]
--> Considering task <EYE-SCAN-P EXTERNAL> [32]
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
--> Considering task <EYE-SCAN-P EXTERNAL>       [32]
    ... Task accomplished: <EYE-SCAN-P EXTERNAL> [32]
```

The situation monitor is now no longer eligible to run because the location has been recently scanned (*i.e.*, by the eye-scan just performed) so the RAP interpreter looks for a new task to run. Since the last family being executed before the interruption included the weapon mounting tasks, the interpreter returns to them. All of the mounting tasks are waiting for the arm motion to complete so we continue just where we left off.

```
--> Switching family <REAL-ARM-MOVE-P ARM2 TOOL-2>          [33]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM2 TOOL-2>    [33]
    ... Task accomplished: <ARM-MOVE-P ARM2 TOOL-2>         [24]
    ... Task accomplished: <ARM-MOVE-TO-THING ARM2 TOOL-2> [30]
    ... Task accomplished: <ARM-MOVE-TO ARM2 TOOL-2>        [29]
    ... Task accomplished: <ARM-MOVE-TO-THING ARM2 TOOL-2> [21]
    ... Task accomplished: <ARM-MOVE-TO ARM2 TOOL-2>        [19]
```

Arm2 is now positioned at tool-2 and must pick it up. A snag develops, however, because the domain physics will not allow an object to be grasped if both arms are positioned at it. Arm2 has just been moved to the tool and arm1 remains there from when it was put down. An ARM-INTERFERENCE failure is reported by the hardware and the interface makes a note in memory that the arms are interfering. As with the ARM-TOO-FULL failure above, the low-level tasks know to not repeat on an ARM-INTERFERENCE failure.

```
--> Considering task <ARM-GRASP-THING ARM2 TOOL-2> [22]
    ... Creating task: ARM-GRASP-P                 [19]
    ... Suspending task: ARM-GRASP-THING           [22]
--> Considering task <ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Creating task: REAL-ARM-GRASP-P        [30]
```

```
    ... Suspending task: ARM-GRASP-P           [19]
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [30]
    ... Primitive operation (ARM-GRASP ARM2 TOOL-2) - ARM-INTERFERENCE
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2>    [30]
    ... Goal failed (INTERFERENCE) - ARM-INTERFERENCE: [30]
    ... Removing task: [30]
--> Considering task <ARM-GRASP-P ARM2 TOOL-2>         [19]
    ... Goal failed (INTERFERENCE) - ARM-INTERFERENCE: [19]
    ... Removing task: [19]
```

At this point, the arm-grasp-thing task comes up for execution again, and chooses a new method based on the assertion that the arms are interfering. This method starts by moving arm1 away from the tool.

```
--> Considering task <ARM-GRASP-THING ARM2 TOOL-2> [22]
    ... Creating task: ARM-MOVE-AWAY               [19]
    ... Creating task: ARM-GRASP-P                 [30]
    ... Suspending task: ARM-GRASP-THING           [22]
--> Considering task <ARM-MOVE-AWAY ARM1 TOOL-2> [19]
    ... Creating task: ARM-MOVE-P                 [29]
    ... Suspending task: ARM-MOVE-AWAY            [19]
--> Considering task <ARM-MOVE-P ARM1 FOLDED> [29]
    ... Creating task: REAL-ARM-MOVE-P         [21]
    ... Suspending task: ARM-MOVE-P            [29]
--> Considering task <REAL-ARM-MOVE-P ARM1 FOLDED> [21]
    ... Primitive operation (ARM-MOVE ARM1 FOLDED) - OKAY
--> Considering task <REAL-ARM-MOVE-P ARM1 FOLDED> [21]
    ... Task accomplished: <REAL-ARM-MOVE-P ARM1 FOLDED> [21]
    ... Task accomplished: <ARM-MOVE-P ARM1 FOLDED>      [29]
    ... Task accomplished: <ARM-MOVE-AWAY ARM1 TOOL-2>   [19]
```

Arm2 is now free to try and pick up the tool. This time, however, the clumsy nature of the arm comes into play and it drops the tool twice before succeeding. The low-level tasks do allow retries on an ARM-DROPPED failure because an effective repair is simply to try again, but the futile loop detector stops the real-arm-grasp-p task after two tries.

```
--> Considering task <ARM-GRASP-P ARM2 TOOL-2> [30]
    ... Creating task: REAL-ARM-GRASP-P        [19]
    ... Suspending task: ARM-GRASP-P           [30]
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Primitive operation (ARM-GRASP ARM2 TOOL-2) - ARM-DROPPED
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
```

```
    ... Primitive operation (ARM-GRASP ARM2 TOOL-2) - ARM-DROPPED
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Goal failed (FUTILE-LOOP) - ARM-DROPPED:    [19]
    ... Removing task: [19]
--> Considering task <ARM-GRASP-P ARM2 TOOL-2> [30]
    ... Creating task: REAL-ARM-GRASP-P        [19]
    ... Suspending task: ARM-GRASP-P           [30]
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Primitive operation (ARM-GRASP ARM2 TOOL-2) - OKAY
--> Considering task <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Task accomplished: <REAL-ARM-GRASP-P ARM2 TOOL-2> [19]
    ... Task accomplished: <ARM-GRASP-P ARM2 TOOL-2>      [30]
    ... Task accomplished: <ARM-GRASP-THING ARM2 TOOL-2>  [22]
    ... Task accomplished: <ARM-PICKUP ARM2 TOOL-2>       [18]
```

At last, `arm2` is holding the tool. The next step in mounting the weapon is for `arm2` to pick the gun up. That requires moving to the gun outside the truck as before.

```
--> Considering task <ARM-MOVE-TO ARM2 GUN-1> [15]
    - lines missing -
    ... Primitive operation (ARM-MOVE ARM2 EXTERNAL) - OKAY
    - lines missing -
    ... Primitive operation (ARM-MOVE ARM2 GUN-1) - OKAY
```

After moving to the gun, the arm must pick it up. Part way through the grasp, the situation monitor interrupts once again.

```
--> Considering task <ARM-GRASP-THING ARM2 GUN-1> [26]
    ... Creating task: ARM-GRASP-P                [15]
    ... Suspending task: ARM-GRASP-THING          [26]
--> Considering task <ARM-GRASP-P ARM2 GUN-1> [15]
    ... Creating task: REAL-ARM-GRASP-P       [22]
    ... Suspending task: ARM-GRASP-P          [15]
--> Considering task <REAL-ARM-GRASP-P ARM2 GUN-1> [22]
    ... Primitive operation (ARM-GRASP ARM2 GUN-1) - OKAY
--> Switching family <MONITOR-SITUATION>   [4]
    ... Creating task: EYE-SCAN-P          [18]
    ... Suspending task: MONITOR-SITUATION [4]
--> Considering task <EYE-SCAN-P EXTERNAL> [18]
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
--> Considering task <EYE-SCAN-P EXTERNAL>       [18]
    ... Task accomplished: <EYE-SCAN-P EXTERNAL> [18]
--> Switching family <REAL-ARM-GRASP-P ARM2 GUN-1>       [22]
    ... Task accomplished: <REAL-ARM-GRASP-P ARM2 GUN-1> [22]
    ... Task accomplished: <ARM-GRASP-P ARM2 GUN-1>      [15]
```

```
    ... Task accomplished: <ARM-GRASP-THING ARM2 GUN-1>  [26]
    ... Task accomplished: <ARM-PICKUP ARM2 GUN-1>       [23]
    ... Task accomplished: <PICKUP GUN-1>                [11]
    ... Task accomplished: <PICKUP-HERE WAREHOUSE GUN-1> [12]
```

The task of picking up the gun has now been completed and it can be moved into the weapon-bay and ungrasped. These steps proceed uneventfully and the mount-weapon task is finally finished.

```
--> Considering task <PUTDOWN-AT GUN-1 WEAPON-BAY> [14]
    ... Creating task: ARM-PUTDOWN-AT               [12]
    ... Suspending task: PUTDOWN-AT                 [14]
    - lines missing -
    ... Primitive operation (ARM-MOVE ARM2 WEAPON-BAY) - OKAY
    - lines missing -
    ... Primitive operation (ARM-UNGRASP ARM2 GUN-1) - OKAY
    - lines missing -
    ... Task accomplished: <ARM-PUTDOWN-AT ARM2 GUN-1 WEAPON-BAY> [12]
    ... Task accomplished: <PUTDOWN-AT GUN-1 WEAPON-BAY>          [14]
    ... Task accomplished: <MOVE-THING-TO GUN-1 WEAPON-BAY>       [13]
    ... Task accomplished: <MOUNT-WEAPON => GUN-1>                [9]
```

### Summary of Trace

Those who actually made it through the above trace will see that there is an enormous amount of detail in simply moving the gun from outside the truck into the weapon-bay. The trace runs on for pages even though only 25 actual primitive actions are executed. The reason for the length of the trace is the number of intermediate RAPs that are required to make little decisions. Items must be selected for examination, an arm must be chosen, strategies must be selected for moving the arm in different situations, and on and on. These details represent exactly the complexity that the RAP system is designed to hide from a planner and the real world is even worse. Not only is there a lot to worry about, but the decisions don't matter very much.

One might argue that a planner would have elected to use `arm2` rather than than `arm1` right from the beginning because the size of the gun was not known. By choosing `arm2` the steps involved in juggling the tool between arms could have been avoided in our example. However, the fact that `arm2` is clumsy and drops things more often, suggests that `arm1` should be used whenever possible. Either way, the possibility of

error would be explicitly introduced: `arm1` might entail a switch, and `arm2` might drop things. As soon as the possibility of error is introduced, some form of adaptive behavior is required to sort things out at run-time. The example shows that the RAP system can handle both errors due to lack of information (choosing `arm1` when it is too small) and errors due to primitive actions failures (dropping things).

### 6.2.2 Reacting to a Situation

The behavior highlighted in the next trace is that of encountering enemy troops on the way to a factory. Notice that the enemy troops are not detected until a sensing operation alerts the system to their presence. At that point, the reaction task for handling enemy troops takes control, fights for awhile, and then runs away. The important point is that the system can drop what it is doing, deal with an emergency situation, and then return to where it left off without problem.

#### Trace 2: Encountering Enemy Troops

This execution trace is much shorter than the last one and everything has been cut out but the primitive action requests. The truck is on its way to factory-2 and along the way it runs into enemy troops. The truck first tries to fight the troops but after a couple of futile shots, it runs away. The RAP describing the strategies for handling enemy troops has been shown before in Figure 5.14.

The trip to the factory starts with the truck moving from place to place using travel RAPs like the one shown in Figure 5.3. Each movement requires pointing the truck toward the road to be travelled down, setting the truck speed, and then moving forward. If all goes well, the truck emerges at the end of the road moves on to the next leg of its trip. We pick up the truck as it moves down a road.

```
--> Considering goal <TRUCK-TRAVEL-TO FACTORY-2> [7]
    ... Primitive operation (TRUCK-TURN W) - OKAY
    ... Primitive operation (TRUCK-SPEED MEDIUM) - OKAY
    ... Primitive operation (TRUCK-MOVE) - OKAY
```

At this point, the truck has changed location, and as discussed previously, all movement tasks include a scan operation to see what is accessible at a new destination.

In this case, the truck has arrived at a location that holds six enemy troop units and the scan reveals there presence. The new assertions in memory about the presence of enemy troops wakes the enemy-troops reaction task up. The enemy task has a higher priority than the travel task so it interrupts and begins to execute.

```
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
--> Switching family <HANDLE-ENEMY-TROOPS> [6]
```

The method chosen by the handle-enemy-troops task involves fighting for a while and then running if things aren't working out. The RAP describing the fighting task is shown in Figure 6.6. It consists of a loop to keep shooting at the enemy until they are gone. As a precondition, the specified weapon must be loaded. The two methods in this RAP are essentially the same, except that the second includes an additional step to move an arm to the weapon if necessary. An item in the world cannot be toggled unless the truck has an arm positioned at it. Each method includes two sensing tasks as well. An arm-examine task is like an eye-examine task but is slightly faster. It is used to check on the the amount of ammunition left in the weapon. The eye-scan task is used to see whether there are still enemy troops left outside the truck. These are excellent examples of the routine sensing required to support RAP execution. The succeed clause for the shoot-enemy-troops RAP requires knowledge about accessible enemy troops and the preconditions clause requires information about the ammunition in the gun. The RAP itself must generate the sensing tasks to gather the information.

The method chosen by the handle-enemy-troops RAP instantiates both a shoot-enemy-troops task and a run-away-from-enemy task that run concurrently. However, the run-away task is constrained to not execute until 18 time units have passed. This delay gives the shoot task a chance to try its hand at getting rid of the enemy troops (see Section 4.5.2). As a result, the shoot-enemy-troops chooses its first method because `arm2` is already located at the gun and fires.

```
    ... Primitive operation (ARM-TOGGLE ARM2 GUN-1) - OKAY
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
    ... Primitive operation (ARM-EXAMINE ARM2 GUN-1) - OKAY
```

After all of these subtasks are complete, the shoot task comes up for execution again and finds that there are still four enemy troops outside. The gun can shoot as

```
(DEFINE-RAP
   (INDEX  (shoot-enemy-troops ?weapon))
   (SUCCEED (not (and (location ?enemy external)
                      (class ?enemy enemy-unit true))))
   (PRECONDITIONS (and (quantity-held ?weapon ?amount)
                       (> ?amount 0))))
   (REPEAT-WHILE (and (location ?enemy external)
                      (class ?enemy enemy-unit true)))
   (METHOD
      (CONTEXT (arm-at ?arm ?weapon))
      (TASK-NET
         (t1 (arm-toggle-p ?arm ?weapon) (for t2) (for t3))
         (t2 (arm-examine ?weapon))
         (t3 (eye-scan-p external))))
   (METHOD
      (CONTEXT (not (arm-at ?arm ?weapon)))
      (TASK-NET
         (t1 (arm-move-to-thing ARM1 ?weapon)
             ((arm-at ?arm ?weapon) for t2))
         (t2 (arm-toggle-p ?arm ?weapon) (for t3) (for t4))
         (t3 (arm-examine ?weapon))
         (t4 (eye-scan-p external)))))
```

**Figure 6.6**: The RAP for Shooting Enemy Troops

many as three each time it is toggled and this time two enemy were driven off. Since troops are still around, the shoot task loops again and chooses the same method. This time, after the shooting, there are still three enemy units left and the eye-scan primitive picks them up. At the same time, the 18 time units have passed and the run-away-from-enemy task wakes up. This task was generated concurrently with the shoot task and at a priority one higher. Thus, it interrupts the shoot task before the arm-examine can be executed and takes control.

```
    ... Primitive operation (ARM-TOGGLE ARM2 GUN-1) - OKAY
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
    ... Primitive operation (TRUCK-SPEED MEDIUM) - OKAY
    ... Primitive operation (TRUCK-MOVE) - OKAY
    ... Primitive operation (EYE-SCAN EXTERNAL) - OKAY
--> Switching family <TRUCK-TRAVEL-TO FACTORY-2> [7]
```

The run-away task immediately picks a direction, straight ahead in this case, and moves the truck. On reaching a new location, the movement tasks generate an eye-scan and, this time, no enemy troops are found. The eye-scan ends the movement task and the run-away comes up for execution again. Since there are no enemy troops outside the truck, the run-away task finishes. The end of the run-away task causes the end of the shoot-enemy-troops task as well, and the handle-enemy-troops task goes back to sleep. At that point, the RAP interpreter looks around for a new task to execute and goes back to the travel task that was interrupted.

### 6.2.3 Discussion of Detailed Examples

To summarize, the two detailed examples described above show that the RAP system can deal effectively with incomplete information, primitive action failure, and dynamic situations. The system was able to cope with the need to switch hands once it was discovered that the gun was too heavy for `arm1` and also had no trouble changing its behavior in the face of the `ARM-INTERFERENCE` and `ARM-DROPPED` failures. The enemy troops example also shows a typical case of the system interrupting one task to deal with a more important one. The examples are somewhat incomplete as they don't cover every behavior necessary in the delivery-truck domain, but I believe they are evidence for the validity of the RAP approach none-the-less.

## 6.3 Large Experiments

Execution traces are excellent for illustrating both the inner workings of a program and the way it handles specific interesting problems. However, such examples often leave lingering doubts about how well a system can cope with much larger problems. In traditional planning research, traces regularly show the construction of ten or twenty step plans to solve some interesting puzzle, but the question of whether the planner can build thousand-step plans always remains unanswered.[4] Planners often employ algorithms with computational properties that are very hard to scale up so this lack of results on large problems is not that surprising.

[4]The DEVISOR [Vere, 1985] and ISIS [Fox, 1986] planning systems are exceptions in this regard. Both authors show experimental results to give an idea of how well their program will do when constructing hundred-step schedules.

Like planning systems, execution systems also face difficulties when problems are scaled up in size. In particular, execution systems run the risk of "losing touch with the world" as time goes on. It is impossible to keep track of everything in the world and, over time, discrepancies between the real world and system's world model may accumulate to the point where the system is incapable of effective action. Similarly, action descriptions cannot account for everything that might go wrong and mistakes will be made. If these mistakes are not corrected, they will accumulate to a point where the system no longer knows what is going on. To head off such questions about the RAP execution system, this section presents a set of experiments with the simulator that involve the generation of thousands of primitive actions.

The experiments described are designed to show that the RAP system behaves in the same way over long problems as it does over short problems. Evidence for the system's ability to cope with incomplete knowledge, primitive failures and interruptions comes most readily from examples of RAP code and from short execution traces that can be studied in some detail. Therefore, if it can be shown that large scale performance is as expected given the smaller examples, the merits of the system can be argued on the small examples alone. The experiments below supply evidence for this claim. It is important to stress that the results of the experiments are only suggestive, however. It is very difficult to design experiments that can be analyzed in such a way as to prove anything about coping behavior. Here, the idea is to show that things work as expected in a general sense, not to make claims about specific performance on a particular problem.

### 6.3.1 The Basic Experiment

All three sets of results discussed below are variations on the same basic experiment. Nine independent delivery tasks are assigned to the truck and the amount of simulated time taken to carry them all out is recorded. Each experiment uses the same nine delivery tasks but varies the shuffling demon interval or effectiveness in some way. When the shuffling demon only appears occasionally, the world will stay constant for long periods of time but, when the shuffling demon appears often, the world around the truck will be changing constantly. During the experiment, enemy troops were restricted to the zones between the quarries and the factories during the experiments

and encounters were kept rare. The reason for this restriction was to keep results from being dominated by interactions between the truck and troops.

Each experimental run begins with the world in the same state and with the truck at the fuel depot. The following tasks are then performed to get things set up starting at time zero:

1. The wakeup task is executed to see what state the world is in.
2. The truck fuels up at priority 4.
3. The fuel-level monitoring task is instantiated at priority 4.
4. The truck situation monitoring task is instantiated at priority 8.
5. The truck moves to the warehouse where the weapon is mounted and loaded.
6. The enemy-troops monitoring task is instantiated at priority 6.

This process takes approximately 200 time units. Starting at time 120, a new delivery task at priority 2 is generated every 60 time units until a total of nine have been introduced. The nine tasks include three tasks to delivery 4 red rocks to factory-1, three to deliver 4 blue rocks to factory-2, and three to delivery 4 yellow rocks to factory-3. The tasks are interleaved by color. Thus, by the time the setup tasks are complete, two delivery tasks will have been introduced and by about time 600 all nine delivery tasks will be on the agenda. When the last delivery task is complete, the experiment stops.

### The Form of the Experimental Data

An example of the experimental data is shown in Figure 6.7. The reports show the results of running the same experiment twice. The shuffling demon was set to wake up every 100 time units with a 40% effectiveness. Each report shows the time every delivery order was introduced and the time it was completed (*i.e.*, the time the last rock was consumed by the correct factory). The first trial satisfied all nine orders but the second trial ended, most unusually, when the truck ran out of gas. The number

of effector and sensor operations is also tallied along with the total running time in simulator time units.[5]

```
ROCK ORDER REPORT - Trial 1       |  ROCK ORDER REPORT - Trial 2
 Number of orders completed: 9    |   Number of orders completed: 4
 Number of orders pending:   0    |   Number of orders pending:   5
 Completed orders:                |   Completed orders:
  ORDER-109: 4 YELLOW - 544, 761  |    ORDER-302: 4 YELLOW - 543, 773
  ORDER-106: 4 YELLOW - 363,1157  |    ORDER-299: 4 YELLOW - 361,1147
  ORDER-110: 4 BLUE   - 600,2049  |    ORDER-303: 4 BLUE   - 600,1808
  ORDER-108: 4 RED    - 491,2564  |    ORDER-301: 4 RED    - 480,2657
  ORDER-107: 4 BLUE   - 428,2936  |   Pending orders:
  ORDER-105: 4 RED    - 301,3589  |    ORDER-300: 4 BLUE   - 429, -
  ORDER-103: 4 YELLOW - 180,4243  |    ORDER-298: 4 RED    - 303, -
  ORDER-104: 4 BLUE   - 242,4489  |    ORDER-297: 4 BLUE   - 244, -
  ORDER-102: 4 RED    - 140,5323  |    ORDER-296: 4 YELLOW - 180, -
 Primitive Operations:            |    ORDER-295: 4 RED    - 145, -
  405 sensor, 772 effector        |   Primitive Operations:
  29 failed                       |    291 sensor, 664 effector
Running time: 0 -> 5323           |     97 failed.
                                  |  Running time: 0 -> 3930
```

**Figure 6.7**: Some Experimental Data

There are several things to observe about these results. First, notice that the orders in the two runs do not start at precisely the same times. This discrepancy arises because of the way time is stepped forward in the simulator. Each time that a primitive action is executed, some simulated time passes before the simulator returns. The amount of time that passes depends on the action executed and, when traversing roads, it is probabilistic. Orders are only generated after simulator updates so the actual time they appear depends somewhat on the actions generated beforehand.

The second thing to notice is that orders are not completed in the order they are generated. When the initialization tasks or a previous order complete, the RAP interpreter looks for a new task to execute and it will select a pending order at random. Also, sometimes an order task will take the truck to a quarry and rocks will not be

[5]For those who are interested, an experimental run of 5000 to 6000 simulated time units takes around six hours on an Apollo DN3000. That time includes many LISP (*i.e.*, NISP over T) garbage collections and execution time for both the RAP interpreter and the delivery truck simulator. A primitive action completes approximately every minute, on average.

available immediately. In such a situation, the task waits on the agenda for time to pass and the interpreter will select a different task to work on in the meantime. Both of these processes make the order of delivery task execution non-deterministic. If desired, a task ordering could be enforced by either ordering the tasks explicitly using a sketchy plan, or by placing a deadline on the task to influence the interpreter's choice.

Another interesting aspect of the results is the variable interval between task finishing times. One would expect the finish times to be separated by equal intervals because the tasks all involve approximately the same number of primitive actions. Minor variations will occur due to variability in the world and chance encounters with enemy troops, but the actual intervals range from 250 to 900 time units — a large variation. The main cause of such variation is the need to refuel. Every now and then the truck runs low on fuel and must go find more. Finding several fuel drums and dumping them into the fuel bay takes a significant amount of time, and that time that time gets included in the next delivery to complete. Refueling is required two or three times during a typical experimental run.

The last observation is that all nine tasks are not always completed. A trial may finish prematurely for two reasons. First, some disaster may befall the truck. During these experiments, the truck may run out of fuel or be captured by enemy troops. Running out of fuel happens very rarely, but on occasion, the refueling RAPs and the shuffling demon interact particularly badly. Consider what might happen when the truck is far from the fuel depot and runs low on fuel. When fuel level is very low, the truck will use its reserve to buy time and sets off looking for full fuel-drums. If the truck gets someplace where it expects to find fuel and the shuffling demon has moved it all away, the truck may run out of fuel on the way to the next place. Usually, this problem does not occur because the shuffling demon also adds fuel drums to new places. As long as any fuel drum is encountered while searching for fuel, it will be opportunistically used to keep the truck alive. Only when the expected fuel source is empty and no fuel is encountered anywhere along the way do difficulties arise. Captures are also rare because encounters with enemy troops are kept infrequent and the RAP strategies for reacting when they do appear are reasonably effective.

Second, a trial may finish prematurely if the truck is forced to give up on some orders. This happens when a delivery task is tried repeatedly and it is unsuccessful

each time. Eventually the futile-loop detector will fail the task, and since it has no parent, the interpreter simply gives up and moves it off the task agenda. When all methods for delivering rocks have been tried twice without getting a single rock of the correct color into the truck, a delivery task fails this way. Such failures only occur when the world gets sufficiently random that the truck cannot keep track of which rocks are which color. The experimental results include data from incomplete delivery tasks, but not from trials that involved truck disasters.

### Analysis of the Data

The primary experimental results presented below take the form of graphs plotting average time required to complete an order against various features of the shuffling demon. The average delivery completion time for a nine task trial was calculated by dividing the total time taken by the number of tasks completed in the trial. The results from several trials were then averaged to get each data point. A vertical range bar also shows the spread in the average trial times used to calculate a point. This is not a very rigorous data analysis methodology but the graph shapes are reproducible and it is the shape of each graph, not actual completion times, that we are concerned with.

The controlled variables in the experiments are the frequency with which the shuffling demon appears in the world, and the efficiency with which it shuffles. The shuffling interval controls the frequency at which shuffling takes place, and the shuffling efficiency controls the likelihood that a given item will get moved during a shuffle. For instance, when the shuffling interval is 20 time units and the shuffling efficiency is 50%, items will be shuffled every 20 units and about half of the items accessible to the truck will be moved each time. Items from other locations may also appear at the truck's location during each shuffle. If the shuffling efficiency is 100% then every item will be moved and if it is 0% then no item will ever move. By changing these variables, the world can be made more or less predictable.

The RAP library, and the delivery tasks assigned to the system, were kept constant over the experiments. Thus, the average amount of time required to complete a delivery task for a given set of shuffling parameters represents a measure of how well the library and system cope with a given amount of uncertainty. One would expect

that the system should be able to cope well with a certain amount of randomness because of built-in robustness. However, as the world gets more random, the system will reach a point where its built-in robustness is not enough to correct errors as they occur, and the RAP system will "lose touch with the world". At that point, the truck will become incapable of delivering rocks effectively and that is exactly the behavior observed.

The RAP system gets robustness from two different sources. First, there is the natural robustness of its situation-driven behavior. As soon as errors in memory are discovered, they are corrected and the system changes its behavior to deal with the new situation. For example, suppose the the truck reaches for a red rock and discovers that it isn't there because the shuffling demon has moved it. The memory will immediately be updated to show the rock is gone and the system will choose a new rock to pick up instead. This type of coping behavior is the heart of the situation-driven execution concept.

Dealing with execution errors usually only helps to remove information from memory that is no longer true. The second source of the RAP system's robustness comes from the sensing tasks it uses to gather new information. Items that get shuffled away can be detected and removed from memory by trying to manipulate them, but new items can only be identified and added to memory by looking around. The situation monitor makes sure new items are not overlooked by waking up every 15 time units and scanning the current location.

### 6.3.2 Experiment One

The first experiment holds the shuffling efficiency fixed at 40% and varies the shuffling interval from 100 time units down to 8 time units. The resulting average delivery times are shown in the graph in Figure 6.8. The black diamond on the graph shows the interval at which the situation monitor task scans the world. As can be seen, shuffling items around in the world has little effect on the performance of the system until it occurs more often than every 30 time units. At that point, the system begins to take longer to satisfy each order and, when the shuffling interval reaches 15 time units, things start to fall apart. At 15 time units the system often fails to fill one or two orders because it gives up on them, and when the shuffling interval gets down to 8

time units the system only manages to fill 4 or 5 deliveries during each run.

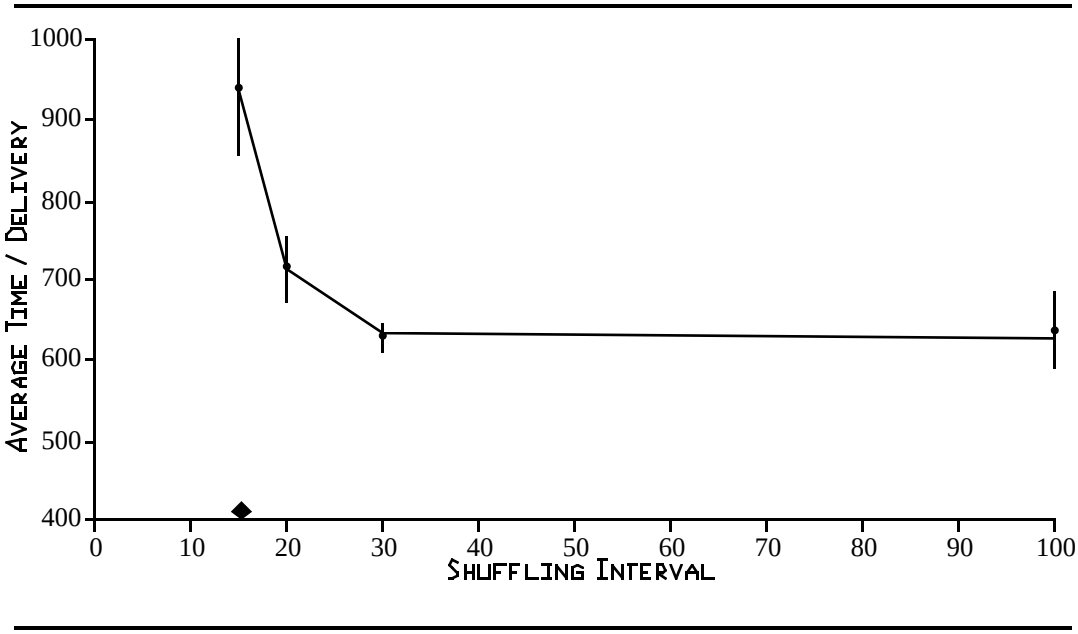



**Figure 6.8**: Results with Different Shuffling Intervals

A shuffling efficiency of 40% means that 4 out of 10 items that the truck thinks are accessible will actually be gone after each shuffle. With a situation monitoring interval of 15 time units, however, the truck will remain confused for at most 15 time units. This means that errors in the truck's model of the world at a given location will not persist for longer than 15 time units before being fully repaired. Thus, one would expect that when the shuffling interval is much longer than 15 time units, the world will be static most of the time and all errors will be easily smoothed out. Infrequent shuffling should have little effect on system performance, and it doesn't.

Each shuffle of the world, however, does generate additional primitive actions and as the frequency of the shuffling goes up, these actions account for more and more of the system's time. By additional actions, I mean actions that do not get the system any closer to satisfying its delivery goals. During the interval after a shuffle, but before a monitor scan, 4 out of 10 primitive action requests referring to items in the world will end in failure. These failed actions will eventually have to be repeated without failing so they add time to the overall experimental run. Similarly, the addition of new items to the current location causes extra sensing to occur. In particular, when a new rock is discovered, the system will have to generate sensing tasks to see if it is the correct color for a given delivery task. When the world remains static, this

| Experiment | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Shuffle Interval | 8 | 15 | 20 | 30 | 100 |
| Tasks Completed | 5 | 9 | 9 | 9 | 9 |
| Time Taken | 7293 | 7300 | 7069 | 5819 | 5323 |
| Effector Actions | 626 | 960 | 940 | 815 | 772 |
| Sensor Actions | 474 | 592 | 568 | 463 | 405 |
| Failed Actions | 127 | 85 | 67 | 45 | 29 |
| Percent Sensor | 43% | 38% | 38% | 36% | 34% |
| Percent Failed | 12% | 5% | 4% | 4% | 2% |

**Table 6.1**: A Series of Experiments with Various Shuffling Intervals

operation only has to be done once per item used for a task. However, when new items keep appearing and disappearing examinations will often be done on items that do not get used before they move away. As the shuffling interval gets shorter, these additional actions begin to increase the time taken per order. This increase appears in the graph as the sharp rise in delivery time between 30 and 15 time units.

Down to a shuffling interval of 15 time units, the system continues to complete all of the delivery tasks. It continues to be successful because only 4 out of 10 item descriptions get confused at a time so there is a better than even chance that a given action will work. When the shuffling interval goes below 15 time units however, damage to the memory no longer always gets repaired before the next shuffle occurs (*i.e.*, two shuffles can occur before a monitor scan). Thus, memory errors begin to rise and eventually actions are more likely to fail than succeed. When this happens, the system can no longer deal effectively with the world and delivery tasks fail to complete at an ever increasing rate.

Further insight may be gained by examining Table 6.1. The table contains a summary of the data from single runs of this experiment at different shuffling intervals. The increase of sensor actions in proportion to effector actions can be clearly seen as the shuffling interval decreases. Similarly, the number of failed actions increases as the shuffling interval goes down. By the time the interval reaches 8 time units, the system spends an inordinate amount of its time checking on rocks, reaching for them, and finding them gone. The system fails to achieve many delivery goals as a result.

### 6.3.3 Experiment Two

The second experiment is similar to the first but it varies the shuffling efficiency from 0% to 100% while holding the interval constant at 15 time units. The resulting curve is shown in Figure 6.9. The graph shows that as the shuffling efficiency goes up, the time required to complete an order also goes up until, at an efficiency just over 40%, the system begins to give up on orders.

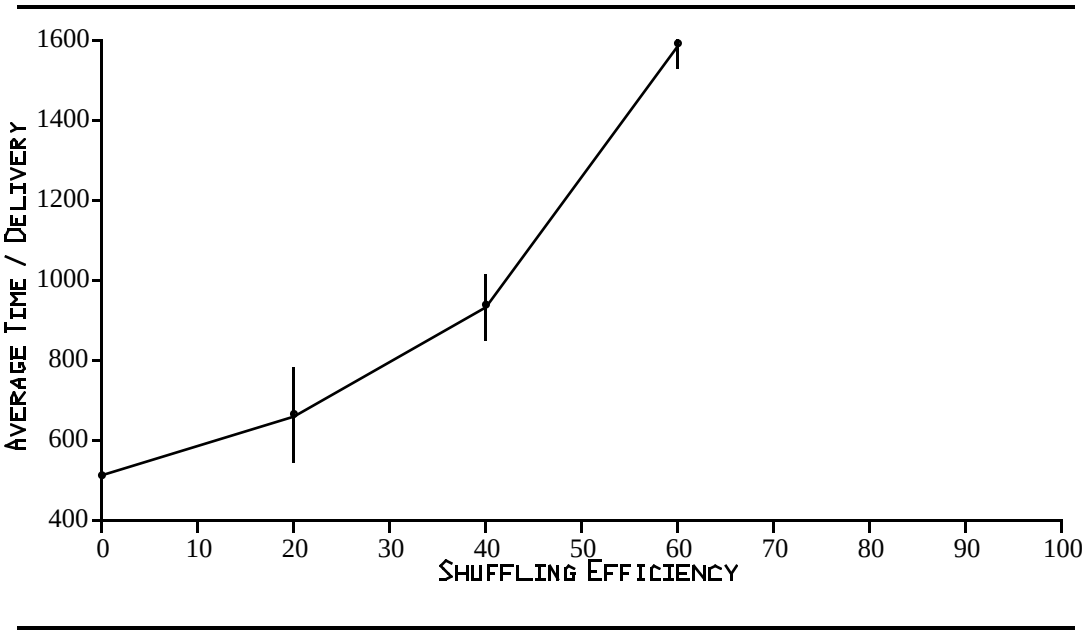



**Figure 6.9**: Results with Different Shuffling Efficiencies

The curve shows the increasing cost of extra actions caused by the increasing number of memory errors and new item arrivals. Not all of the extra time is due to failed primitive actions, however. Incorrect actions are taken on a grander scale as well. When the truck sets off to find new rocks or full fuel-drums at some location, the chances of them actually being there decreases with increased shuffling efficiency. This effect does not result in actual failed primitive actions because the truck moves around without problem and notices missing items on arrival, before trying to manipulate them. The extra travelling causes more refueling stops, though, and that increases average delivery times.

The sample of single run results shown in Table 6.2 shows these trends quite clearly. Notice in particular, the large jump in the percentage of sensing tasks between a static world (shuffling efficiency 0%) and a very random world (shuffling efficiency

| Experiment | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Shuffle Efficiency | 0% | 20% | 40% | 60% | 80% |
| Tasks Completed | 9 | 9 | 9 | 7 | 4 |
| Time Taken | 3966 | 7028 | 7300 | 9102 | 7330 |
| Effector Actions | 659 | 859 | 960 | 1123 | 842 |
| Sensor Actions | 281 | 494 | 592 | 783 | 719 |
| Failed Actions | 16 | 61 | 85 | 116 | 221 |
| Percent Sensor | 30% | 37% | 38% | 41% | 46% |
| Percent Failed | 2% | 5% | 5% | 6% | 14% |

**Table 6.2**: A Series of Experiments with Various Shuffling Efficiencies

80%). There is a similar large increase in the number of failed primitive actions and top-level delivery tasks.

### 6.3.4 Experiment Three

The third experiment is the same as the second but the shuffling efficiency is always reset to 0% at time 3960 (approximately half way through each run). The bottom curve in Figure 6.10 shows the results with square data points. The top curve shows the results from the second experiment (*i.e.*, it is the same curve as in Figure 6.9). The results show the same increasing delivery time as in experiment two, but the increase is not as fast. Furthermore, the system continues to complete all of its delivery tasks up to an initial shuffling interval over 60%.

The decrease in steepness is due to the error correcting properties of the execution system. Constant shuffling keeps the memory in state of disarray and many erroneous actions are the result. However, at time 3960 the world becomes static and the system begins to correct the errors in its memory. Eventually, all of the errors are corrected and no more mistakes are made. The system seldom fails to finish a delivery task that it has not given up on before time 3960.

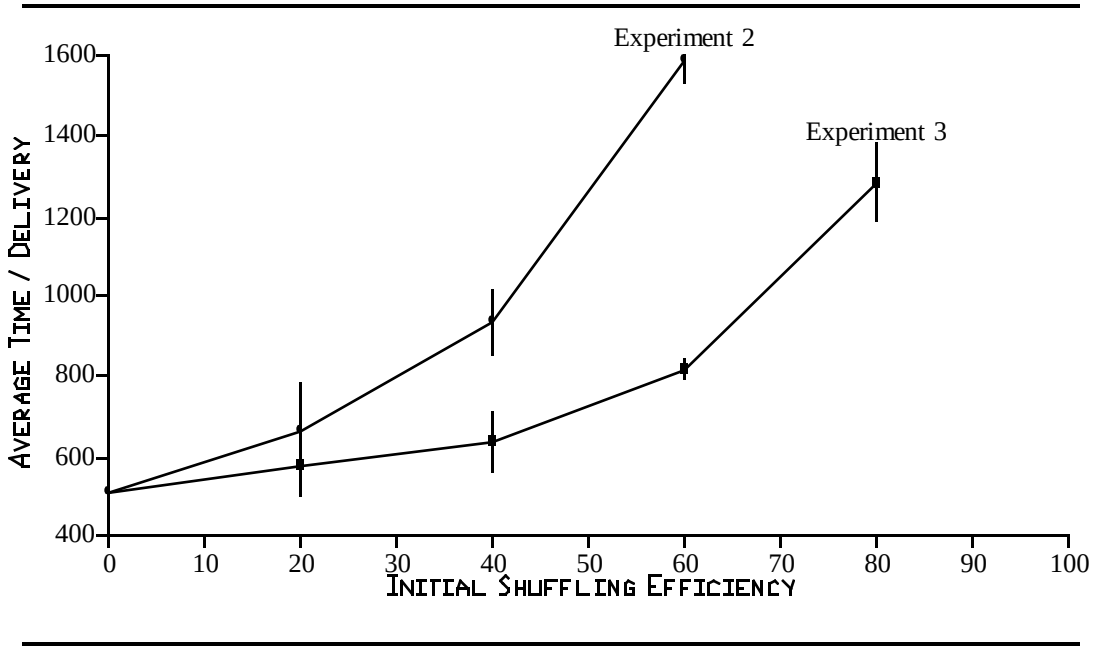



**Figure 6.10**: More Results with Different Shuffling Efficiencies

### 6.3.5 Discussion of Large Experiments

The main point of these experiments is to show that the RAP system continues to cope effectively over long series of actions. Errors in the RAP memory do not remain uncorrected and accumulate so the system manages to stay in touch with the world most of the time. The robustness that ensures this behavior is a combination of both the inherent tendency of the RAP system to retry failed actions and the specific sensing strategies used by the RAP library to deal with the delivery domain.

The first experimental curve shows that the RAP system copes quite effectively with occasional randomness in the world. In fact, shuffling intervals greater than 30 time units do not have any an appreciable effect on the system's behavior. However, an interval below 30 time units begins to slow the system down.

The second curve shows more clearly the way the RAP system slows down as its memory becomes less and less accurate. The slow down is due primarily to actions chosen on mistaken information. Such actions take up time but do not produce desired effects so they must be repeated when new information becomes available. When the world is changing rapidly, new information becomes available often and the system also spends more and more time in sensing tasks. In a static world information only

has to be gathered once at a given location, but when items move around it must be gathered again and again.

Finally, the third curve shows that the system can recover from even very random situations and carry on effectively when the world calms down again. This lack of error accumulation and propagation is an important requirement for any execution system that must deal with a dynamic world.

## 6.4 Summary of Experiments and Examples

This chapter offers evidence that the RAP system is an effective way to implement situation-driven execution behavior in a complex, dynamic domain. The evidence consists of two types of data gathered using an actual implementation of the system in the simulated delivery-truck domain.

The first type of evidence presented consists of execution traces following the truck through the execution of two typical behaviors. The first trace shows the way the system adapts its actions to recover from errors due to incomplete information and primitive action failure. The second trace shows the system noticing and reacting to an emergency situation. These two traces taken together show that the RAP system can cope with complexity and dynamic situations on a small scale.

The second type of evidence presented consists of aggregate experimental data to show that the system's small scale behavior scales up to much larger problems. While the experimental evidence cannot be particularly rigorous, it is does show clearly that, in the delivery domain at least, the RAP system is capable of coping robustly with continuing levels of uncertainty and randomness in the world. Furthermore, the experiments show that even when the world becomes too random to deal with effectively, the RAP system can recover after the randomness decreases by slowly repairing its model of the world.

### 6.4.1 Unexpected Complexity

In Chapter 5, the claim was made that acting in complex, dynamic worlds requires the use of behaviors that are more intricate than might first be imagined. The need

for idiosyncratic sensing strategies to support many tasks, the need to exploit plan-revision opportunities, and difficulties in maintaining constraints on the world all complicate the construction of effective RAP representations. As further evidence for the claim, let me relate two examples of unexpected behavior that required fine tuning the RAPs for the delivery truck domain.

The first problem appeared after I thought I had finished the RAPs required to describe the rock delivery tasks. The algorithm I encoded was essentially:

1. Go to a location with rocks of the correct color
2. Load as many rocks as needed (or can be carried)
3. Go to the destination location
4. Unload the rocks and count them as they are put down
5. If the count shows more rocks are required to fill the order, go to step 1

I made a few trial runs and in each case the truck went to a location with several rocks, picked up as many as it needed, and delivered them as expected. Later, however, I watched the truck perform the following sequence of actions. It was assigned the task of delivering three red rocks to factory-1 and set off to find some. Along the way to the red quarry, a red rock was observed at another location and the truck, taking a plan-revision opportunity, stopped to use it instead. The truck picked up the rock, went to factory-1 and put it down. At this point, two more rocks were still required and the closest red rock, of course, was the one it had just put down. Thus, the truck delivered one rock and then picked that same rock up and put it down again two more times. There are many ways to fix this bug and the most appropriate one depends on the semantics one wants to give the delivery task. For example, one might have the succeed clause check the rocks on hand at the factory rather than count those delivered, or always get new rocks only from the quarry. I changed step 1 in the algorithm to be:

- Go to a location holding rocks of the correct color but which is not the rock destination

The number of rocks delivered is still determined by count because the factories in the delivery truck domain consume rocks as they arrive. If more than one trip was required and the factory consumes the first batch of rocks before the truck gets back, checking the succeed clause against the rocks on hand will not work.

The second problem appeared only when the shuffling demon was reasonably active and resulted in the truck getting into an almost endless search for fuel. The refueling algorithm encoded into the RAPs at the time was:

1. If there are no accessible fuel drums, go to a location containing fuel drums
2. Dump local fuel-drums into the fuel-bay
3. If the truck is not full, go to step 1

This algorithm works fine when fuel drums are kept together in groups, but when the shuffling demon is active, fuel drums get scattered around the world in sets of one or two. When that happens, the system can get into a situation where it goes to a location expecting to refuel and only find one full drum. It pours that drum into the truck, finds it is still low on fuel, and goes to another location where it again only finds one full drum. If the truck uses most of its new fuel travelling between drums, this cycle of chasing single fuel drums around the world can go on for a long time. To correct this problem I changed step 1 in the refueling algorithm to be:

1. If there are no accessible fuel drums, go to a location that is *likely* to hold enough fuel to fill up.

Such a condition can be represented using a BELIEVE clause to check the age of each fuel drum location assertion and reject old ones that are likely to have been changed by the shuffling demon. The BELIEVE time interval should change with the shuffling demon if the RAPs are to cope with different shuffling frequencies, but I wanted to keep the RAPs constant over the experiments so I fixed the time interval below the smallest shuffling interval. This has the effect of almost always sending the truck to the fuel depot when it needs fuel because it is the only location where fuel drums are always likely to be present.

The interesting point is that dealing with dynamic worlds requires the semantics of tasks to be worked out very carefully. The complexity of real domains has a significant

effect on the way behaviors must be encoded and large experiments must be used to show that representations are adequate.

# Chapter 7

# Conclusions

The preceding chapters describe the RAP approach to situation-driven execution and give examples of the behavior of an implementation of the RAP system in a simulated delivery-truck domain. Chapters 2 and 3 describe the RAP execution system and the way it interacts with the world to generate the behavior required for situation-driven execution. Chapter 4 describes the RAP representation language and the way it formally encodes run time task behaviors. Chapter 5 talks about several interesting issues that come up when trying to construct RAPs (and hence, sketchy plans) that must cope with a complex, dynamic world. In particular, sensing strategies, opportunism, state maintenance, and reaction task organization are discussed. Each of these issues has been addressed only peripherally in the traditional planning literature even though they all have a profound effect on intelligent execution-time behavior. Finally, Chapter 6 shows examples of the way the RAP system works in simulated delivery truck domain. This chapter wraps up discussion of the RAP approach to execution by comparing it to other recent attempts to break out of the traditional plan/execute paradigm for robot control. Along the way, the outline of a complete robot control system based on the RAP system is also proposed.

## 7.1 Summary of RAP System

Traditional AI approaches to robot control revolve around the notion of a central planner. The planner takes all of the goals assigned to the robot and constructs a

detailed plan consisting of primitive robot actions. The robot then executes each action in turn to achieve its goals. To make this procedure work, the planner must be able to anticipate every state the robot will encounter so that it can choose appropriate primitive actions to put in the plan. Unfortunately, in complex, dynamic worlds, it is impossible to project the robot's future in detail. There is no way to know the initial state of the whole world, and uncertain action outcomes make the projection of what is known difficult. Agents and processes not under the planner's control complicate things even further. Without the ability to project the robot's future in detail, planners cannot construct detailed plans of action.

The solution to the problem of inadequate projection is to use planning operators that are more abstract than primitive actions and which hide much of the complexity of the real world. Such operators must adapt to a variety of situations and handle a variety of routine error conditions on their own so they will be applicable in many world states. Those world states can then be treated as one by the planner because, during execution, the operator will adjust to the situation encountered and generate appropriate results regardless of the initial state. Therefore, a planner need only move from one range of states to another range of states. As long as an applicable operator can be found for each range of states, there is no need to go into more detail.

The RAP system described in the previous chapters is a concrete proposal for constructing planning operators that adapt to different world states at execution time. RAPs have the following major characteristics:

- They adapt to the actual situation encountered at execution time.
- They are robust in the face of primitive action errors, uncertainty, and interference.
- They do not look into the future.

A RAP is designed to carry out a particular behavior starting from a variety of possible initial states. Each RAP packages up methods for achieving its goal from different initial states and, at execution time, chooses one based on the situation actually encountered. This ability to choose between different methods at execution time is what makes the RAP system adaptable.

RAPs are also robust in the face of execution errors. A RAP method may fail for several reasons: it may issue a primitive action request that fails because of a hardware problem, it may issue an action request based on an incorrect picture of the current situation, or another agent may interfere to prevent the actions generated by the method from achieving their purpose. To cope with such failures, a RAP maintains a success check and, if a method does not achieve the state desired, the RAP tries again by choosing another method. Only when all possible methods have been tried several times does a RAP give up. Since most primitive action failures are either transient (like dropping an object) or generate new information (like finding an object too heavy), they can be corrected by trying again. Similarly, uncertain information is usually corrected even when a method fails and when the RAP chooses a new method this new information will be taken into account.

While a planner is primarily interested in RAPs as abstract operators, at execution time the RAP system must be much more. When the world is simple and friendly, a plan may proceed from start to finish and all that is needed at execution time is for each RAP-defined task to choose methods as required. However, realistic worlds will often present opportunities for both improvement and disaster. Opportunities arise whenever situations in the world cannot be predicted by the planning system. As new information is uncovered during execution, it may be appropriate to interrupt the current plan and do something else. Emergency situations are exactly the same — a new fact comes to light and the system must drop what it is doing and respond immediately.

The RAP system deals with opportunities and emergencies by allowing reaction tasks to exist on the execution agenda concurrently with the tasks making up a plan. Examples of such tasks are procedures to deal with running low on fuel, discovering a fire, or encountering enemy troops. Reaction tasks wait on the task agenda until the situation requiring their execution arises and then they wake up and handle the problem. The RAP system manages this type of behavior using three mechanisms: state monitors, task priorities, and task families. Each task to be executed by the RAP system is given both a priority and a family. When choosing a task from the execution agenda, the RAP system always selects the highest priority task that is eligible to run. If several tasks have the same priority, then a task from the same family as the last task is selected. Executing the tasks from one family at a time focuses the

system's attention on one high level goal at a time. However, priorities allow the system's attention to be shifted from one goal to another when appropriate. As long as reaction tasks are given a high priority, they will shift attention to themselves as soon as they become eligible to run. To prevent reaction tasks from monopolizing the system's attention and shutting other tasks out altogether, reaction tasks remain ineligible to run until their monitor clauses are believed satisfied in the world.

Reaction tasks can be set up by RAPs, and hence be part of a planner's original sketchy plan, or they can be a permanent part of the system's execution agenda. In the robot delivery-truck domain, it is best to place a task to deal with low fuel on the execution agenda permanently because it is difficult for a planner to anticipate every trip that may have to be made while carrying out a given goal. To ensure the truck never (well, almost never) runs out of fuel, this task is always on hand to fill up when fuel runs low. On the other hand, when a planner sends the truck down a very sparsely populated road, it may be wise to set up a special task to stop and get fuel whenever a depot is encountered. When depots are few and far between, it is better to fill up when possible, rather than wait for fuel to run low while the truck is too far away from a depot to recover.

An important feature of the RAP execution system is the way it returns to a task family after a reaction task interruption. When a high priority reaction task becomes eligible to run, it immediately displaces the current task family and takes control of execution. After the emergency or opportunity has been dealt with, tasks from the interrupted family will be preferred over other tasks of the same priority. However, the next task in that family is likely to be part of a method instantiated before the interruption in the context of a different situation. For example, the truck may have been travelling to the factory before it was interrupted, and a new route will be necessary because the interruption moved the truck to another location. The inherent robustness and adaptability of the RAP system will come into play at this point and the old method will fail and be replaced by a new one. Thus, the same execution characteristics that make the RAP system appropriate as an abstract plan-operator representation also help to deal with interruptions effectively.

## 7.2 Extending the RAP System

The discussion of the RAP execution system has so far treated the system as if it were designed to carry out tasks all by itself. The experiments described in Chapter 6 begin with tasks defined by single RAPs and proceed to completion entirely within the RAP system. No planning is done and primitive actions are completely self-contained entities that get executed one at a time. However, the RAP system is not intended to stand on its own; it is designed to be the middle layer in a three-layer robot control system. A robot behavior controller must exist below the RAP system to handle concurrent, continuous interactions with the environment, and a planner must exist above the RAP system to look into the future and head off problems before they arise.

Many design decisions confronted while constructing the RAP system were made with a view to the larger system in which it might be embedded. For instance, there are several obvious limitations in the way the the RAP system interacts with the world but, while these limitations restrict the types of behavior the RAP system can exhibit by itself, they can be eliminated through the addition of a planner and behavior controller. This section discusses the major limitations of the RAP system and outlines a three-layer robot control model that will remove them.

### 7.2.1 Limitations of the RAP Approach

There are four major limitations on the RAP approach to execution control:

- Assumptions about primitive actions and sensor reports are not realistic.
- Choosing a method often requires committing to a course of action before all relevant information is known.
- No look-ahead is used in task and method choice even though some would obviously be useful.
- The time used by the RAP interpreter is not accounted for in the approach.

Each of these limitations is discussed below.

### Unrealistic Primitive Actions

Unrealistic assumptions about primitive actions take two forms: primitive actions are atomic, and primitive actions can only be executed one at a time. Atomic primitive actions imply that only a single command is required to carry out a behavior like "eye-examine", "arm-grasp", or "arm-toggle". However, it is quite clear that a complicated set of joint movements tightly coupled to sensory feedback will be required to actually carry out such actions. The RAP system can be pushed further by choosing simpler primitive actions, but at some point they must meet more traditional control processes.

Similarly, there are situations where it is obviously possible to do more than one thing at a time and, if the RAP interpreter continued running while the robot hardware was carrying out a primitive action, it would be able to choose other primitives to exploit those situations.[1] For example, one task might request that `arm1` pickup `item-34`, and, while waiting for this to complete, the RAP system could work on another task that wants `arm2` to pickup `item-57`. Two problems arise when the interpreter does not stop and wait, however. The first is that the RAP memory cannot reflect the true state of the world while a primitive action is in progress. The world will be changing (*i.e.*, `arm1` will be moving) while the tasks choosing to use `arm2` are executing. The second problem is that primitive actions requiring the same resource must not execute at the same time. The robot cannot move `arm1` to two different places at the same time. RAPs would have to wait, and not fail, when required resources were busy.

Another unrealistic assumption about primitive actions is that sensory information takes the form of symbolic messages about items in the world. The assumption that sensory data come in as predicates has a long tradition in AI and, while the RAP system removes the assumption of fixed item designators, current sensor technology has trouble delivering predicates in any form. This is an area that requires much more investigation.

[1] Recall from Chapter 3 that the current RAP interpreter algorithm waits while a primitive action is executed. No further task expansions are considered until results from the primitive action are returned by the hardware interface.

### Interpreter Time Ignored

A limitation that further complicates the use of primitive actions is that no provision has been made to account for the time taken by the RAP interpreter. In the simulated delivery-truck domain, "real" time passes only during the execution of primitive actions — the time required to assimilate new sensory data and choose the next primitive action is essentially "off line". This limitation is easily lifted, however, simply by letting time pass while the interpreter is running. The only problem with this change is that, when several task expansions are required before a primitive action is issued, the system may lose touch with changes occuring in the world around it. The solution to this problem is to assimilate new sensor information into memory as part of each task execution cycle rather than only when primitive actions finish. The RAP system cycle of "choose a task", "choose a method", "instantiate", has been kept simple and short, and queries to RAP memory do not include inference, only unification. Therefore, each cycle will be short and, if memory updates are done during each cycle, the system should not lose touch with the world. Occasionally the system will cycle through many method choices before actually executing a primitive action but at each cycle it will be able to react to changes in the world. As long as reaction tasks do not require too many cycles, the system will have no trouble coping. Kaelbling [1986a] discusses some of these same issues.

While incorporating "thinking" time into RAP execution is straightforward, it can have a significant effect on planning. When a planner is trying to construct a sketchy plan to achieve time-critical goals, it must be able to estimate the duration of each task in the plan. Such estimates will have to include the time taken by the RAP interpreter as well as the time taken by the primitive actions themselves. This has always been a problem in planning research [Dean and Boddy, 1988], and while the RAP system does not make it any easier, it doesn't make it any harder either.

### Premature Method Commitment

Another aspect of the RAP system is a problem from some points of view but not others. When a task chooses and instantiates a method, it commits itself to a future course of action and does not get a chance to change its mind until the method completes. Thus, task methods must explicitly set up plan-revision opportunity mon-

itors when they feel they might want to change their minds in the future. Similarly, methods must make subtask preconditions explicit, through FOR clauses, to cause execution to stop when things are not working out as expected. Without such notations, every step in a method will run before its parent can reconsider and make a different choice. Incorporating appropriate notations is often complex and non-intuitive (see Section 5.1.2 for an example).

On the other hand, committing to a course of action has several advantages if look-ahead can be used to coordinate the actions of independent tasks. Should a planning system be watching over the RAP system to keep it out of trouble, the RAP system's expected behavior must be available for examination. The planner will want to know what the RAP system is going to do so changes can be made before trouble develops. Because RAP described tasks commit to complete methods all at once, the task agenda is just such a picture of the RAP system's intentions. All of the system's goals are represented on the agenda and, as methods for a goal's execution are chosen, the steps required are added to the agenda. Furthermore, if the agenda is changed, the system's behavior will change as well. Thus, committing to a course of action provides a clear picture of the expected future at the expense of requiring explicit mechanisms for abandoning methods when they become inappropriate.

### No Projection of the Future

The final obvious limitation of the RAP system is that it never looks into the future. This was a design goal, and not an oversight, so, strictly speaking, it is not a problem with the approach. However, anticipating the future is an important part of intelligent behavior and can often be used to avoid difficulties. For example, choosing a route to the factory that includes crossing a low capacity bridge is a mistake when several heavy boxes are to be picked up along the way. Without thinking about the boxes before the bridge is reached, there is no way to anticipate the problem of being too heavy. Once at the bridge, the RAP system can prevent disaster by moving the boxes across in several trips, or by choosing a route around the bridge, but it would be wiser to avoid the bridge in the first place.

### 7.2.2 A Complete Robot Control System

The limitations of the RAP approach to situation-driven execution are mitigated by placing the RAP system into a complete three-tiered robot control system. Such a system would include:

- A planner to project the future and anticipate problems.
- The RAP system to adapt behavior to the current situation.
- An action system to actually perform primitive actions.

In fact, the RAP system was designed with exactly such a three-tiered robot control system in mind. Figure 7.1 shows the way the whole system might be structured.

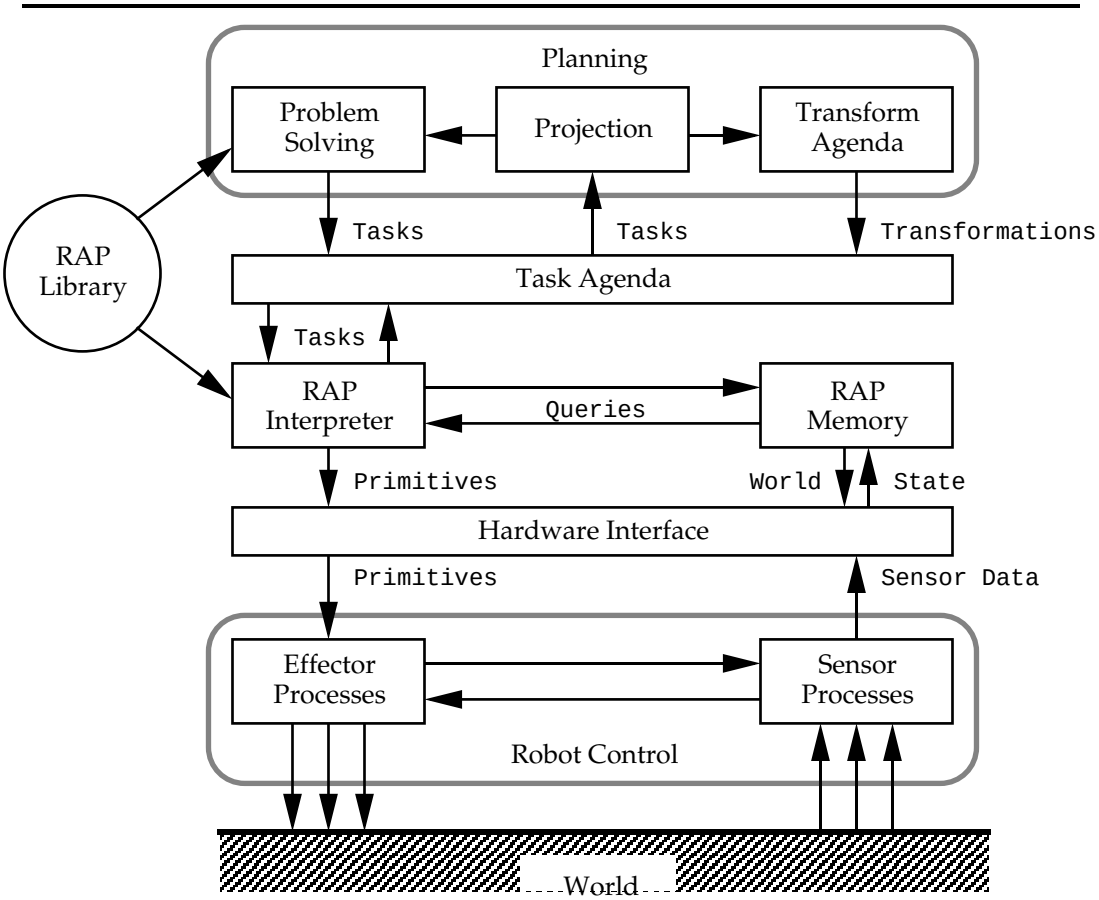



**Figure 7.1**: A Complete Robot Control System

Each of the three layers in this design deals with a different facet of robot control. The low-level action system coordinates all of the concurrent, continuous processes required to control real robot hardware. In general, every primitive robot action executed will involve moving many joints simultaneously and coordinating those motions with immediate, continuous sensory feedback. For example, grasping a wrench might require positioning a robot's arm and gripper according to orientation information from vision sensors. Furthermore, the pressure required to hold onto the wrench will depend on its weight and surface properties which can be determined through feedback from the gripper. Procedures to carry out these behaviors will most easily be encoded as separate programs so they can be made fast and tightly interconnected. The purpose of the action system is to package up bundles of such continuous procedures into more abstract primitive actions for use by the RAP system.

The RAP system coordinates the generation of primitive actions to carry out higher-level goals in varied and changing situations. Primitive actions implement single, discrete behaviors; RAPs package such behaviors into methods for achieving more complex goals. However, the RAP system does not look into the future to anticipate interactions between the methods chosen for different goals, and it does not try to construct new methods when none of the methods it knows is appropriate.

The top-level planning system has the job of constructing sketchy plans for goals that do not already have RAPs and the job of coordinating the execution of methods chosen by independent RAPs. The RAP system gives the planner the choice of creating sketchy plans at many different levels of abstraction because it can execute tasks that are specified at many different levels of detail. However, as RAP execution of a sketchy plan proceeds, task methods chosen by the interpreter may interact in ways that produce problems or inefficiencies. Therefore, at execution time, the planner has the additional responsibility of watching the execution agenda and transforming it when the proposed courses of action from different goals suggest undesirable futures.

To ease the planner's manipulation of tasks, the RAP system hides the problems of coping with failures and changing situations. At the same time, to ease the RAP system's manipulation of simple behaviors, the execution system hides the problems involved in continuous robot control. It is important to note that there is an overlapping of functionality at the interfaces between these layers. Many primitive RAPs might just as easily be encoded as procedures in the action system and the most

abstract RAPs might easily be replaced by planning. Such flexibility is encouraging because it is intuitively clear that the lines between planning, acting, and adaptability cannot be sharp.

## 7.3 A Robot Action Controller

The main purpose of a low-level action controller is to hide the complexity of controlling a real robot from the RAP execution system. The RAP system treats primitive actions as if they were discrete, atomic entities; however, the world does not behave in discrete, atomic ways. Real robots move effectors with continuous motor commands, and real sensors must track quantities that change continuously over time. Pouring a given amount of water from a pitcher into a glass is an exquisite example of motion and sensing coordination that requires continuous real-time arm position adjustment. The RAP system assumptions about primitive actions can be preserved, however, if a robot controller can be built to carry out discrete primitive-actions using continuous robot-commands.

### 7.3.1 Executing Primitive Actions

Many of the requirements of an action controller are areas of current active research. The most promising idea (in my opinion) is for discrete processes like "travel down the hall", and "move arm1 to item-34", to be implemented as a collection of cooperating, concurrent procedures. For example, "travel down the hall" might break down into independent procedures like "move straight ahead", "stay way from the left wall", "stay away from the right wall", and "stop if something is blocking the way". The move ahead procedure will send commands to the wheels to move forward, while the wall avoiding procedures will direct sensors at the appropriate wall and steer the wheels away if the wall gets to close. Finally, the stop procedure will signal the end of the whole behavior once something looms in front of the robot. Sensors and effectors are employed as necessary by each procedure to carry out its simple function and interesting aggregate behavior emerges from the interplay between all active procedures and the real world.

A RAP primitive action would become a request for the action controller to initiate and monitor a particular set of concurrent procedures. The action controller would then do three things:

- Set up the procedures needed to carry out the primitive action.
- Monitor the progress of each procedure and terminate them all when any one signals that the primitive action as a whole has either succeeded or failed.
- Arbitrate contradictory robot commands originating from separate procedures.

Like RAPs themselves, primitive actions must activate not just procedures to move effectors around, but also procedures to monitor success and failure conditions so that the logical end of each primitive action can be detected in the world.[2] A well developed implementation of just such a system is described in Payton [1986]. The system is called a reflexive planner and it is used as a low-level controller for the ALV. Arkin [1987] discusses a very similar system although less of it has been implemented.

### 7.3.2 Generating Sensory Predicates

Some work has also been done on turning continuously changing properties of the world into discrete assertions about the current world state. The basic idea is very similar to that of using concurrent procedures to implement discrete actions. Significant features of the world are assigned separate continuous processes. Each process monitors the state of the feature and, as long as the state stays within some range of acceptability, the process stays quiet. Whenever the feature changes in a significant way, however, the process generates a notification to memory. For example, each item moving near the robot might be assigned a separate process. As long as no item was

[2]The overlap of capabilities between the RAP system and the controller is particularly apparent here. RAPs can be used to deal with robot control to a very low level, but eventually a point is reached where it easier and more perspicuous to use processes coded more like traditional feedback control programs. Consider the problem of plugging two extension cords together. A cord must be picked up in each hand and the mating pieces must be brought together. Even if the arms have a low positioning error, frequent visual feedback is required to ensure that the end pieces meet correctly when they get close together. While RAPs can do much of this work (if primitive actions are pushed to the level of joint motor commands), it probably makes more sense to use specific arm motion processes that couple tightly with appropriate visual feedback. More experience with robots is required to determine the best dividing line between RAPs and lower level processes.

in danger of colliding with the robot, each process would be silent. Should an item veer into the robot's path, however, the process watching it would generate a sensor notification saying that the item was on a collision course. These ideas are well presented in Hendler and Sanborn [1987], where the processes are called state-of-affairs monitors. The robot memory needs to be modified only when the state of affairs changes. Payton [1986] discusses a similar idea under the name of virtual sensors and the same type of processes are included in Arkin [1987].

State-of-affairs monitors help partition continuous sensor data into discrete units but they also support the RAP memory's notion of local sensor name. Assigning processes to track individual items while they remain within sensor range is the same as assigning the items temporary sensor names.

A great deal of work remains to be done in extracting predicates about item properties from real sensor data. I believe that the RAP system supplies a new starting point for applying top-down information to such problems, however. RAP memory expectation sets supply a set of candidates to begin matching real items against and thus suggest special purpose sensory routines to apply in many situations. Furthermore, tasks can invoke methods that involve even more specific sensory routines in particular circumstances. Perhaps the RAP library would be a good place to encode mechanisms for invoking procedures like Ullman's [1984] visual routines.

### 7.3.3 Interfacing to the RAP System

For the RAP system to take advantage of a low-level action controller, two of the system limitations discussed above must be lifted:

- The restriction that only one primitive action be executed at a time.
- The assumption that RAP interpretation takes no time.

If actions become collections of concurrent processes, it should be easy to instantiate more than one action at a time as long as the processes involved do not conflict. If we assume that the controller notifies the RAP system when a new primitive request conflicts with processes already active, the RAP interpreter would know when it would have to wait. Instead of always waiting for primitives to finish, it would only have

to wait when a primitive could not yet be started. Such a change would let the interpreter continue to execute tasks from the agenda while primitive actions were under way.

With the use of separate sensing processes, sensory data will become available any time a state-of-affairs monitor is active and not just during primitive actions. However, if the interpreter cycle is extended to be "pick a task", "pick a method", "instantiate method", "consider new sensor data", as suggested in Section 7.2.1, everything will be fine. Instead of memory being updated at the end of a primitive action, it would be updated after every task expansion cycle. Task expansion cycles can then take up real time as well. If memory changes in a significant way in between primitive actions, any errors in task expansions due to the change will be handled exactly as they would be if the changes occurred only at primitive action boundaries.

Except for changing the RAP interpreter algorithm to allow memory updates at every task execution cycle, an action controller like that described above would leave the RAP system virtually unchanged.

## 7.4 Planning with The RAP System

The top layer of the three-tiered robot control architecture of which the RAP system is to be a part is a planner with two separate responsibilities. The planner must create the original sketchy plans required to carry out the system's goals and, at execution time, the planner must watch the RAP system's progress at following the sketchy plan, and adjust things when it looks as if problems may develop. This section discusses each of these responsibilities separately.

### 7.4.1 Problem Solving and Sketchy Plans

The first application for planning in a complete robot control system is in constructing the initial sketchy plans required to achieve particular goals. The RAP system has the responsibility of actually executing sketchy plans so, as discussed in Section 4.10.1, sketchy plans must take the same form as RAP methods. Let us call the process of constructing a new RAP, or constructing a new method for an existing RAP, "problem solving" to differentiate it from other responsibilities that a planner might have.

The characteristics of problem solving situations are those of traditional AI planning research. A goal, or conjunction of goals, is assigned to the planning system along with an initial description of the world, and the planner must find a combination of known RAPs that will achieve the assigned goals. The whole RAP library is at the disposal of the planner as a source of planning operators and almost any planning paradigm can be supported. Single RAPs can be used as linear planning operators to support STRIPS type planning [Fikes and Nilsson, 1971], and RAPs at different levels can be used to support hierarchical planning [Sacerdoti, 1975, Wilkins, 1988, Dean *et al.*, 1988],

### Problem Solving for the RAP System

There are actually two places that problem solving finds application in the RAP system: in the initial construction of sketchy plans, and in the construction of new methods at execution time. A sketchy plan is a RAP constructed on demand to describe a task for achieving a new system goal. When the system is assigned goals that map directly into RAPs in the library there is no need for initial problem solving — an appropriate task can be instantiated immediately using the RAP in the library.[3] However, one can imagine assigning a robot goals for which it does not have a RAP at its fingertips. In such cases, a new RAP must be constructed. General planning techniques and specialized problem solvers are both applicable in such situations. Note that this type of top-level planning can often be done "off line" before the robot actually begins to execute anything.

Problem solving can also be used to construct new RAP methods while plan execution is underway. RAPs in the library will often be incomplete, holding methods that cover only some situations and not others. For example, within the spirit of the RAP approach, routes between locations must be hard coded into RAP methods. Clever indexing schemes can be used to reduce the problems this causes (*i.e.*, there can be a set of routes between cities, another set within each city to reach certain areas, and then other routes within areas), but at some point the system will almost certainly want to travel between two points not included in its known

[3]This use of existing RAPs to carry out goals was used in the examples and experiments in Chapter 6. All initial reaction tasks and all delivery tasks assigned to the system mapped directly into RAPs in the library. Thus, no sketchy plan construction was required.

routes. This desire will manifest itself as the instantiation of a task something like `(travel-between quarry-3 factory-4)` but where the travel-between RAP will not have a method linking `quarry-3` and `factory-4`. Rather than simply fail in a situation like this, the RAP system for the delivery truck domain was extended to include a route planning problem solver. The problem solver has a map of the domain and is called the first time the truck must travel between two locations. Notice that once a method is constructed, it need never be constructed again — it is just added to the "travel-known-route" RAP in the library.

### Extensions to the RAP System for Problem Solving

The RAP system must be extended in two ways to support these uses of planner problem solving. First, RAPs in the RAP library must have additional annotations to make them into full-blown planning operators. RAPs already have succeed clauses to specify what they will achieve, and precondition clauses to specify the situations in which they will achieve it. However, RAPs will generally have effects that are not included in any of their clauses. For instance, a travel RAP will consume fuel as a side-effect, and a shooting RAP will consume ammunition. At execution time, the RAP system lets the world keep track of these things, but a planner will want to know these effects in advance so that it can properly project the future. Therefore, RAP descriptions have to be extended to include a list of effects that can be expected when a task described by the RAP is executed in the world. Such descriptions will have to be approximate, like a task's duration, because they are needed before a method is chosen for the task. The use of operator descriptions before the operators are expanded into more detail is discussed in Dean *et al.* [1987].

The second possible alteration of the RAP system is to have it call problem solvers when particular RAPs need new methods. Such an extension is very simple and, as mentioned above, a preliminary algorithm was incorporated into the RAP implementation in the delivery truck domain. When a task comes up for execution and *none* of its methods are applicable in the current situation, the interpreter checks to see if there is a special purpose problem solver for the task. If there is, the interpreter calls the problem solver to see if it can generate a new method for the task to use in the current situation. If a new method is produced, it is instantiated, and if a new method is not produced the task fails. Only the route planning problem solver

is used in the delivery truck domain, but one can imagine adding problem solvers for many specialized situations — complicated grasping problems or playing games for example.

The fact that a given problem needs to be solved only once is an important feature that separates problem solving from other responsibilities of the control system planner. Building a sketchy plan is building a new RAP, and constructing a new method at execution time is adding to an existing RAP. In both cases, the results can be added to the RAP library to be used whenever the same problem arises in the future. Saving and reusing the results of problem solving activity this way may provide a natural link between cased-based planning research and RAP execution. A case-based planner looks at the ways a problem has been solved in the past and attempts to adapt those solutions to new situations when they arise [Hammond, 1986]. Since the RAP library saves previous problem solutions for reuse anyway, it might be possible to extend into a case library as well as a traditional planning operator library. This would be an interesting avenue for further research.

### 7.4.2 Altering Plans During Execution

Another responsibility of the planner in a robot control system is to look into the future to detect and prevent problems with proposed courses of action before they arise. The task agenda contains all of the things that the RAP system intends to do, and can be viewed as the system's current plan. Thus, the planner would have the job of monitoring the task agenda, predicting the future to expect should all tasks on the agenda be executed, and altering those tasks to achieve a "better" future when appropriate.

For example, suppose the agenda contains a task to deliver two blue rocks to factory-1 and a separate, independent task to deliver two yellow rocks to the same factory. Given the RAP system task-selection heuristics discussed in Chapter 3, one of these tasks will be selected at random and will be run to completion before the other is considered. If yellow and blue rocks are located near each other, it would be more efficient to collect all of the rocks and make a single trip to the factory. The planner should be able to realize that fact and replace the two separate tasks with a single task to deliver two yellow and two blue rocks to factory-1.

### Transformational Planning

To monitor the RAP system at execution time, the planner would act on the task agenda using the following cycle:

- Project the expected future of the task agenda as best it can.
- Look for bugs or inefficiencies in the projection.
- Use the bugs or inefficiencies to suggest transformations to the agenda.
- Apply a suggested transformation.

To support this algorithm, the planner must be able to project the future of a plan containing steps at different levels of detail. The agenda holds tasks at various levels of expansion but all must be considered to generate a reasonable picture of the future. Projecting the future in this situation is exactly the same as projecting the future during problem solving, and thus requires that RAPs at all levels of abstraction be annotated with their expected effects. The problem of projecting context-dependent, uncertain effects is an active area of current AI research [Hanks, 1988, Dean and Kanazawa, 1988, Shoham, 1986].

Once a bug, or area of inefficiency, has been identified in the task agenda, the planner should apply a transformation to the agenda to fix the problem. A transformation might change all or part of the agenda by adding new tasks, reordering old tasks, or replacing old tasks with new ones, but it must leave the task agenda in an executable state. Assuming that planning takes a significant amount of time (not an unreasonable assumption given current planners), the RAP system will often be executing tasks while the planner is working to remove bugs and inefficiencies from other parts of the agenda. To support this concurrency, transformations that leave the agenda in a usable state will be necessary. If bugs or inefficiencies are inadvertently left in the agenda, or cannot be fixed quickly enough, the RAP system will sort them out as best it can when they actually arise. A planning system designed very much along these lines has been described by Linden [1987] as a top-level mission planner for the AIV. A restricted form of transformation is used to fix bugs in geological

interpretations in Simmons [1988] and transformational techniques are also being investigated in automatic programming research. A good pointer into that literature is Fickas [1985].

#### Extensions to the RAP System for Execution-Time Planning

If the planner and the RAP interpreter are run concurrently, the planner will have the problem of trying to project the future of a changing agenda. On the other hand, if the planner is run interleaved with the interpreter, there may be long pauses for planning in which the robot is "dead to the world". Some sort of compromise will be necessary based on two things: the planner must be interruptable, and the RAP system must be able to provide estimates of how long it can wait before executing the next task. If the RAP interpreter tells the planner how long it can wait before executing the next task on the agenda, the planner will be able to alter its behavior — looking hard for bugs and making complex transformations when there is time, and being more superficial when time is short. Dean and Boddy [1988] suggest "anytime algorithms" as the basis of such a planner. The RAP system can already calculate allowable delays using task deadlines and durations, so making the information available to the planner should not be difficult.

The RAP system must also remain "awake" during planner activity so that it can process sensory information and execute high priority reaction tasks when necessary. If enemy troops show up while the planner is thinking about the best order in which to execute a set of delivery tasks, the task to handle enemy troops must still be executed immediately. Therefore, some type of handshaking is necessary between the planner and the interpreter. The best form of handshaking remains to be investigated.

### 7.4.3 Summary of Planning Issues

As discussed above, there are two major extensions required to interface the RAP system with a high-level planner. First, RAP representations in the library must be annotated to include projection information. Such information is needed to describe the effects of executing tasks described by the RAP in the real world. The second extension required is a handshaking protocol to allow the planner and RAP interpreter

to share the task agenda at execution time. The agenda tells the interpreter what tasks it needs to perform, and it tells the planner what to expect in its future. Thus, the interpreter is always trying to carry out the tasks on the agenda and the planner is always trying to organize them into a more coherent whole. The planner must be able to tell the interpreter when to wait before executing a task and the interpreter must be able to interrupt the planner and act immediately when the situation changes suddenly.

The planner itself must also tackle two different types of problem. It must solve well encapsulated problems while building an initial sketchy plan or a new method for an existing RAP. It must also watch over the task agenda at execution time and suggest improvements when they become appropriate. A transformational planning approach seems most promising for the latter purpose. A transformational planner can call on many different procedures to suggest fixes to the task agenda: it might have a library of fixed transformations, it might call on domain "experts" like route planners, or it might invoke a general purpose problem solver. Since transformational planning as a paradigm includes the invocation of problem solvers of many types, it is also appropriate for build new RAPs and RAP methods. Therefore, a single transformational planner may be all that is required as the top tier in a complete robot control system.

## 7.5 Related Work on Reactive Planning

"Situated robot control" and "embedded systems" have been the focus of a flurry of research in Artificial Intelligence during the last few years and three bodies of work require particular attention. The first is work on situated machines done by Brooks and Agre and Chapman. The second is work on robot control system design by Kaelbling and Rosenschein, and the third is work on the Procedural Reasoning System by Georgeff, Lansky and Schoppers. These authors claim to be solving the same problems as the RAP system, so it seemed appropriate to save discussion of their work until after the whole RAP system had been presented.

However, before discussing this work on robot control systems, there are two particularly interesting Cognitive Science books to mention. The first is a new book

by Suchman from which the term "situation-driven execution" is derived, and the second is an older book (relatively speaking) by Miller, Galanter, and Pribram which presents an idea for encapsulating behaviors that is very much like the RAP.

### 7.5.1 Situated Action and Coping Behavior

Suchman [1987] argues persuasively that actions, plans, and language are meaningless without a surrounding context. The context may be explicit, or only implied, but without it there is no way to make sense of sequences and referents. For instance, it is hard to imagine how an instruction like "push the lights inward and orient the logo upwards" could ever make sense, but the Apollo DN3000 that I am writing on has a little front panel that can be rotated a quarter turn when the machine is stood on its side. The panel has little lights on it and when you push it in it turns, just like the instruction says. The point is that actions and plans can never be specified in enough detail to avoid the need to interpret them in context at execution time. Thus, actions and plans are *situated* and have no meaning without reference to the way they will be interpreted. The RAP system can be seen as an attempt to create a situation-based interpreter for sketchy plans. Detailed RAP behavior hinges on the context in which RAPs find themselves and, therefore, the behavior described by the plan changes from situation to situation. Comparison cannot be pushed too far, however, because Suchman's book is oriented toward human-machine interaction and not toward plan and action representation.

The book by Miller, Galanter, and Pribram [1960] is oriented toward plan and action representation. They introduce the idea of a TOTE unit. TOTE stands for "test, operate, test, exit" and each TOTE unit represents a behavior. When a person wants to achieve a goal an appropriate TOTE unit is activated to carry it out. The TOTE unit first "tests" the current situation and chooses an "operation" to perform. After the operation is complete the unit "tests" the world again to make sure that goal is achieved and if so the unit "exits". If the goal is not satisfied, the TOTE unit starts over from the beginning. This basic algorithm is very similar to that used by the RAP system and Miller, Galanter, and Pribram discuss ways to use it to perform a wide variety of cognitive functions. Unfortunately, the authors do not go into the algorithm in any more detail, or present a language for representing TOTE units. Most

of the work in this thesis is aimed at exactly those problems.

While these two books are interesting, and help to put the ideas of situated behavior in focus, it is recent work in building actual robot control systems that is most relevant to the RAP system.

### 7.5.2 Machines without Representation

One recent approach to robot control discussed in the AI literature is that of building all action decision procedures into hardware. There are two main ideas driving this approach. The first is that behavior which appears complex on the surface can often arise out of simple, interacting processes. The second is that a central representation of the world is not necessary for most tasks — the world can be used to keep track of itself. To illustrate these ideas, control systems are built out of real hardware (or simulated real hardware) that enforces the constraint of no internal state.

Brooks implements these ideas using a subsumption architecture to control real robots moving around in the world [Brooks, 1986, Horswill and Brooks, 1988]. His control system consists of distinct layers of hardware that instantiate more and more abstract behaviors. A bottom level behavior might be something like "move away when an obstacle is near" while a higher level behavior might be "move forward if not blocked" or "look for a reachable location and move there". Higher levels act only by influencing the behavior of lower levels, hence, high level behaviors subsume lower level behaviors. For example, "move forward" will bias the robot motors to move forward but the lower level "avoid obstacles" will continue to turn the robot away from stationary and moving obstructions. The important aspects of this work are that all behaviors are built into hardware, and that each behavior maintains only as much state as required to carry out its local, hardwired goals. There is no overall control structure, no explicit plan, and no internal model of the world [Brooks, 1987].

Chapman and Agre argue for a similar model of robot control they call "situated activity" [Chapman and Agre, 1986]. The implementation of their model in the PENGI system does not have an explicit layered structure, but rather consists of a visual routine processor and a decision network built up of simple logic gates [Agre and Chapman, 1987]. Information about the world comes in through the visual processor and supplies input to the decision network. The decision network takes the these

inputs and selects actions to execute in the world. The decision network contains no state and is best thought of as simply a switch between inputs and actions. As the world changes, the inputs to the network change and hence different actions are switched on and off. The system exhibits abstract behaviors as actions are stepped through in response to changes in the world, but no description of these abstract behaviors is explicitly represented anywhere in the hardware.

An interesting aspect of the PENGI work is the idea of indexical-functional cues from the world. The visual routine processor is structured to return facts about the world in the form "the-bee-in-front-of-me" or "the-block-to-my-left". The referents of these facts change from moment to moment and there is no notion of fixed designators for objects in the world. There is a similarity between indexical-functional cues and the item sensor names assumed by the RAP memory. An item's sensor name can be thought of equally well as `drum-45` or "the-drum-to-my-left". Long-term memory references to an item cannot be taken for granted and must be explicitly sorted out by the RAP system.

The common features of these two approaches to robot control are:

- All behaviors are hand-crafted into the hardware.
- There is no explicit internal representation of the world.
- There is no explicit representation of the system's current goals.
- No reference can be made to specific objects in the world.

The RAP system is different with respect to each of these features.

The RAP system explicitly generates an internal model of the world and RAPs and tasks can make references to specific items in the world. It is argued in Chapter 2 that these two processes go hand in hand. A model of the world is required to remember where things are in the world and, if one assumes that sensors cannot attach fixed designators to objects in the world, a memory is required to hold descriptions of an item so it can be identified as the same object from encounter to encounter. The RAP system builds up descriptions of items in the world for both of these reasons. Brooksian robots and totally indexical-functional machines cannot remember or recognize items.

The RAP system also represents its intentions as explicit tasks on the execution agenda. The agenda represents a "plan for the future" and, thus, supplies an avenue of communication between planning and acting. The ability to analyze and alter ones behavior seems an important part of intelligent action and it requires manipulating a representation of what one intends to do. In a machine with no state and a hard-wired decision network, all possible goals and behaviors must be built in from the beginning. There is no place for deliberation in such a machine. While the RAP system itself does not deliberate either, it was designed to support the process (see Section 7.4).

Further comparison of the RAP system to Brooks robots and PENGI is not really appropriate because they currently act at very different levels. PENGI and the robots can only perform very simple behaviors, and the RAP system assumes a low-level robot controller that looks like a hard-wired machine. However, Brooks, Chapman and Agre argue that their approach is all that is necessary and therefore demand to be confronted. While their systems are very interesting and I agree wholeheartedly that complex behavior emerges from the interaction of simple processes with the world, representation obviously has its uses — hand-crafted hardware cannot be all there is. I believe the RAP system addresses important issues in the connection between symbolic computation and hard-wired behavior that they ignore. On the other hand, Brooks has real working robots and I don't — yet.

### 7.5.3 Virtual Machines

Another approach to building robot control systems suggested by Kaelbling [1986a, 1988] is very similar to the machines discussed above. The idea is to specify both the goals to be achieved by a robot, and the methods for their achievement in a language called GAPPS. The specification is then compiled into a language called REX which can be compiled into a virtual machine. The machine maps inputs from sensors into actions for effectors with a guaranteed response time [Rosenschein and Kaelbling, 1986] using no internal state. While a complete robot control system is hinted by Kaelbling [1986b], the GAPPS language deals only with goals that direct effectors — sensing goals are assumed to be implemented in some other, unspecified way. Furthermore, for a virtual machine description to make sense, the robot's goals

are assumed to be fixed in advance. Thus, this approach is very similar to that of Brooks and Chapman and Agre, except for the additional GAPPS description of the machine.

The RAP language appears to support all of the behaviors that can be encoded in the GAPPS language, but the properties of a GAPPS defined machine are hard to pin down because of two assumptions implicit in the work:

- Sensing will compile a complete and accurate model of the world without the need for sensing actions to be specified in any GAPPS operators.
- The sensing system will generate arbitrary plans and data structures (such as routes through the world) without the GAPPS machine needing to ask for them.

GAPPS machine descriptions often refer to specific items in the world, and make use of situation dependent routes and plans, without having to specify where the required sensory information comes from.

Kaelbling does not address any of the sensing issues raised during discussion of the RAP memory in Chapter 2 or discussion of RAP coding strategies in Chapter 5. Sensing issues have had a significant effect on shaping the RAP system so it is hard to compare it with systems that do not discuss sensing at all. Perhaps, as more of the GAPPS based execution system is presented in the literature, more useful comparisons will be possible.

Schoppers [1987a, 1987b] presents an approach to robot control in uncertain worlds that comes very close to the idea of building a virtual machine. He gives a procedure for constructing *universal plans* that can be used to achieve goals when autonomous processes are changing the world, or when the initial world state cannot be determined. A universal plan is a decision graph that maps the current world state into the best action to execute. When actions execute successfully they alter the world state and the new state initiates the next step toward the goal. In fact, when the world state changes for any reason, an appropriate action is initiated to get things moving back toward the goal. Thus, a universal plan is similar in concept to a machine designed specifically to achieve a fixed set of tasks. Schoppers does not address the problem of sensing the world state either.

### 7.5.4 The Procedural Reasoning System

The Procedural Reasoning System (PRS) [Georgeff *et al.*, 1986, Georgeff, 1988] requires special mention because much of it is very similar to the RAP system. The PRS system consists of four major pieces:

- A database of beliefs about the world and the state of the system itself.
- A library of plans called KAs.
- A graph of intentions.
- An interpreter.

A KA is a method for carrying out a goal, and an intention is a goal coupled with an instantiated KA for achieving it. An intention is somewhat similar to a RAP-defined task. The PRS interpreter proceeds by selecting the first intention from the intention graph and moving from one step in its KA to the next. When a step that requires invoking another KA is encountered, a new intention is created and added to the intention graph. If a step in a KA fails, the KA is abandoned and the interpreter tries to instantiate another KA for the intention. If no other KA is applicable, the intention fails. A given KA is tried only once for a given intention. Except for terminology and superficial internal structuring, execution of an intention is similar to execution of a task in the RAP system.

The significant difference between PRS and the RAP system is the way the algorithms are implemented. The RAP system has a fixed and well defined algorithm for the execution of a task. PRS uses META-KAs to control each step in the execution of an intention, allowing every KA to be processed in a different way. Consider two aspects of task execution: selecting a task to execute and determining when a task is complete. The RAP system schedules task execution according to the heuristics presented in Section 3.2.4. Those heuristics depend only on a task's priority, duration, and deadlines and not on the task's type or the type of other tasks on the agenda. The RAP system also uses a simple and well defined procedure for determining when a task is complete; the task's succeed clause must be satisfied.

The PRS system handles intention scheduling and completion quite differently. The PRS belief database holds, not just assertions about the state of the world, but

assertions about the system's current intentions as well. For instance, there are assertions that state which intentions are active, which intentions have finished running an instantiated KA, and which intentions have been satisfied. With this additional state available for examination, the PRS can support META-KAs that manipulate internal intentions, and they are used for both intention scheduling and for checking intention satisfaction. Whenever a new intention is generated as a step in an executing KA, its existence is asserted in the belief database and that activates META-KAs to determine where it should be placed in the current intention graph. These META-KAs may do an arbitrary computation before deciding where the new intention should go. Similarly, when a KA completes, an assertion to that effect is made in the belief database and META-KAs are activated to decide whether the KA has done what it was supposed to. These META-KAs check can also do arbitrary, context dependent computation.

Georgeff argues that META-KAs are a natural way to integrate planning and acting into the same architecture, and to some extent he is correct. However, so much of PRS's behavior is encoded as META-KAs that it is hard to separate the two. Intention scheduling and satisfaction depend on the existence of appropriate META-KAs and there is no commitment to an algorithm for these processes in the absence of applicable META-KAs. After removing the META-KAs from the system, PRS becomes a general purpose rule interpreter with no particular orientation towards robot task execution. The interesting aspects of the system are tied up in the META-KAs, but unfortunately, these are purposefully left to be specified differently for different tasks in different domains. While this leaves PRS as flexible as possible, it is hard to compare PRS with the RAP system, which makes stronger commitments.

For example, the RAP system commits itself to the notion of each task having a succeed clause that must be satisfied before the task can be considered complete. Such a commitment makes the RAP a useful planning operator because a planner can assume that the task will keep trying until the clause is satisfied. Furthermore, the information needed to characterize the RAP as a planning operator is explicitly represented in the RAP itself. In PRS, intentions are satisfied when they have first completed a KA, and then a META-KA, or set of META-KAs, have checked the results. Since KAs and META-KAs are all defined independently and can be further influenced by other META-KAs, it is hard to say what can be inferred about the world after an intention completes. The way that intentions and KAs interact with the world is

defined only through the META-KAs that control them.

As a final point, sensory data processing is only touched on briefly in papers about PRS and it, too, is apparently controlled by the invocation of KAs. Sensors are assumed to return either completely reliable predicates that can be asserted directly into the belief database, or they return unreliable information that must be processed. The processing is done via the initiation of KAs when messages are generated by the sensors, but few examples of such processing are given.

In summary, the PRS architecture is an interesting way to control rule invocation, and it looks very similar to the RAP system at that level. However, PRS makes no commitment to robot control issues beyond the claim that rule invocation should do the job. The behavior of a PRS controlled robot when confronted with uncertainty, errors, or interruptions depends almost entirely on the META-KA library and the contents of that library are assumed domain dependent. Thus, while PRS claims to be an architecture that supports both planning and execution, it does not commit itself to a methodology for either one.

### 7.5.5 Summary of Reactive Execution Work

Overall, current work on reactive execution systems breaks down into two basic areas: work on hard-wiring robot behaviors, and work on integrating planning and execution under a single architecture. Both areas of work are generating interesting results, but neither covers quite the same ground as the RAP system. The RAP system is at a more abstract level than the machine builders have yet reached and, while they claim that more abstract representation is not useful, the RAP system shows how it can be used to grapple with uncertainty and low-level failure. However, the RAP system must stretch downward to more realistic hardware and, as it does, there will be more and more connection with ideas from machine building.

Work on integrating planning and execution will also be important as the RAP system reaches upward to include a planning system. The planner and the RAP interpreter will have to work together and share information, but it is not clear that merging planning and action into a single homogeneous architecture is the correct route to follow. The RAP system is designed to fulfil two goals: to carry out tasks without supervision in a complex, uncertain world, and to provide a firm basis for

further planning research. The latter goal is supported by committing the RAP system to a simple, fixed control structure and a uniform action representation. These two features work together to give RAPs a well-defined semantics for planner's to reason about. If RAP representation and control were blurred together under a more general architecture, RAPs would lose their claim to being a representation for action, and only the system as a whole would be meaningful.

As a final comment, it is odd that reactive execution systems described in the literature seem so unconcerned with the problems of gathering information about the world. The RAP system makes a commitment to gathering sensory data only as a result of deliberate action and this has a profound influence on the way RAPs must be represented because it implies that information will always be missing and incorrect. Systems that assume complete and correct information about the world are cutting corners that cannot be cut.

## 7.6 Conclusions

Given the three-tiered robot control system discussed previously in this chapter, the RAP system can be seen as an interface between the processes of *action* and *deliberation*.[4] Real action necessarily requires multiple, concurrent behaviors tightly coupled with fast sensory feedback. In contrast, deliberation requires time to look into the future and consider the effects of alternate courses of action. A robot must react quickly to cope with changes in the world but it must think things through to avoid making stupid mistakes. One way to reconcile these conflicting requirements is to package simple reactions into larger, abstract actions. Deliberation using the abstract actions can be more efficient because there is less detail to worry about, yet response time in critical situations can be held down if deliberation can be bypassed when necessary. The RAP system is designed to package and execute actions in exactly this way.

The RAP system uses RAP definitions to describe abstract actions. At execution time, RAP-defined actions are instantiated into tasks and strong commitments are made with respect to task execution.

[4]The terminology and ideas in this discussion developed during many long and enjoyable discussions with Steve Hanks. He discusses the same ideas from a different angle in Hanks [1989].

- Action choices are based on the situation encountered. There is no looking into the future.
- Execution decisions are based on a memory of past sensory data because sensors cannot be aware of everything at the same time. The memory will inevitably contain incorrect and out-of-date information, and task definitions must take that fact into account.
- A task may require several attempts to carry out its packaged actions successfully. Uncertainty in the world situation may cause inappropriate initial method choices, and interference from other agents may cause appropriate methods to fail or be abandoned.
- The execution system must respond to situation changes by interrupting the pursuit of one task in favor of another. Opportunities and emergencies can only be dealt with effectively by allowing the system to change its focus of attention as the situation demands.

The experiments and examples in this thesis show that these commitments, as embodied in the RAP system, enable tasks to succeed even in complex, dynamic worlds.

By separating action and deliberation, while retaining the ability to cope with the local situation, the RAP system opens up new avenues of investigation. One can pursue methods of low-level robot control that involve cooperating, concurrent processes and use the RAP system to generate more structured behavior by switching between process sets. Two separate requirements for planning are also raised: new RAPs and RAP methods must be generated, and the RAP task agenda must be monitored to catch problems and fix inefficiencies that become evident only at execution time. It remains to be seen if a single planning paradigm can be applied to both problems or whether a collection of paradigms will be appropriate. However, in the meantime, the RAP system's reliance on situation-driven execution, rather than on planning, shows that projection of the future is not a requirement for action.

There is much work still to be done. The RAP system cannot be considered complete or proven until a whole three-tiered robot control system has been constructed. I am confident that the RAP notion of a primitive action can be implemented and that RAPs will make good planning operators, but proof requires demonstration. The

ability of the current implementation to control a simulated robot in a complex, dynamic world is an encouraging start. There is also work to be done characterizing and codifying sensing and coping strategies like those discussed in Chapter 5. I am less confident that this can be done — task descriptions appear to be very *ad-hoc* in structure and there is little reason to believe that sensing strategies will be any different. However, if coping and sensing strategies cannot be codified, the need for the RAP system only increases. When less structure can be placed on interaction with the world, it is harder to construct courses of action in advance and more important to pull out specific behaviors as the situation demands.

We used to believe one could not act without being able to plan, but it is now clear that one cannot plan without being able to act.

# Bibliography

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Sixth National Conference on Artificial Intelligence*, Seattle, WA, July 1987. AAAI.

[Allen and Bajcsy, 1985] Peter Allen and Ruzena Bajcsy. Object recognition using vision and touch. In *Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985. IJCAI.

[Arbib, 1981] M. Arbib. Perceptual structures and distributed motor control. In V. Brooks, editor, *Handbook of Physiology — The Nervous System II*. 1981.

[Arkin, 1987] Ronald C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *International Conference on Robotics and Automation*, Raleigh, NC, March 1987. IEEE.

[Arkin, 1988] Ronald C. Arkin. Neuroscience in motion: The application of schema theory to mobile robotics. In P. Evert and M. Arbib, editors, *Sensimotor Coordination in Amphibians: Experiments, Comparisons, Models and Robots*. Plenum Publishing Co., 1988.

[Birnbaum, 1986] Lawrence Birnbaum. Integrated processing in planning and understanding. Technical Report YALEU/CSD/RR #489, Computer Science Department, Yale University, December 1986.

[Bowen and Kang, 1988] James Bowen and Jianchu Kang. Conflict resolution in fuzzy forward chaining production systems. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Brooks, 1982] Rodney Brooks. Symbolic error analysis and robot planning. AI Memo 685, MIT, September 1982.

[Brooks, 1984] Rodney A. Brooks. Model-based three-dimensional interpretations of two-dimensional images. *IEEE PAMI*, 5(2), 1984.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[Brooks, 1987] Rodney Brooks. Intelligence without representation. In *Workshop on the Foundations of Artificial Intelligence*, MIT, June 1987.

[Brownston *et al.*, 1985] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison-Wesley, Reading, MA, 1985.

[Chapman and Agre, 1986] David Chapman and Philip E. Agre. Abstract reasoning as emergent from concrete activity. In *Workshop on Planning and Reasoning about Action*, Portland, Oregon, June 1986.

[Chapman, 1985] David Chapman. Planning for conjunctive goals. Technical Report TR 802, MIT Artificial Intelligence Laboratory, 1985.

[Charniak and McDermott, 1985] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company, 1985.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Dean and Kanazawa, 1988] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Dean *et al.*, 1987] Thomas Dean, R. James Firby, and David Miller. The forbin paper. Technical Report YALEU/CSD/RR #550, Computer Science Department, Yale University, July 1987.

[Dean *et al.*, 1988] Thomas Dean, R. James Firby, and David Miller. Hierarchical planning involving deadlines, travel time and resources. *Computational Intelligence*, 4(3), August 1988.

[Dean, 1985] Thomas Dean. Temporal imagery: An approach to reasoning about time for planning and problem solving. Technical Report YALEU/CSD/RR #433, Computer Science Department, Yale University, October 1985.

[Doyle *et al.*, 1986] Richard Doyle, David Atkinson, and Rajkumar Doshi. Generating perception requests and expectations to verify the execution of plans. In *Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986. AAAI.

[Fickas, 1985] Stephen F. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11), 1985.

[Fikes and Nilsson, 1971] R.E. Fikes and N.J. Nilsson. Strips: A new approach to the application of theorum proving to problem solving. *Artificial Intelligence*, 2, 1971.

[Firby and Hanks, 1987a] R. James Firby and Steve Hanks. A simulator for mobile robot planning. In *Knowledge-Based Planning Workshop*, Austin, TX, December 1987. DARPA.

[Firby and Hanks, 1987b] R. James Firby and Steve Hanks. The simulator manual. Technical Report YALEU/CSD/RR #563, Computer Science Department, Yale University, November 1987.

[Firby and McDermott, 1987] R. James Firby and Drew McDermott. Representing and solving temporal planning problems. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*. Springer Verlag, New York, NY, 1987.

[Fox, 1986] Mark S. Fox. Observations on the role of constraints in problem solving. In *Sixth Canadian Conference on Artificial Intelligence*, Montreal, Quebec, May 1986. CSCSI.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman Publishing, Los Altos, CA, 1987.

[Georgeff *et al.*, 1986] Michael P. Georgeff, Amy L. Lansky, and Marcel J. Schoppers. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Tech Note 380, AI Center, SRI International, 1986.

[Georgeff, 1988] Michael P. Georgeff. An embedded reasoning and planning system. Australian Artificial Intelligence Center, 1988.

[Gini *et al.*, 1985] Maria Gini, Rajkumar S. Doshi, Sharon Garber, Marc Gluch, Richard Smith, and Imran Zualkenian. Symbolic reasoning as a basis for automatic error recovery in robots. Technical Report TR 85-24, University of Minnesota, July 1985.

[Hammond, 1986] Kristian John Hammond. Case-based planning: An integrated theory of planning, learning and memory. Technical Report YALEU/CSD/RR #488, Computer Science Department, Yale University, October 1986.

[Hanks and McDermott, 1987] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33, 1987.

[Hanks, 1988] Steve Hanks. Representing and computing temporally scoped beliefs. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Hanks, 1989] Steven Hanks. *Heuristic Plan Projection and Evaluation.* PhD thesis, Computer Science Department, Yale University, 1989.

[Hayes-Roth and Hayes-Roth, 1979] B. Hayes-Roth and F. Hayes-Roth. A cognitive model of planning. *Cognitive Science*, 3, 1979.

[Hendler and Sanborn, 1987] James A. Hendler and James C. Sanborn. A model of reaction for planning in dynamic environments. In *Knowledge-Based Planning Workshop*, Austin, TX, December 1987. DARPA.

[Hogge, 1988] John C. Hogge. Prevention techniques for a temporal planner. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Horswill and Brooks, 1988] Ian Douglas Horswill and Rodney Allen Brooks. Situated vision in a dynamic world: Chasing objects. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Joslin *et al.*, 1986] David E. Joslin, David E., and John W. Roach. An analysis of conjunctive-goal planning. Technical Report TR 86-34, Virginia Tech, 1986.

[Kaelbling, 1986a] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In *Workshop on Reasoning about Actions and Plans*, Timberline, Oregon, June 1986.

[Kaelbling, 1986b] Leslie Pack Kaelbling. Rex programmer's manual. Technical Note 381, SRI International, May 1986.

[Kaelbling, 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Lifschitz, 1986] Vladimir Lifschitz. Pointwise circumscription. In *Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986. AAAI.

[Linden, 1987] Theodore A. Linden. Transformational synthesis applied to alv mission planning. In *Knowledge-Based Planning Workshop*, Austin, TX, December 1987. DARPA.

[Lowe, 1987] David G. Lowe. Three-dimensional object recognition from single two-dimensional images. *Artificial Intelligence*, 31(3), March 1987.

[McCalla *et al.*, 1982] G.I. McCalla, L. Reid, and P.F. Schneider. Plan creation, plan execution and knowledge acquisition in a dynamic microworld. *International Journal of Man-Machine Studies*, 16, 1982.

[McCarthy and Hayes, 1969] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Richie, editors, *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, UK, 1969.

[McCarthy, 1981] John McCarthy. Circumscription — a form of non-monotonic reasoning. In Bonnie Lynn Webber and Nils J. Nilsson, editors, *Readings in Artificial Intelligence*. Tioga Publishing Co., Palo Alto, CA, 1981.

[McDermott and Doyle, 1980] Drew V. McDermott and Jon Doyle. Non-monotonic logic i. *Artificial Intelligence*, 13, 1980.

[McDermott and Forgy, 1978] J. McDermott and C. Forgy. Production system conflict resolution strategies. In D.A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*. Academic Press, 1978.

[McDermott, 1977] Drew V. McDermott. Flexibility and efficiency in a computer program for designing circuits. Technical Report 402, MIT AI Laboratory, 1977.

[McDermott, 1978] Drew V. McDermott. Planning and acting. *Cognitive Science*, 2, 1978.

[McDermott, 1980] Drew V. McDermott. Contexts and data dependencies: A synthesis. *IEEE Transactions on Pattern Recognition and Machine Intelligence*, PAMI-5(3), 1980.

[McDermott, 1982] Drew V. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6, 1982.

[McDermott, 1985] Drew McDermott. Reasoning about plans. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*. Ablex Publishing Corporation, 1985.

[Miller *et al.*, 1960] G. Miller, E. Galanter, and K. Pribram. *Plans and the Structure of Behavior*. Holt, Rinehart and Winston, New York, NY, 1960.

[Miller, 1985] David P. Miller. Planning by search through simulations. Technical Report YALEU/CSD/RR 423, Yale University Department of Computer Science, 1985.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto, CA, 1980.

[Payton, 1986] D.W. Payton. An architecture for reflexive autonomous vehicle control. In *International Conference on Robotics and Automation*, San Francisco, CA, 1986. IEEE.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13, 1980.

[Rosenschein and Kaelbling, 1986] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Conference on Theoretical Aspects of Reasoning about Knowledge*, Asilomar, CA, 1986.

[Sacerdoti, 1975] Earl D. Sacerdoti. A structure for plans and behavior. Technical Note 109, Stanford Research Institute, 1975.

[Schank and Abelson, 1977] Roger Schank and Robert Abelson. *Scripts Plans Goals and Understanding.* Lawrence Erlbaum Associates, Hillside, NJ, 1977.

[Schoppers, 1987a] M.J. Schoppers. Composing and decomposing universal plans. Research note from Advanced Decision Systems, 1987.

[Schoppers, 1987b] M.J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987. IJCAI.

[Shoham, 1986] Yoav Shoham. Reasoning about change: Time and causation from the standpoint of artificial intelligence. Technical Report YALEU/CSD/RR #507, Computer Science Department, Yale University, December 1986.

[Simmons, 1988] Reid G. Simmons. A theory of debugging plans and interpretations. In *Seventh National Conference on Artificial Intelligence*, St. Paul, MN, August 1988. AAAI.

[Slack and Miller, 1987] Marc Slack and David Miller. Planning through time and space in dynamic domains. In *Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987. IJCAI.

[Suchman, 1987] Lucy A. Suchman. *Plans and Situated Actions: The problem of human/machine communication.* Cambridge University Press, 1987.

[Sussman, 1975] Gerald J. Sussman. *A Computer Model of Skill Aquisition.* American Elsevier, New York, NY, 1975.

[Tate, 1977] Austin Tate. Generating project networks. In *Fifth International Joint Conference on Artificial Intelligence.* IJCAI, 1977.

[Ullman, 1984] Shimon Ullman. Visual routines. *Cognition*, 18, 1984.

[Vere, 1983] Steven Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5, 1983.

[Vere, 1985] Steven Vere. Temporal scope of assertions and window cutoff. In *Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985. IJCAI.

[Wilensky, 1983] Robert Wilensky. *Planning and Understanding: A Computational Approach to Human Reasoning.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[Wilkins, 1985] David E. Wilkins. Recovering from execution errors in sipe. *Computational Intelligence*, 1(1), February 1985.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

# Appendix A

# RAP Memory Functions

This appendix describes the functions used to interface with the RAP memory. These functions are used by the hardware interface to update the memory to reflect changes in the world. None of the functions appear explicitly in RAP's themselves, and only the query function is actually called by the RAP interpreter.

This information is included so that those interested can study the RAP memory, and the way it interacts with the simulated delivery truck hardware, more closely.

## A.1 Declarations

Before the RAP memory is used, all assertion types to be used in memory must be declared. Plain propositions are declared with the form:

`(declare-memory-proposition` *name*`)`

This declares assertions that start with *name* to be legal in memory. Plain propositions are not treated as fluents (see below) and, once asserted, are only removed by explicitly erasing them.

Item property assertions are declared with the form:

`(declare-memory-property` *name default-function*`)`

which says that assertions beginning with *name* are to be treated as item properties; meaning they will be treated as fluents (see Section 2.2). The default function is used to generate a value for the property at query time if no assertion for it exists. A default function takes two arguments: the memory structure begin queried, and the index of the property needing the default. The default function used most often is:

```
(defun unknown-default ( memory property-index )
   (ignore memory property-index)
   'unknown)
```

but any function can be used, including functions that make further memory queries. Default functions should not alter memory.

User functions can also be declared for use in memory queries using the form:

`(declare-memory-function` *name function*`)`

Within queries, forms that begin with *name* become function calls to *function*.

## A.2 Memory Structures

The RAP memory system is designed to support multiple instances of memory structures. Thus, except for global declarations, all memory access functions take a memory structure as an argument. Memory structures are created using the function:

`(create-memory-structure` *name*`)` `=>` *memory-structure*

where *name* is a symbol used for identification purposes only. Memory structures are a bookkeeping device for recording which assertions belong to which user.

## A.3 Assertions

Assertions are made to the RAP memory database using the form:

`(memory-assert` *memory proposition*`)`

where *memory* is the memory structure to which the assertion is being made, and *proposition* is the assertion. Only a single proposition declared via `declare-memory-proposition` or `declare-memory-property` can be asserted at one time. The assertion must not contain any variables.

Assertions can be erased from the RAP memory database with:

`(memory-erase` *memory clause*`)`

where *memory* is the structure from which the assertions are to be erased, and *clause* is an assertion prefix. Every assertion in the database that has the same prefix as the clause argument is erased. For example, `(memory-erase mem '(color item-23 red))` erases the single assertion that `item-23` is red, while `(memory-erase mem '(color))` erases every assertion about color in the database. This function is seldom used by the robot hardware because most item properties are single valued (*i.e.*, fluents whose old values are overwritten automatically).

## A.4 Queries

Queries to RAP memory are made using the form:

`(memory-query` *memory binding-environment query*`)` `=>` *boolean*

where *memory* is the memory structure being checked, *binding-environment* is a structure to record free variable bindings, and *query* is the query to be checked in the database.

Binding environments are created using the function:

`(create-binding-environment)` `=>` *binding-environment*

which takes no arguments and returns an empty structure. This structure can then be used in several ways. First, it can be used in `memory-query` calls. If a free variable in the query is bound during the call, the binding is recorded in the environment. Later, if the environment is used with another query, bindings it holds will stay in effect over the query. For example, suppose the query

```
(memory-query memory-1 env-1 '(color item-23 ?color))
```

is made and, in the process of satisfying it, `?color` is bound to `red`; the environment `env-1` will hold that binding. If the following query is then made:

```
(memory-query memory-1 env-1 '(color item-43 ?color))
```

it will only succeed if `item-43` is red because `?color` is already bound in the environment.

A binding environment can also be asked for the current value of the bindings it contains. The form:

`(binding-value` *binding-environment variable*`)` `=>` *value*

is used for this purpose and returns the current value bound to *variable*. For the example above,

```
(binding-value env-1 '?color)
```

would return `'red`.

## The Query Language

The queries that appear in `memory-query` are constructed using the RAP query language. This language consists of declared propositions and functions, predefined functions, the connectives `and`, `or` and `not`, and the special form `believe`. A base query consists of either a single proposition or a single query function. Examples of base queries are:

```
'(color item-34 red)
'(color ?item red)
'(= ?color red)
```

The following functions are predefined:

```
< > <= >= =
```

The first four of these are defined on numbers only, and the last one is defined on symbols and strings as well.

Base queries are tied together using `and`, `or` and `not` in the obvious way. These connectives act like their boolean counterparts but use query success and failure rather than true and false as their arguments. In a query, `and` succeeds if each of its constituents succeeds, `or` succeeds if any one of its constituents succeeds, and `not` succeeds if its constituent query fails. The constituents of a connective can be any valid query, including another connective. For example:

```
'(and (color item-34 red)
      (size item-34 big)
      (contains item-34 item-56 true))
'(or (color item-34 red)
     (color item-34 blue))
'(and (color item-34 red)
      (not (size item-34 big)))
```

The final query language construct is `believe`. The basic form of believe is:

`(believe` *proposition time-interval*`)`

where the *proposition* is a single, non-nested, user-declared proposition and *time-interval* is an interval of time. The semantics of the construct is different when the time interval takes on different forms. If the time interval is bound, either by being specified as an integer or by being a variable bound to an integer, the `believe` succeeds if an assertion matching the proposition has been made within the specified amount of time. For example:

```
'(believe (color item-34 red) 45)
'(believe (color item-34 red) ?time)
```

will behave exactly the same assuming `?time` is bound to 45. It will succeed if `item-34`'s color has been asserted as red within the last 45 time units. The query will fail if a contradicting assertion has been made within the time interval. If no assertion has been made within the interval, and the default value for the proposition matches the proposition, then the query will succeed.

If the time interval in a believe form is an unbound variable, then the form behaves slightly differently — the form becomes a query asking when a matching assertion was made. If an assertion matching the proposition exists in the database, the `believe` will succeed with the time interval variable bound to the time since the assertion was made. For example, if the color of `item-34` was asserted to be blue 15 time units ago, the query:

```
'(believe (color item-34 ?color) ?time)
```

will succeed with `?color` bound to `'blue` and `?time` bound to 15.

### Query Processing

When a query is checked against the database, it is processed from the first clause to the last with variables being bound along the way. This means that a little bit of care is required to make sure variables are bound where they are supposed to be. For example, the query:

```
'(and (believe (capacity item-34 ?size))
      (< ?size 10))
```

will succeed if `item-34` has capacity less than 10, but the similar query:

```
'(and (< ?size 10)
      (believe (capacity item-34 ?size)))
```

will not work because `?size` will not yet be bound when `<` is called.

## A.5 Description Matching

Description matching is done automatically in the memory system (see Section 2.3), but the memory must be notified when new item descriptions are introduced, and when they change enough so that matching should be tried. The functions in this section are used at the interface between the robot hardware and the RAP memory to give the memory those notifications. All of these functions are specialized to the domain used for the examples in the thesis. In particular, expectation sets are assumed to be indexed by item CLASS and LOCATION, both of which are immediately accessible from the sensors.

## Declaring Comparator Functions

The memory does not use a fixed algorithm for comparing item descriptions because context can often make a big difference on how descriptions should be merged. Therefore, the comparator functions used by the memory are declared in advance using the forms:

`(declare-indistinct-class` *class default-comparator*`)`
`(declare-class-comparator` *class location comparator*`)`

The first of these declares a default comparator function to be used for items of type *class* and the second declares a more specific function to be used for items of a given *class* at a particular *location.*

A comparator function takes three arguments: the memory structure being used, and the two items to be compared. Each item is an internally defined data-structure with the following three access functions:

`(memory-item-class` *item*`)` `=>` *item class*
`(memory-item-location` *item*`)` `=>` *item location*
`(memory-item-properties` *item*`)` `=>` *list of item properties*

The first returns the class of the *item*, the second returns the item's location, and the third returns all properties that have been asserted for the item. An item property looks just like a memory assertion and only properties defined with `declare-memory-property` are returned. The comparator function should generate an integer value scoring the match of the two items it is given. A higher score means a better match.

The comparator function used most often in the robot deliver truck domain is:

```
(defun default-item-comparator ( memory item1 item2 )
   (ignore memory)
   (let ( (props1 (memory-item-properties item1))
          (props2 (memory-item-properties item2)) )
      (loop for ( (prop1 in props1)
                  (score 0) )
            result score
            (let ( (prop2 (car (member= equal-item-properties
```

```
                                prop1 props2))) )
    (if (and (not (null prop2))
             (not (or (eq (lastelt prop1) 'unknown)
                      (eq (lastelt prop2) 'unknown))))
        (setf score (fx+ score 10)))))))
```

This function scores 10 for every property that both items have in common as long as the value of the property is not unknown.

## Manipulating Items in Expectation Sets

When a new item is detected and assigned a new sensor name, the memory must be told using the function:

(`introduce-new-item` *memory sensor-name location class*)

The *memory* argument is the memory structure for the item to be stored in, *sensor-name* is the name assigned to this object by the sensors, and *class* and *location* are used to specify the expectation set into which this item should be placed. This function informs the memory that a new item has become accessible and the memory then assigns it a memory name, puts it into the local model and associates it with the appropriate expectation set.

Two other functions are used to give the memory information so that item matching can proceed correctly. The first tells the memory that an item has been moved from one expectation set to another, and the second tells the memory that all accessible items in a given expectation set have been detected. These functions take the form:

(`new-item-location` *memory sensor-name location*)
(`expectation-set-fix-cardinality` *memory location*)

The arguments have the obvious meaning, with *class* and *location* being used to reference the expectation set being effected.

Finally, there is a function to let the memory know that an expectation set has become inaccessible and a new layer should be pushed. For the simulated world, moving from one location to another causes this to happen to every expectation set, so there is a single call to expire all active expectation sets at once. The call is:

`(expectation-set-expire` *memory*`)`

which instructs the memory that all expectation sets have become inaccessible.

## Delaying Item Matching

In many situations, the sensors will return several messages about a given item at one time. In such cases, it is often appropriate to make all property assertions before item matching takes place. Therefore, item matching can be delayed temporarily with the form:

`(with-delayed-item-matching .` *body*`)`

No change in memory made within the scope of this form will result in item matching until the form is exited.

# Appendix B

# The Robot Delivery Truck Domain

This appendix is included to give those who are interested a more detailed picture of the simulated delivery truck domain. The specific domain outlined below is implemented using an extensible simulator construction tool described in Firby and Hanks [1987b]. Basically, the simulation consists of a delivery truck that moves around in a graph, where the graph nodes are locations and the graph edges are roads. As the truck moves from one location to another, it will encounter and interact with various items. The behavior of items in the world can be varied extensively and this appendix discusses only those that were used for the experiments in Chapter 6. This appendix does not give enough information to recreate the experiments exactly, but does give a feeling for the way the domain works. Chapter 6 and Section 1.4.2 also discuss aspects of the delivery truck domain.

## B.1 The Delivery Truck

The robot delivery truck is the executor's access to the simulated world. The truck responds to external commands and returns sensory data about its surroundings. It can move from one location to another and has several cargo bays, loading arms and a vision-like sensor. The loading arms are used to manipulate items near the truck and items can be carried from one location to another in the cargo bays. The sensor is used to determine what items are at the current location and when new items arrive. It can also examine particular items in detail. For example, the truck might arrive at

a new location, scan the area with its sensor and discover a fuel container, pick the container up and examine it to determine how much fuel is inside.

The deilvery truck has two arms: `arm1` with capacity 8 and dropping frequency of 15%, and `arm2` with capacity 15 and dropping frequency of 50%. The truck also has two cargo bays: `bay1` with capacity 20 and `bay2` with capacity 100. There is also a weapon bay, in which a weapon must be mounted before it can be used, and a fuel bay, into which fuel is poured to refuel the truck. The primitive commands used to control the delivery truck are described below.

## Arm Commands

Commands to control the truck arms are the most frequently used in the domain. These commands often refer to a *place* which is either an item, an arm, a bay, or `external` which means outside of the the truck. Arm commands may fail as described.

(`arm-move` *arm place*): Move *arm* to *place*. This command can fail with the message `ARM-CANT-FIND` which means that the requested place is inaccessible.

(`arm-grasp` *arm item*): If the *arm* is at the *item*, pick it up. This command can fail with messages `ARM-TOO-FULL`, `ARM-NOT-AT`, `ARM-DROPPED`, `ARM-CANT-GRASP`, or `ARM-INTERFERENCE`. The last of these occurs when both arms are at the item to be grasped and the second last means that a tool is required to grasp the item.

(`arm-ungrasp` *arm item*): If the *arm* is holding the *item*, put it down. This command can fail with messages `ARM-NOT-HOLDING` or `CONTAINER-FULL`. The latter means that the place currently holding the arm is too full to hold the item to be put down.

(`arm-pour` *arm vessel*): Pour the contents of *vessel* into the item that the *arm* is at. This command is used for transfering liquids between containers. The only possible failure is `ARM-NOT-HOLDING`.

(`arm-ladle` *arm vessel*): Fill the *vessel* from the item the *arm* is at. This command also moves liquids around. Possible failures are: `ARM-NOT-HOLDING` and `ARM-NOT-AT`. The latter occurs if the arm is not positioned at a vessel holding a liquid.

(`arm-toggle` *arm item*): If the *arm* is at *item*, turn it on or off. This command is used to make active items act (*i.e.*, the weapon is fired by toggling it). This command can fail with messages `ARM-NOT-AT` or `ARM-CANT-TOGGLE`. The second of these occurs when the item is broken or is not active.

The arm commands are used to manipulate items in the world. The `arm-move` command is used to move one of the loading arms from one item to another to move into or out of cargo bays and containers. The `arm-grasp` and `arm-ungrasp` commands are used to pick up items near an arm or to put down items that an arm is holding. To move a fuel drum from outside the truck into a cargo bay the following commands might be issued:

```
(ARM-MOVE    arm1 external)    ; Move arm outside the truck
(ARM-MOVE    arm1 fuel-drum1)  ; Move arm to fuel drum
(ARM-GRASP   arm1 fuel-drum1)  ; Pick the fuel drum up
(ARM-MOVE    arm1 bay1)        ; Move arm to cargo bay
(ARM-UNGRASP arm1 fuel-drum1)  ; Put fuel drum down in cargo bay
```

The `arm-pour` and `arm-ladle` commands are used to move liquids from one container to another and the `arm-toggle` command is used to start and stop items that have specific functions. For example, one way to create ice cubes might be:

```
(ARM-GRASP  arm1 pitcher1)      ; Pickup a pitcher
(ARM-MOVE   arm1 water-tank1)   ; Move arm to the water tank
(ARM-LADLE  arm1 pitcher1)      ; Fill the pitcher from the tank
(ARM-MOVE   arm1 ice-machine1)  ; Move the arm to the ice machine
(ARM-POUR   arm1 pitcher1)      ; Fill the ice machine from the pitcher
(ARM-TOGGLE arm1 ice-machine1)  ; Turn the ice machine on
```

## Movement Commands

The primitive commands to move the truck around are:

(`truck-turn` *direction*): Turn the truck to face in *direction*.

(`truck-speed` *speed*): Set the truck speed to *speed*.

(`truck-move`): If facing a road, travel down it. This command can fail with the message `TRUCK-MISHAP` which means that something untoward has occured and also means the end of the simulation for the experiments used in Chapter 6.

The `truck-turn` command is used to point the truck in a particular direction. There are eight directions corresponding to the eight major compass points and each location may have a road leading off in any or all directions. The `truck-move` command is used to send the truck off down a road. The truck will stop when it reaches the end of the road if everything goes fine or somewhere along the road if some kind of mishap occurs. On some roads the truck might roll over if it is going too fast, it might get stuck in the mud if it is raining, it might run out of gas or it might encounter enemy troops.

## Sensory Commands

The truck sensor is called the eye and there are four commands to control it:

`(eye-examine` *item*`)`: Get some sensory data about *item* if it is accessible. This command can fail with the message `EYE-CANT-FIND`.

`(eye-scan` *place*`)`: Scan *place* and return accessible items. This commands generates only `object-seen` and `road-seen` sensor messages.

`(eye-monitor` *class*`)`: Notice if an item of type *class* becomes accessible

`(eye-unmonitor` *class*`)`: Stop looking for items of type *class*

The `eye-scan` command returns all of the items at a given location; either immediately outside the truck or inside one of its cargo bays. The `eye-examine` command looks closely at a single item and reports back detailed information about it: how much and what type of liquid a container holds, how many bullets are in a gun, whether a machine is running or idle and so forth. The `eye-monitor` command is used to watch for a particular class of item to arrive at the current location. After every command, if the sensor was not used for anything else, it will notice whether an item of that class appears. The truck will also stop while traversing a road if it encounters an item that the sensor is scanning for. As an example, to travel down a road looking for a fuel drum the following commands might be issued:

```
(EYE-MONITOR fuel-drum)  ; Start looking for fuel-drums
(TRUCK-TURN  north)      ; Turn towards the north road
(TRUCK-SPEED fast)       ; Set truck speed to fast
(TRUCK-MOVE  fast)       ; Set off down the road
```

After executing this set of commands the truck will be either at a location along the road with one or more fuel drums present or at the end of the road if no fuel drums were seen.

## B.2 Items in the World

There are two aspects to items in the world: they can be manipulated by the truck arms, and they can be examined by the truck sensors. Item manipulations are described throughout the main text, so this section discusses only item sensing. When an item is sensed, sensory messages are generated. These messages contain information to be used by the hardware interface to update the RAP memory. The two sensing commands are `eye-scan` and `eye-examine`. The eye-scan command generates an `object-seen` and `road-seen` messages and the eye-examine command produces almost all other messages.

### Item Characteristics

The items used during the experiments are:

`ammo-box`: A box of class `ammo-box` and `vessel`, size 7 and capacity 15. Normally contains 15 units of `ammo` which is treated as a liquid in our simulation. Examination produces `vessel` and `capacity` sensor messages.

`fuel-drum`: A drum of class `fuel-drum` and `vessel`, size 5 and capacity 5. Normally contains 5 units of the liquid `fuel`. Examination produces `vessel` and `capacity` messages.

`rock`: A simple item of class `rock` and size 7. Examination produces `color` messages.

`weapon`: A complex item of class `gun` and `weapon` and size 10. Has the capacity to hold 15 units of `ammo` and when arm-toggled, fires up to three shots at local enemy-units. Each shot consumes one unit of ammo and has a 60% chance of making an enemy-unit go away. A weapon an be reloaded by pouring in new ammo. Examination produces `shots-left` messages.

`gun-tool`: A simple item of class `gun-tool` and size 4. Must be being held before a weapon can be grasped in a truck arm.

`enemy-unit`: An item of class `enemy-unit`. Cannot be manipulated — only shot at.

## Sensory Information about Items

The various sensory messages that might be produced by the above items are:

(`object-seen` *location sensor-name class*): Generated by eye-scan primitive actions. States that an item with the given sensor name and class is currently accessible at the given location. The location corresponds to the location being scanned.

(`vessel` *sensor-name liquid-type amount*): States that an item holds an amount of a liquid.

(`shots-left` *sensor-name number*): States that the gun with the given sensor name holds a certain amount of ammunition.

(`capacity` *sensor-name size*): States that an item can hold the given amount of liquid.

(`color` *sensor-name color*): States that an item has the given color.

(`road-seen` *direction road-name*): Generated by eye-scan primitive actions. States that there is a road with the given name in a particular direction.

(`new-location` *location*): Generated by truck-move primitive actions. States that the truck has arrived at the given location.

These messages are the only means of getting information about the world other than those features of the truck that are directly accessible. The directly accessible truck features are: speed, direction, fuel level, and the current time.

## B.3 The Memory Interface

This section discusses interfacing the simulator to the RAP memory. As described in Section 3.3, there is a hardware interface to update the RAP memory in response to information from the robot sensors and effectors. That section describes the way that the interface is constructed and gives several examples of the functions that are used to process information from primitive actions (see Figures 3.14, 3.15 and 3.16). The information available to be processed consists of the sensor and primitive action failure messages described above. This section briefly outlines the contents of RAP memory that are altered in response to those messages.

The three types of item that are indistinct and cannot be recognized from encounter to encounter are fuel-drums, ammo-boxes and rocks. To match descriptions of these items within expectation sets, the default item comparitor function discussed in Appendix A is used. Thus, these item types are declared as:

```
(declare-indistinct-class 'fuel-drum #'default-item-comparitor)
(declare-indistinct-class 'ammo-box  #'default-item-comparitor)
(declare-indistinct-class 'rock      #'default-item-comparitor)
```

The memory system declarations for the item properties kept track of are listed below. The default function for each is also given.

```
(declare-memory-property 'class         #'unknown-default)
(declare-memory-property 'holding       #'unknown-default)
(declare-memory-property 'location      #'unknown-default)
(declare-memory-property 'capacity      #'unknown-default)
(declare-memory-property 'color         #'unknown-default)
(declare-memory-property 'liquid-held   #'unknown-default)
(declare-memory-property 'quantity-held #'unknown-default)
(declare-memory-property 'tool-needed   #'false-default)
```

The `unknown-default` function is described in Appendix A. The `false-default` is almost the same:

```
(defun unknown-default ( memory property-index )
   (ignore memory property-index)
   'false)
```

In addition to item properties, the RAP memory keeps track of various features of the delivery truck itself. These features are declared as follows:

```
(declare-memory-proposition 'arm-place)
(declare-memory-proposition 'arm-at)
(declare-memory-proposition 'arm-holding)
(declare-memory-proposition 'arm-interference)

(declare-memory-proposition 'too-small-for)
(declare-memory-proposition 'too-full-for)

(declare-memory-proposition 'direction)

(declare-memory-proposition 'eye-examined)
(declare-memory-proposition 'eye-scanned)
(declare-memory-proposition 'arm-examined)
(declare-memory-proposition 'examined-since-move)
(declare-memory-proposition 'scanned-since-move)
(declare-memory-proposition 'examined-since-toggle)
(declare-memory-proposition 'examined-since-pour)
(declare-memory-proposition 'monitoring)

(declare-memory-proposition 'truck-location)
(declare-memory-proposition 'truck-heading)
(declare-memory-proposition 'truck-fuel)
(declare-memory-proposition 'truck-speed)
(declare-memory-proposition 'truck-status)
(declare-memory-proposition 'truck-time)
(declare-memory-proposition 'time-arrived)
```

The hardware interface translates between sensory messages and action results and the memory propositions defined above. RAPs only make queries that refer to memory propositions and not the sensory messages.