11 PLANNING AND ACTING IN THE REAL WORLD

In which we see how more expressive representations and more interactive agent architectures lead to planners that are useful in the real world.

The previous chapter introduced the most basic concepts, representations, and algorithms for planning. Planners that are are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This chapter shows how. Section 11.1 extends the classical language for planning to talk about actions with durations and resource constraints. Section 11.2 describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Section 11.3 presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems. Section 11.4 shows how to plan when the environment contains other agents.

11.1 TIME, SCHEDULES, AND RESOURCES

The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is "plan first, schedule later": that is, we divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

$$
\begin{aligned}
&\mathit{Jobs}(\{\mathit{AddEngine1} \prec \mathit{AddWheels1} \prec \mathit{Inspect1}\},\\
&\qquad \{\mathit{AddEngine2} \prec \mathit{AddWheels2} \prec \mathit{Inspect2}\})\\
&\mathit{Resources}(\mathit{EngineHoists}(1),\ \mathit{WheelStations}(1), \mathit{Inspectors}(2), \mathit{LugNuts}(500))\\
&\mathit{Action}(\mathit{AddEngine1}, \text{DURATION}{:}30,\\
&\qquad \text{USE}{:}\mathit{EngineHoists}(1))\\
&\mathit{Action}(\mathit{AddEngine2}, \text{DURATION}{:}60,\\
&\qquad \text{USE}{:}\mathit{EngineHoists}(1))\\
&\mathit{Action}(\mathit{AddWheels1}, \text{DURATION}{:}30,\\
&\qquad \text{CONSUME}{:}\mathit{LugNuts}(20), \text{USE}{:}\ \mathit{WheelStations}(1))\\
&\mathit{Action}(\mathit{AddWheels2}, \text{DURATION}{:}15,\\
&\qquad \text{CONSUME}{:}\mathit{LugNuts}(20), \text{USE}{:}\ \mathit{WheelStations}(1))\\
&\mathit{Action}(\mathit{Inspect}_i, \text{DURATION}{:}10,\\
&\qquad \text{USE}{:}\mathit{Inspectors}(1))
\end{aligned}
$$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

11.1.1 Representing temporal and resource constraints

A typical **job-shop scheduling problem**, as first introduced in Section 6.1.2, consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. Each constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be *produced* by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

JOB

DURATION

CONSUMABLE

REUSABLE

MAKESPAN

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form $[\mathit{AddEngine}, \mathit{AddWheels}, \mathit{Inspect}]$. Then the

Resources statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are "borrowed" at the start of an action and released at the action's end.

AGGREGATION

The representation of resources as numerical quantities, such as $Inspectors(2)$, rather than as named entities, such as $Inspector(I_1)$ and $Inspector(I_2)$, is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent $Inspect$ actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all $10!$ ways of assigning inspectors to actions.

11.1.2 Solving scheduling problems

CRITICAL PATH METHOD

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with $Start$ and ending with $Finish$. (For example, there are two paths in the partial-order plan in Figure 11.2.)

CRITICAL PATH

SLACK

SCHEDULE

The **critical path** is that path whose total duration is longest; the path is "critical" because it determines the duration of the entire plan—shortening other paths doesn't shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity $LS - ES$ is known as the **slack** of an action. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the ES and LS times for all the actions constitute a **schedule** for the problem.

The following formulas serve as a definition for ES and LS and also as the outline of a dynamic-programming algorithm to compute them. A and B are actions, and $A \prec B$ means that A comes before B:

$$
\begin{aligned}
&ES(Start) = 0 \\
&ES(B) = \max_{A \prec B} ES(A) + Duration(A) \\
&LS(Finish) = ES(Finish) \\
&LS(A) = \min_{B \succ A} LS(B) - Duration(A)\ .
\end{aligned}
$$

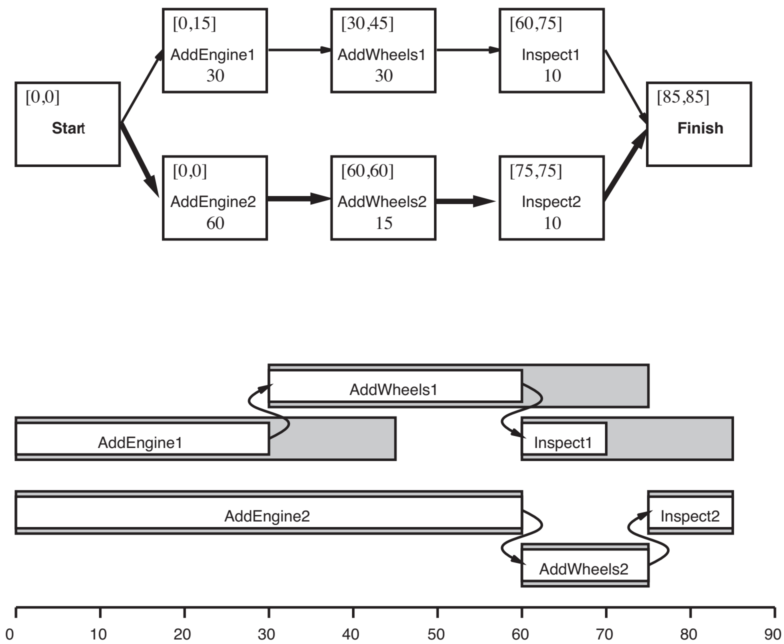

Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

The idea is that we start by assigning $ES(Start)$ to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the $Finish$ action.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the $AddEngine$ actions, which begin at the same time in Figure 11.2,

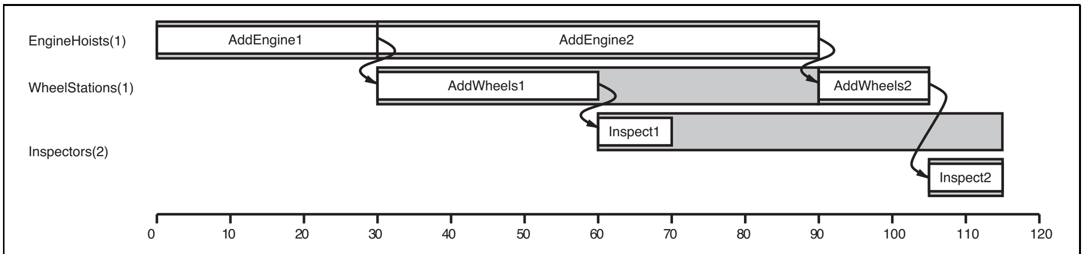

Figure 11.3 A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

require the same $EngineHoist$ and so cannot overlap. The "cannot overlap" constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.*, 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Chapters 3 and 4. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the ES and LS times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130–minute solution, not the 115–minute solution of Figure 11.3.

MINIMUM SLACK

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.

11.2 HIERARCHICAL PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

For plans executed by the human brain, atomic actions are muscle activations. In very round numbers, we have about 10^3 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 10^9 seconds in all. Thus, a human life contains about 10^{13} actions, give or take one or two orders of magnitude. Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 10^{10} actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be "Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home." Given such a plan, the action "Go to San Francisco airport" can be viewed as a planning task in itself, with a solution such as "Drive to the long-term parking lot; park; take the shuttle to the terminal." Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequences.

HIERARCHICAL DECOMPOSITION

In this example, we see that planning can occur both before and during the execution of the plan; for example, one would probably defer the problem of planning a route from a parking spot in long-term parking to the shuttle bus stop until a particular parking spot has been found during execution. Thus, that particular action will remain at an abstract level prior to the execution phase. We defer discussion of this topic until Section 11.3. Here, we concentrate on the aspect of **hierarchical decomposition**, an idea that pervades almost all attempts to manage complexity. For example, complex software is created from a hierarchy of subroutines or object classes; armies operate as a hierarchy of units; governments and corporations have hierarchies of departments, subsidiaries, and branch offices. The key benefit of hierarchical structure is that, at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a *small* number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small. Nonhierarchical methods, on the other hand, reduce a task to a *large* number of individual actions; for large-scale problems, this is completely impractical.

11.2.1 High-level actions

HIERARCHICAL TASK NETWORK

PRIMITIVE ACTION

HIGH-LEVEL ACTION

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. As in classical planning (Chapter 10), we assume full observability and determinism and the availability of a set of actions, now called **primitive actions**, with standard precondition–effect schemas. The key additional concept is the **high-level action** or HLA—for example, the action "Go to San Francisco airport" in the

$Refinement(Go(Home, SFO),$
  STEPS: $[Drive(Home, SFOLongTermParking),$
    $Shuttle(SFOLongTermParking, SFO)]\ )$
$Refinement(Go(Home, SFO),$
  STEPS: $[Taxi(Home, SFO)]\ )$

$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $a = x \land b = y$
  STEPS: $[\,]\ )$
$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $Connected([a, b], [a - 1, b])$
  STEPS: $[Left, Navigate([a - 1, b], [x, y])]\ )$
$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $Connected([a, b], [a + 1, b])$
  STEPS: $[Right, Navigate([a + 1, b], [x, y])]\ )$
...

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

REFINEMENT

example given earlier. Each HLA has one or more possible **refinements**, into a sequence[1] of actions, each of which may be an HLA or a primitive action (which has no refinements by definition). For example, the action "Go to San Francisco airport," represented formally as $Go(Home, SFO)$, might have two possible refinements, as shown in Figure 11.4. The same figure shows a **recursive** refinement for navigation in the vacuum world: to get to a destination, take a step, and then go to the destination.

These examples show that high-level actions and their refinements embody knowledge about *how to do things*. For instance, the refinements for $Go(Home, SFO)$ say that to get to the airport you can drive or take a taxi; buying milk, sitting down, and moving the knight to e4 are not to be considered.

IMPLEMENTATION

An HLA refinement that contains only primitive actions is called an **implementation** of the HLA. For example, in the vacuum world, the sequences $[Right, Right, Down]$ and $[Down, Right, Right]$ both implement the HLA $Navigate([1, 3], [3, 2])$. An implementation of a high-level plan (a sequence of HLAs) is the concatenation of implementations of each HLA in the sequence. Given the precondition–effect definitions of each primitive action, it is straightforward to determine whether any given implementation of a high-level plan achieves the goal. We can say, then, that *a high-level plan achieves the goal from a given state if at least one of its implementations achieves the goal from that state.* The "at least one" in this definition is crucial—not *all* implementations need to achieve the goal, because the agent gets

[1] HTN planners often allow refinement into partially ordered plans, and they allow the refinements of two different HLAs in a plan to *share* actions. We omit these important complications in the interest of understanding the basic concepts of hierarchical planning.

to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning—each of which may have a different outcome—is not the same as the set of possible outcomes in nondeterministic planning. There, we required that a plan work for *all* outcomes because the agent doesn't get to choose the outcome; nature does.

The simplest case is an HLA that has exactly one implementation. In that case, we can compute the preconditions and effects of the HLA from those of the implementation (see Exercise 11.3) and then treat the HLA exactly as if it were a primitive action itself. It can be shown that the right collection of HLAs can result in the time complexity of blind search dropping from exponential in the solution depth to linear in the solution depth, although devising such a collection of HLAs may be a nontrivial task in itself. When HLAs have multiple possible implementations, there are two options: one is to search among the implementations for one that works, as in Section 11.2.2; the other is to reason directly about the HLAs—despite the multiplicity of implementations—as explained in Section 11.2.3. The latter method enables the derivation of provably correct abstract plans, without the need to consider their implementations.

11.2.2 Searching for primitive solutions

HTN planning is often formulated with a single "top level" action called *Act*, where the aim is to find an implementation of *Act* that achieves the goal. This approach is entirely general. For example, classical planning problems can be defined as follows: for each primitive action a_i, provide one refinement of *Act* with steps $[a_i, \mathit{Act}]$. That creates a recursive definition of *Act* that lets us add actions. But we need some way to stop the recursion; we do that by providing one more refinement for *Act*, one with an empty list of steps and with a precondition equal to the goal of the problem. This says that if the goal is already achieved, then the right implementation is to do nothing.

The approach leads to a simple algorithm: repeatedly choose an HLA in the current plan and replace it with one of its refinements, until the plan achieves the goal. One possible implementation based on breadth-first tree search is shown in Figure 11.5. Plans are considered in order of depth of nesting of the refinements, rather than number of primitive steps. It is straightforward to design a graph-search version of the algorithm as well as depth-first and iterative deepening versions.

In essence, this form of hierarchical search explores the space of sequences that conform to the knowledge contained in the HLA library about how things are to be done. A great deal of knowledge can be encoded, not just in the action sequences specified in each refinement but also in the preconditions for the refinements. For some domains, HTN planners have been able to generate huge plans with very little search. For example, O-PLAN (Bell and Tate, 1985), which combines HTN planning with scheduling, has been used to develop production plans for Hitachi. A typical problem involves a product line of 350 different products, 35 assembly machines, and over 2000 different operations. The planner generates a 30-day schedule with three 8-hour shifts a day, involving tens of millions of steps. Another important aspect of HTN plans is that they are, by definition, hierarchically structured; usually this makes them easy for humans to understand.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
     if EMPTY?(frontier) then return failure
     plan ← POP(frontier)  /* chooses the shallowest plan in frontier */
     hla ← the first HLA in plan, or null if none
     prefix,suffix ← the action subsequences before and after hla in plan
     outcome ← RESULT(problem.INITIAL-STATE, prefix)
     if hla is null then  /* so plan is primitive and outcome is its result */
        if outcome satisfies problem.GOAL then return plan
     else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

The computational benefits of hierarchical search can be seen by examining an idealized case. Suppose that a planning problem has a solution with d primitive actions. For a nonhierarchical, forward state-space planner with b allowable actions at each state, the cost is $O(b^d)$, as explained in Chapter 3. For an HTN planner, let us suppose a very regular refinement structure: each nonprimitive action has r possible refinements, each into k actions at the next lower level. We want to know how many different refinement trees there are with this structure. Now, if there are d actions at the primitive level, then the number of levels below the root is $\log_k d$, so the number of internal refinement nodes is $1 + k + k^2 + \cdots + k^{\log_k d - 1} = (d-1)/(k-1)$. Each internal node has r possible refinements, so $r^{(d-1)/(k-1)}$ possible regular decomposition trees could be constructed. Examining this formula, we see that keeping r small and k large can result in huge savings: essentially we are taking the kth root of the nonhierarchical cost, if b and r are comparable. Small r and large k means a library of HLAs with a small number of refinements each yielding a long action sequence (that nonetheless allows us to solve any problem). This is not always possible: long action sequences that are usable across a wide range of problems are extremely precious.

The key to HTN planning, then, is the construction of a plan library containing known methods for implementing complex, high-level actions. One method of constructing the library is to *learn* the methods from problem-solving experience. After the excruciating experience of constructing a plan from scratch, the agent can save the plan in the library as a method for implementing the high-level action defined by the task. In this way, the agent can become more and more competent over time as new methods are built on top of old methods. One important aspect of this learning process is the ability to *generalize* the methods that are constructed, eliminating detail that is specific to the problem instance (e.g., the name of the builder or the address of the plot of land) and keeping just the key elements of the plan. Methods for achieving this kind of generalization are described in Chapter 19. It seems to us inconceivable that humans could be as competent as they are without some such mechanism.

11.2.3 Searching for abstract solutions

The hierarchical search algorithm in the preceding section refines HLAs all the way to primitive action sequences to determine if a plan is workable. This contradicts common sense: one should be able to determine that the two-HLA high-level plan

$$[Drive(Home, SFOLongTermParking), Shuttle(SFOLongTermParking, SFO)]$$

gets one to the airport without having to determine a precise route, choice of parking spot, and so on. The solution seems obvious: write precondition–effect descriptions of the HLAs, just as we write down what the primitive actions do. From the descriptions, it ought to be easy to prove that the high-level plan achieves the goal. This is the holy grail, so to speak, of hierarchical planning because if we derive a high-level plan that provably achieves the goal, working in a small search space of high-level actions, then we can commit to that plan and work on the problem of refining each step of the plan. This gives us the exponential reduction we seek. For this to work, it has to be the case that every high-level plan that "claims" to achieve the goal (by virtue of the descriptions of its steps) does in fact achieve the goal in the sense defined earlier: it must have at least one implementation that does achieve the goal. This property has been called the **downward refinement property** for HLA descriptions.

DOWNWARD REFINEMENT PROPERTY

Writing HLA descriptions that satisfy the downward refinement property is, in principle, easy: as long as the descriptions are *true*, then any high-level plan that claims to achieve the goal must in fact do so—otherwise, the descriptions are making some false claim about what the HLAs do. We have already seen how to write true descriptions for HLAs that have exactly one implementation (Exercise 11.3); a problem arises when the HLA has *multiple* implementations. How can we describe the effects of an action that can be implemented in many different ways?

One safe answer (at least for problems where all preconditions and goals are positive) is to include only the positive effects that are achieved by *every* implementation of the HLA and the negative effects of *any* implementation. Then the downward refinement property would be satisfied. Unfortunately, this semantics for HLAs is much too conservative. Consider again the HLA $Go(Home, SFO)$, which has two refinements, and suppose, for the sake of argument, a simple world in which one can always drive to the airport and park, but taking a taxi requires $Cash$ as a precondition. In that case, $Go(Home, SFO)$ doesn't always get you to the airport. In particular, it fails if $Cash$ is false, and so we cannot assert $At(Agent, SFO)$ as an effect of the HLA. This makes no sense, however; if the agent didn't have $Cash$, it would drive itself. Requiring that an effect hold for *every* implementation is equivalent to assuming that *someone else*—an adversary—will choose the implementation. It treats the HLA's multiple outcomes exactly as if the HLA were a **nondeterministic** action, as in Section 4.3. For our case, the agent itself will choose the implementation.

The programming languages community has coined the term **demonic nondeterminism** for the case where an adversary makes the choices, contrasting this with **angelic nonde-**

DEMONIC NONDETERMINISM

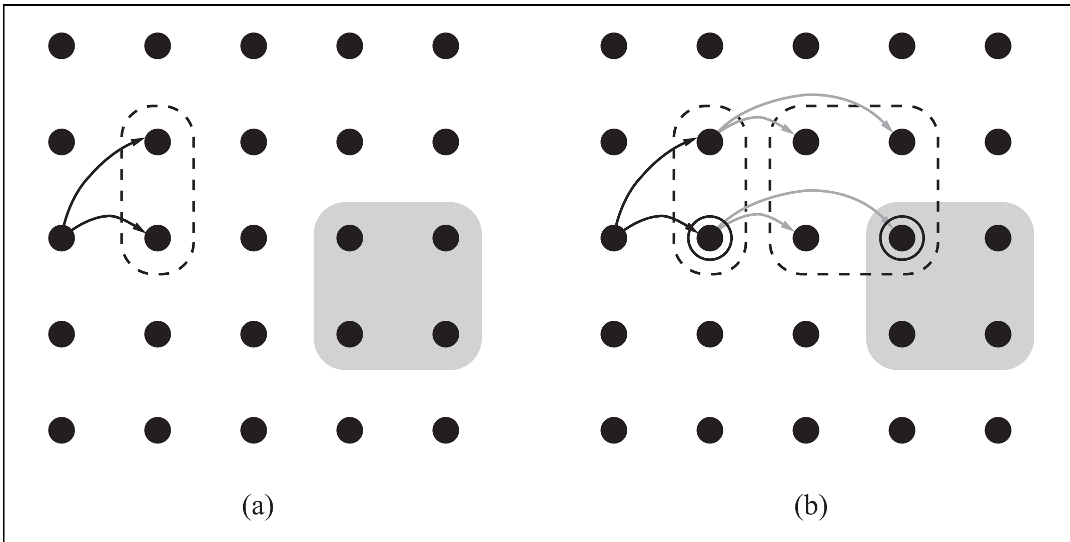

Figure 11.6 Schematic examples of reachable sets. The set of goal states is shaded. Black and gray arrows indicate possible implementations of h_1 and h_2, respectively. (a) The reachable set of an HLA h_1 in a state s. (b) The reachable set for the sequence $[h_1, h_2]$. Because this intersects the goal set, the sequence achieves the goal.

ANGELIC NONDETERMINISM
ANGELIC SEMANTICS
REACHABLE SET

terminism, where the agent itself makes the choices. We borrow this term to define **angelic semantics** for HLA descriptions. The basic concept required for understanding angelic semantics is the **reachable set** of an HLA: given a state s, the reachable set for an HLA h, written as $\text{REACH}(s, h)$, is the set of states reachable by any of the HLA's implementations. The key idea is that the agent can choose *which* element of the reachable set it ends up in when it executes the HLA; thus, an HLA with multiple refinements is more "powerful" than the same HLA with fewer refinements. We can also define the reachable set of a sequences of HLAs. For example, the reachable set of a sequence $[h_1, h_2]$ is the union of all the reachable sets obtained by applying h_2 in each state in the reachable set of h_1:

$$\text{REACH}(s, [h_1, h_2]) = \bigcup_{s' \in \text{REACH}(s, h_1)} \text{REACH}(s', h_2) .$$

Given these definitions, a high-level plan—a sequence of HLAs—achieves the goal if its reachable set *intersects* the set of goal states. (Compare this to the much stronger condition for demonic semantics, where every member of the reachable set has to be a goal state.) Conversely, if the reachable set doesn't intersect the goal, then the plan definitely doesn't work. Figure 11.6 illustrates these ideas.

The notion of reachable sets yields a straightforward algorithm: search among high-level plans, looking for one whose reachable set intersects the goal; once that happens, the algorithm can *commit* to that abstract plan, knowing that it works, and focus on refining the plan further. We will come back to the algorithmic issues later; first, we consider the question of how the effects of an HLA—the reachable set for each possible initial state—are represented. As with the classical action schemas of Chapter 10, we represent the *changes* made to each fluent. Think of a fluent as a state variable. A primitive action can *add* or *delete* a variable or leave it *unchanged*. (With conditional effects (see Section 11.3.1) there is a fourth possibility: flipping a variable to its opposite.)

An HLA under angelic semantics can do more: it can *control* the value of a variable, setting it to true or false depending on which implementation is chosen. In fact, an HLA can have nine different effects on a variable: if the variable starts out true, it can always keep it true, always make it false, or have a choice; if the variable starts out false, it can always keep it false, always make it true, or have a choice; and the three choices for each case can be combined arbitrarily, making nine. Notationally, this is a bit challenging. We'll use the $\tilde{}$ symbol to mean "possibly, if the agent so chooses." Thus, an effect $\tilde{+}A$ means "possibly add A," that is, either leave A unchanged or make it true. Similarly, $\tilde{-}A$ means "possibly delete A" and $\tilde{\pm}A$ means "possibly add or delete A." For example, the HLA $Go(Home, SFO)$, with the two refinements shown in Figure 11.4, possibly deletes $Cash$ (if the agent decides to take a taxi), so it should have the effect $\tilde{-}Cash$. Thus, we see that the descriptions of HLAs are *derivable*, in principle, from the descriptions of their refinements—in fact, this is required if we want true HLA descriptions, such that the downward refinement property holds. Now, suppose we have the following schemas for the HLAs h_1 and h_2:

$$Action(h_1, \text{PRECOND:}\neg A, \text{EFFECT:} A \wedge \tilde{-}B) \ ,$$
$$Action(h_2, \text{PRECOND:}\neg B, \text{EFFECT:} \tilde{+}A \wedge \tilde{\pm}C) \ .$$

That is, h_1 adds A and possible deletes B, while h_2 possibly adds A and has full control over C. Now, if only B is true in the initial state and the goal is $A \wedge C$ then the sequence $[h_1, h_2]$ achieves the goal: we choose an implementation of h_1 that makes B false, then choose an implementation of h_2 that leaves A true and makes C true.

The preceding discussion assumes that the effects of an HLA—the reachable set for any given initial state—can be described exactly by describing the effect on each variable. It would be nice if this were always true, but in many cases we can only approximate the effects because an HLA may have infinitely many implementations and may produce arbitrarily wiggly reachable sets—rather like the wiggly-belief-state problem illustrated in Figure 7.21 on page 271. For example, we said that $Go(Home, SFO)$ possibly deletes $Cash$; it also possibly adds $At(Car, SFOLongTermParking)$; but it cannot do both—in fact, it must do exactly one. As with belief states, we may need to write *approximate* descriptions. We will use two kinds of approximation: an **optimistic description** $\text{REACH}^+(s, h)$ of an HLA h may overstate the reachable set, while a **pessimistic description** $\text{REACH}^-(s, h)$ may understate the reachable set. Thus, we have

OPTIMISTIC DESCRIPTION
PESSIMISTIC DESCRIPTION

$$\text{REACH}^-(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^+(s, h) \ .$$

For example, an optimistic description of $Go(Home, SFO)$ says that it possible deletes $Cash$ *and* possibly adds $At(Car, SFOLongTermParking)$. Another good example arises in the 8-puzzle, half of whose states are unreachable from any given state (see Exercise 3.4 on page 113): the optimistic description of *Act* might well include the whole state space, since the exact reachable set is quite wiggly.

With approximate descriptions, the test for whether a plan achieves the goal needs to be modified slightly. If the optimistic reachable set for the plan doesn't intersect the goal,

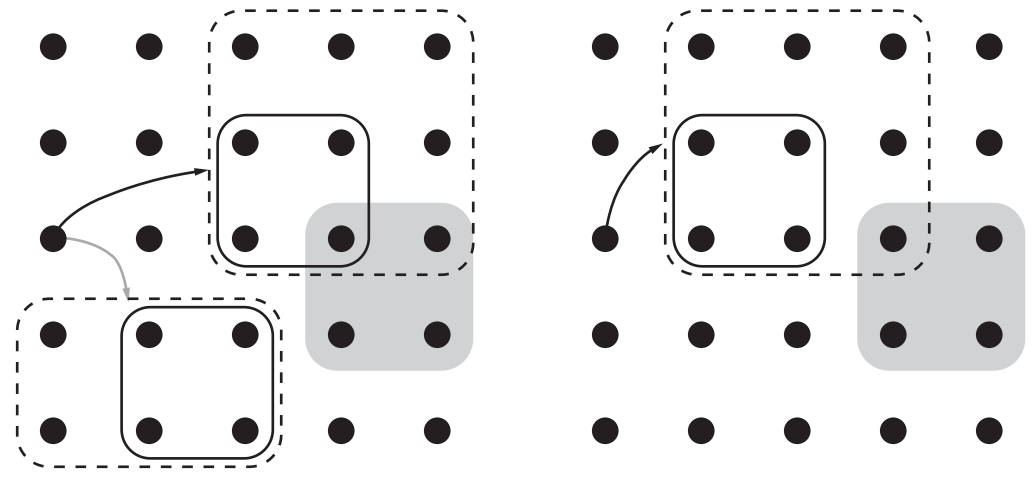

Figure 11.7 Goal achievement for high-level plans with approximate descriptions. The set of goal states is shaded. For each plan, the pessimistic (solid lines) and optimistic (dashed lines) reachable sets are shown. (a) The plan indicated by the black arrow definitely achieves the goal, while the plan indicated by the gray arrow definitely doesn't. (b) A plan that would need to be refined further to determine if it really does achieve the goal.

then the plan doesn't work; if the pessimistic reachable set intersects the goal, then the plan does work (Figure 11.7(a)). With exact descriptions, a plan either works or it doesn't, but with approximate descriptions, there is a middle ground: if the optimistic set intersects the goal but the pessimistic set doesn't, then we cannot tell if the plan works (Figure 11.7(b)). When this circumstance arises, the uncertainty can be resolved by refining the plan. This is a very common situation in human reasoning. For example, in planning the aforementioned two-week Hawaii vacation, one might propose to spend two days on each of seven islands. Prudence would indicate that this ambitious plan needs to be refined by adding details of inter-island transportation.

An algorithm for hierarchical planning with approximate angelic descriptions is shown in Figure 11.8. For simplicity, we have kept to the same overall scheme used previously in Figure 11.5, that is, a breadth-first search in the space of refinements. As just explained, the algorithm can detect plans that will and won't work by checking the intersections of the optimistic and pessimistic reachable sets with the goal. (The details of how to compute the reachable sets of a plan, given approximate descriptions of each step, are covered in Exercise 11.5.) When a workable abstract plan is found, the algorithm *decomposes* the original problem into subproblems, one for each step of the plan. The initial state and goal for each subproblem are obtained by regressing a guaranteed-reachable goal state through the action schemas for each step of the plan. (See Section 10.2.2 for a discussion of how regression works.) Figure 11.6(b) illustrates the basic idea: the right-hand circled state is the guaranteed-reachable goal state, and the left-hand circled state is the intermediate goal obtained by regressing the

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns solution or fail
  frontier ← a FIFO queue with initialPlan as the only element
  loop do
      if EMPTY?(frontier) then return fail
      plan ← POP(frontier)   /* chooses the shallowest node in frontier */
      if REACH+(problem.INITIAL-STATE, plan) intersects problem.GOAL then
          if plan is primitive then return plan   /* REACH+ is exact for primitive plans */
          guaranteed ← REACH−(problem.INITIAL-STATE, plan) ∩ problem.GOAL
          if guaranteed≠{ } and MAKING-PROGRESS(plan, initialPlan) then
              finalState ← any element of guaranteed
              return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
          hla ← some HLA in plan
          prefix,suffix ← the action subsequences before and after hla in plan
          for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
              frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

```
function DECOMPOSE(hierarchy, s0, plan, sf) returns a solution
  solution ← an empty plan
  while plan is not empty do
     action ← REMOVE-LAST(plan)
     si ← a state in REACH−(s0, plan) such that sf ∈REACH−(si, action)
     problem ← a problem with INITIAL-STATE = si and GOAL = sf
     solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
     sf ← si
  return solution
```

Figure 11.8 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the *initialPlan*.

goal through the final action.

The ability to commit to or reject high-level plans can give ANGELIC-SEARCH a significant computational advantage over HIERARCHICAL-SEARCH, which in turn may have a large advantage over plain old BREADTH-FIRST-SEARCH. Consider, for example, cleaning up a large vacuum world consisting of rectangular rooms connected by narrow corridors. It makes sense to have an HLA for *Navigate* (as shown in Figure 11.4) and one for *CleanWholeRoom*. (Cleaning the room could be implemented with the repeated application of another HLA to clean each row.) Since there are five actions in this domain, the cost for BREADTH-FIRST-SEARCH grows as 5^d, where d is the length of the shortest solution (roughly twice the total number of squares); the algorithm cannot manage even two 2×2 rooms. HIERARCHICAL-SEARCH is more efficient, but still suffers from exponential growth because it tries all ways of cleaning that are consistent with the hierarchy. ANGELIC-SEARCH scales approximately linearly in the number of squares—it commits to a good high-level sequence and prunes away the other options. Notice that cleaning a set of rooms by cleaning each room in turn is hardly rocket science: it is easy for humans precisely because of the hierarchical structure of the task. When we consider how difficult humans find it to solve small puzzles such as the 8-puzzle, it seems likely that the human capacity for solving complex problems derives to a great extent from their skill in abstracting and decomposing the problem to eliminate combinatorics.

The angelic approach can be extended to find least-cost solutions by generalizing the notion of reachable set. Instead of a state being reachable or not, it has a cost for the most efficient way to get there. (The cost is ∞ for unreachable states.) The optimistic and pessimistic descriptions bound these costs. In this way, angelic search can find provably optimal abstract plans without considering their implementations. The same approach can be used to obtain effective **hierarchical lookahead** algorithms for online search, in the style of LRTA* (page 152). In some ways, such algorithms mirror aspects of human deliberation in tasks such as planning a vacation to Hawaii—consideration of alternatives is done initially at an abstract level over long time scales; some parts of the plan are left quite abstract until execution time, such as how to spend two lazy days on Molokai, while others parts are planned in detail, such as the flights to be taken and lodging to be reserved—without these refinements, there is no guarantee that the plan would be feasible.

HIERARCHICAL LOOKAHEAD

11.3 PLANNING AND ACTING IN NONDETERMINISTIC DOMAINS

In this section we extend planning to handle partially observable, nondeterministic, and unknown environments. Chapter 4 extended search in similar ways, and the methods here are also similar: **sensorless planning** (also known as **conformant planning**) for environments with no observations; **contingency planning** for partially observable and nondeterministic environments; and **online planning** and **replanning** for unknown environments.

While the basic concepts are the same as in Chapter 4, there are also significant differences. These arise because planners deal with factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent **belief states**—the sets of possible physical states the agent might be in—for unobservable and partially observable environments. We can also take advantage of many of the domain-independent methods given in Chapter 10 for calculating search heuristics.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

$$Init(Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2) \land InView(Table))$$
$$Goal(Color(Chair, c) \land Color(Table, c))$$

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can. The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable list. That is, $Paint(x, can)$ does not mention the variable c, representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action $Paint(x, can, c)$. But in the partially observable case, we might or might not know what color is in the can. (The variable c is universally quantified, just like all the other variables in an action schema.)

$$\begin{aligned}
&Action(RemoveLid(can),\\
&\quad \text{PRECOND:}\, Can(can)\\
&\quad \text{EFFECT:}\, Open(can))\\
&Action(Paint(x, can),\\
&\quad \text{PRECOND:}\, Object(x) \land Can(can) \land Color(can, c) \land Open(can)\\
&\quad \text{EFFECT:}\, Color(x, c))
\end{aligned}$$

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, PERCEPT(s). For planning, we augment PDDL with a new type of schema, the **percept schema**:

PERCEPT SCHEMA

$$\begin{aligned}
&Percept(Color(x, c),\\
&\quad \text{PRECOND:}\, Object(x) \land InView(x)\\
&Percept(Color(can, c),\\
&\quad \text{PRECOND:}\, Can(can) \land InView(can) \land Open(can)
\end{aligned}$$

The first schema says that whenever an object is in view, the agent will perceive the color of the object (that is, for the object x, the agent will learn the truth value of $Color(x, c)$ for all c). The second schema says that if an open can is in view, then the agent perceives the color of the paint in the can. Because there are no exogenous events in this world, the color of an object will remain the same, even if it is not being perceived, until the agent performs an action to change the object's color. Of course, the agent will need an action that causes objects (one at a time) to come into view:

$$\begin{aligned}
&Action(LookAt(x),\\
&\quad \text{PRECOND:}\, InView(y) \land (x \neq y)\\
&\quad \text{EFFECT:}\, InView(x) \land \lnot InView(y))
\end{aligned}$$

For a fully observable environment, we would have a $Percept$ axiom with no preconditions for each fluent. A sensorless agent, on the other hand, has no $Percept$ axioms at all. Note that even a sensorless agent can solve the painting problem. One solution is to open any can of paint and apply it to both chair and table, thus **coercing** them to be the same color (even though the agent doesn't know what the color is).

A contingent planning agent with sensors can generate a better plan. First, look at the table and chair to obtain their colors; if they are already the same then the plan is done. If not, look at the paint cans; if the paint in a can is the same color as one piece of furniture, then apply that paint to the other piece. Otherwise, paint both pieces with any color.

Finally, an online planning agent might generate a contingent plan with fewer branches at first—perhaps ignoring the possibility that no cans match any of the furniture—and deal with problems when they arise by replanning. It could also deal with incorrectness of its action schemas. Whereas a contingent planner simply assumes that the effects of an action always succeed—that painting the chair does the job—a replanning agent would check the result and make an additional plan to fix any unexpected failure, such as an unpainted area or the original color showing through.

In the real world, agents use a combination of approaches. Car manufacturers sell spare tires and air bags, which are physical embodiments of contingent plan branches designed to handle punctures or crashes. On the other hand, most car drivers never consider these possibilities; when a problem arises they respond as replanning agents. In general, agents plan only for contingencies that have important consequences and a nonnegligible chance of happening. Thus, a car driver contemplating a trip across the Sahara desert should make explicit contingency plans for breakdowns, whereas a trip to the supermarket requires less advance planning. We next look at each of the three approaches in more detail.

11.3.1 Sensorless planning

Section 4.4.1 (page 138) introduced the basic idea of searching in belief-state space to find a solution for sensorless problems. Conversion of a sensorless planning problem to a belief-state planning problem works much the same way as it did in Section 4.4.1; the main differences are that the underlying physical transition model is represented by a collection of action schemas and the belief state can be represented by a logical formula instead of an explicitly enumerated set of states. For simplicity, we assume that the underlying planning problem is deterministic.

The initial belief state for the sensorless painting problem can ignore $InView$ fluents because the agent has no sensors. Furthermore, we take as given the unchanging facts $Object(Table) \land Object(Chair) \land Can(C_1) \land Can(C_2)$ because these hold in every belief state. The agent doesn't know the colors of the cans or the objects, or whether the cans are open or closed, but it does know that objects and cans have colors: $\forall\, x \;\; \exists\, c \;\; Color(x, c)$. After Skolemizing, (see Section 9.5), we obtain the initial belief state:

$$b_0 = Color(x, C(x)) \; .$$

In classical planning, where the **closed-world assumption** is made, we would assume that any fluent not mentioned in a state is false, but in sensorless (and partially observable) planning we have to switch to an **open-world assumption** in which states contain both positive and negative fluents, and if a fluent does not appear, its value is unknown. Thus, the belief state corresponds exactly to the set of possible worlds that satisfy the formula. Given this initial belief state, the following action sequence is a solution:

$$[RemoveLid(Can_1), Paint(Chair, Can_1), Paint(Table, Can_1)] \; .$$

We now show how to progress the belief state through the action sequence to show that the final belief state satisfies the goal.

First, note that in a given belief state b, the agent can consider any action whose preconditions are satisfied by b. (The other actions cannot be used because the transition model doesn't define the effects of actions whose preconditions might be unsatisfied.) According to Equation (4.4) (page 139), the general formula for updating the belief state b given an applicable action a in a deterministic world is as follows:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\}$$

where RESULT_P defines the physical transition model. For the time being, we assume that the initial belief state is always a conjunction of literals, that is, a 1-CNF formula. To construct the new belief state b', we must consider what happens to each literal ℓ in each physical state s in b when action a is applied. For literals whose truth value is already known in b, the truth value in b' is computed from the current value and the add list and delete list of the action. (For example, if ℓ is in the delete list of the action, then $\neg\ell$ is added to b'.) What about a literal whose truth value is unknown in b? There are three cases:

1. If the action adds ℓ, then ℓ will be true in b' regardless of its initial value.
2. If the action deletes ℓ, then ℓ will be false in b' regardless of its initial value.
3. If the action does not affect ℓ, then ℓ will retain its initial value (which is unknown) and will not appear in b'.

Hence, we see that the calculation of b' is almost identical to the observable case, which was specified by Equation (10.1) on page 368:

$$b' = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a) .$$

We cannot quite use the set semantics because (1) we must make sure that b' does not contain both ℓ and $\neg\ell$, and (2) atoms may contain unbound variables. But it is still the case that $\text{RESULT}(b, a)$ is computed by starting with b, setting any atom that appears in $\text{DEL}(a)$ to false, and setting any atom that appears in $\text{ADD}(a)$ to true. For example, if we apply $\mathit{RemoveLid}(\mathit{Can}_1)$ to the initial belief state b_0, we get

$$b_1 = \mathit{Color}(x, C(x)) \wedge \mathit{Open}(\mathit{Can}_1) .$$

When we apply the action $\mathit{Paint}(\mathit{Chair}, \mathit{Can}_1)$, the precondition $\mathit{Color}(\mathit{Can}_1, c)$ is satisfied by the known literal $\mathit{Color}(x, C(x))$ with binding $\{x/\mathit{Can}_1, c/C(\mathit{Can}_1)\}$ and the new belief state is

$$b_2 = \mathit{Color}(x, C(x)) \wedge \mathit{Open}(\mathit{Can}_1) \wedge \mathit{Color}(\mathit{Chair}, C(\mathit{Can}_1)) .$$

Finally, we apply the action $\mathit{Paint}(\mathit{Table}, \mathit{Can}_1)$ to obtain

$$\begin{aligned} b_3 = {} & \mathit{Color}(x, C(x)) \wedge \mathit{Open}(\mathit{Can}_1) \wedge \mathit{Color}(\mathit{Chair}, C(\mathit{Can}_1)) \\ & \wedge \mathit{Color}(\mathit{Table}, C(\mathit{Can}_1)) . \end{aligned}$$

The final belief state satisfies the goal, $\mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{Chair}, c)$, with the variable c bound to $C(\mathit{Can}_1)$.

The preceding analysis of the update rule has shown a very important fact: *the family of belief states defined as conjunctions of literals is closed under updates defined by PDDL action schemas.* That is, if the belief state starts as a conjunction of literals, then any update will yield a conjunction of literals. That means that in a world with n fluents, any belief state can be represented by a conjunction of size $O(n)$. This is a very comforting result, considering that there are 2^n states in the world. It says we can compactly represent all the subsets of those 2^n states that we will ever need. Moreover, the process of checking for belief states that are subsets or supersets of previously visited belief states is also easy, at least in the propositional case.

The fly in the ointment of this pleasant picture is that it only works for action schemas that have the *same effects* for all states in which their preconditions are satisfied. It is this property that enables the preservation of the 1-CNF belief-state representation. As soon as the effect can depend on the state, dependencies are introduced between fluents and the 1-CNF property is lost. Consider, for example, the simple vacuum world defined in Section 3.2.1. Let the fluents be AtL and AtR for the location of the robot and $CleanL$ and $CleanR$ for the state of the squares. According to the definition of the problem, the *Suck* action has no precondition—it can always be done. The difficulty is that its effect depends on the robot's location: when the robot is AtL, the result is $CleanL$, but when it is AtR, the result is $CleanR$.

CONDITIONAL EFFECT

For such actions, our action schemas will need something new: a **conditional effect**. These have the syntax "**when** $condition$: $effect$," where *condition* is a logical formula to be compared against the current state, and *effect* is a formula describing the resulting state. For the vacuum world, we have

$$\begin{array}{l} Action(Suck, \\ \quad \text{EFFECT:}\,\textbf{when}\ AtL\text{: } CleanL \wedge \textbf{when}\ AtR\text{: } CleanR)\ . \end{array}$$

When applied to the initial belief state $True$, the resulting belief state is $(AtL \wedge CleanL) \vee (AtR \wedge CleanR)$, which is no longer in 1-CNF. (This transition can be seen in Figure 4.14 on page 141.) In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states of exponential size in the worst case.

It is important to understand the difference between preconditions and conditional effects. *All* conditional effects whose conditions are satisfied have their effects applied to generate the resulting state; if none are satisfied, then the resulting state is unchanged. On the other hand, if a *precondition* is unsatisfied, then the action is inapplicable and the resulting state is undefined. From the point of view of sensorless planning, it is better to have conditional effects than an inapplicable action. For example, we could split $Suck$ into two actions with unconditional effects as follows:

$$\begin{array}{l} Action(SuckL, \\ \quad \text{PRECOND:}\,AtL;\ \text{EFFECT:}\,CleanL) \\ Action(SuckR, \\ \quad \text{PRECOND:}\,AtR;\ \text{EFFECT:}\,CleanR)\ . \end{array}$$

Now we have only unconditional schemas, so the belief states all remain in 1-CNF; unfortunately, we cannot determine the applicability of $SuckL$ and $SuckR$ in the initial belief state.

It seems inevitable, then, that nontrivial problems will involve wiggly belief states, just like those encountered when we considered the problem of state estimation for the wumpus world (see Figure 7.21 on page 271). The solution suggested then was to use a **conservative approximation** to the exact belief state; for example, the belief state can remain in 1-CNF if it contains all literals whose truth values can be determined and treats all other literals as unknown. While this approach is *sound*, in that it never generates an incorrect plan, it is *incomplete* because it may be unable to find solutions to problems that necessarily involve interactions among literals. To give a trivial example, if the goal is for the robot to be on a clean square, then $[Suck]$ is a solution but a sensorless agent that insists on 1-CNF belief states will not find it.

Perhaps a better solution is to look for action sequences that keep the belief state as simple as possible. For example, in the sensorless vacuum world, the action sequence $[Right, Suck, Left, Suck]$ generates the following sequence of belief states:

$$\begin{aligned} b_0 &= True \\ b_1 &= AtR \\ b_2 &= AtR \wedge CleanR \\ b_3 &= AtL \wedge CleanR \\ b_4 &= AtL \wedge CleanR \wedge CleanL \end{aligned}$$

That is, the agent *can* solve the problem while retaining a 1-CNF belief state, even though some sequences (e.g., those beginning with *Suck*) go outside 1-CNF. The general lesson is not lost on humans: we are always performing little actions (checking the time, patting our pockets to make sure we have the car keys, reading street signs as we navigate through a city) to eliminate uncertainty and keep our belief state manageable.

There is another, quite different approach to the problem of unmanageably wiggly belief states: don't bother computing them at all. Suppose the initial belief state is b_0 and we would like to know the belief state resulting from the action sequence $[a_1, \ldots, a_m]$. Instead of computing it explicitly, just represent it as "b_0 then $[a_1, \ldots, a_m]$." This is a lazy but unambiguous representation of the belief state, and it's quite concise—$O(n + m)$ where n is the size of the initial belief state (assumed to be in 1-CNF) and m is the maximum length of an action sequence. As a belief-state representation, it suffers from one drawback, however: determining whether the goal is satisfied, or an action is applicable, may require a lot of computation.

The computation can be implemented as an entailment test: if A_m represents the collection of successor-state axioms required to define occurrences of the actions $a_1, \ldots, a_m$—as explained for SATPLAN in Section 10.4.1—and G_m asserts that the goal is true after m steps, then the plan achieves the goal if $b_0 \wedge A_m \models G_m$, that is, if $b_0 \wedge A_m \wedge \neg G_m$ is unsatisfiable. Given a modern SAT solver, it may be possible to do this much more quickly than computing the full belief state. For example, if none of the actions in the sequence has a particular goal fluent in its add list, the solver will detect this immediately. It also helps if partial results about the belief state—for example, fluents known to be true or false—are cached to simplify subsequent computations.

The final piece of the sensorless planning puzzle is a heuristic function to guide the search. The meaning of the heuristic function is the same as for classical planning: an estimate (perhaps admissible) of the cost of achieving the goal from the given belief state. With belief states, we have one additional fact: solving any subset of a belief state is necessarily easier than solving the belief state:

$$\text{if } b_1 \subseteq b_2 \text{ then } h^*(b_1) \leq h^*(b_2)\ .$$

Hence, any admissible heuristic computed for a subset is admissible for the belief state itself. The most obvious candidates are the singleton subsets, that is, individual physical states. We can take any random collection of states $s_1, \ldots, s_N$ that are in the belief state b, apply any admissible heuristic h from Chapter 10, and return

$$H(b) = \max\{h(s_1), \ldots, h(s_N)\}$$

as the heuristic estimate for solving b. We could also use a planning graph directly on b itself: if it is a conjunction of literals (1-CNF), simply set those literals to be the initial state layer of the graph. If b is not in 1-CNF, it may be possible to find sets of literals that together entail b. For example, if b is in disjunctive normal form (DNF), each term of the DNF formula is a conjunction of literals that entails b and can form the initial layer of a planning graph. As before, we can take the maximum of the heuristics obtained from each set of literals. We can also use inadmissible heuristics such as the ignore-delete-lists heuristic (page 377), which seems to work quite well in practice.

11.3.2 Contingent planning

We saw in Chapter 4 that contingent planning—the generation of plans with conditional branching based on percepts—is appropriate for environments with partial observability, nondeterminism, or both. For the partially observable painting problem with the percept axioms given earlier, one possible contingent solution is as follows:

$$\begin{array}{l}
[\mathit{LookAt}(\mathit{Table}), \mathit{LookAt}(\mathit{Chair}),\\
\quad \textbf{if } \mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{Chair}, c) \textbf{ then } \mathit{NoOp}\\
\quad\quad \textbf{else } [\mathit{RemoveLid}(\mathit{Can}_1), \mathit{LookAt}(\mathit{Can}_1), \mathit{RemoveLid}(\mathit{Can}_2), \mathit{LookAt}(\mathit{Can}_2),\\
\quad\quad\quad \textbf{if } \mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{can}, c) \textbf{ then } \mathit{Paint}(\mathit{Chair}, \mathit{can})\\
\quad\quad\quad \textbf{else if } \mathit{Color}(\mathit{Chair}, c) \wedge \mathit{Color}(\mathit{can}, c) \textbf{ then } \mathit{Paint}(\mathit{Table}, \mathit{can})\\
\quad\quad\quad \textbf{else } [\mathit{Paint}(\mathit{Chair}, \mathit{Can}_1), \mathit{Paint}(\mathit{Table}, \mathit{Can}_1)]]]
\end{array}$$

Variables in this plan should be considered existentially quantified; the second line says that if there exists some color c that is the color of the table and the chair, then the agent need not do anything to achieve the goal. When executing this plan, a contingent-planning agent can maintain its belief state as a logical formula and evaluate each branch condition by determining if the belief state entails the condition formula or its negation. (It is up to the contingent-planning algorithm to make sure that the agent will never end up in a belief state where the condition formula's truth value is unknown.) Note that with first-order conditions, the formula may be satisfied in more than one way; for example, the condition $\mathit{Color}(\mathit{Table}, c) \wedge \mathit{Color}(\mathit{can}, c)$ might be satisfied by $\{\mathit{can}/\mathit{Can}_1\}$ and by $\{\mathit{can}/\mathit{Can}_2\}$ if both cans are the same color as the table. In that case, the agent can choose any satisfying substitution to apply to the rest of the plan.

As shown in Section 4.4.2, calculating the new belief state after an action and subsequent percept is done in two stages. The first stage calculates the belief state after the action, just as for the sensorless agent:

$$\hat{b} = (b - \textsc{Del}(a)) \cup \textsc{Add}(a)$$

where, as before, we have assumed a belief state represented as a conjunction of literals. The second stage is a little trickier. Suppose that percept literals $p_1, \ldots, p_k$ are received. One might think that we simply need to add these into the belief state; in fact, we can also infer that the preconditions for sensing are satisfied. Now, if a percept p has exactly one percept axiom, $Percept(p, \text{PRECOND}{:}c)$, where c is a conjunction of literals, then those literals can be thrown into the belief state along with p. On the other hand, if p has more than one percept axiom whose preconditions might hold according to the predicted belief state $\hat{b}$, then we have to add in the *disjunction* of the preconditions. Obviously, this takes the belief state outside 1-CNF and brings up the same complications as conditional effects, with much the same classes of solutions.

Given a mechanism for computing exact or approximate belief states, we can generate contingent plans with an extension of the AND–OR forward search over belief states used in Section 4.4. Actions with nondeterministic effects—which are defined simply by using a disjunction in the EFFECT of the action schema—can be accommodated with minor changes to the belief-state update calculation and no change to the search algorithm.[2] For the heuristic function, many of the methods suggested for sensorless planning are also applicable in the partially observable, nondeterministic case.

11.3.3 Online replanning

Imagine watching a spot-welding robot in a car plant. The robot's fast, accurate motions are repeated over and over again as each car passes down the line. Although technically impressive, the robot probably does not seem at all *intelligent* because the motion is a fixed, preprogrammed sequence; the robot obviously doesn't "know what it's doing" in any meaningful sense. Now suppose that a poorly attached door falls off the car just as the robot is about to apply a spot-weld. The robot quickly replaces its welding actuator with a gripper, picks up the door, checks it for scratches, reattaches it to the car, sends an email to the floor supervisor, switches back to the welding actuator, and resumes its work. All of a sudden, the robot's behavior seems *purposive* rather than rote; we assume it results not from a vast, precomputed contingent plan but from an online replanning process—which means that the robot *does* need to know what it's trying to do.

EXECUTION MONITORING

Replanning presupposes some form of **execution monitoring** to determine the need for a new plan. One such need arises when a contingent planning agent gets tired of planning for every little contingency, such as whether the sky might fall on its head.[3] Some branches of a partially constructed contingent plan can simply say $Replan$; if such a branch is reached during execution, the agent reverts to planning mode. As we mentioned earlier, the decision as to how much of the problem to solve in advance and how much to leave to replanning is one that involves tradeoffs among possible events with different costs and probabilities of occurring. Nobody wants to have their car break down in the middle of the Sahara desert and only then think about having enough water.

[2] If cyclic solutions are required for a nondeterministic problem, AND–OR search must be generalized to a loopy version such as LAO* (Hansen and Zilberstein, 2001).

[3] In 1954, a Mrs. Hodges of Alabama was hit by meteorite that crashed through her roof. In 1992, a piece of the Mbale meteorite hit a small boy on the head; fortunately, its descent was slowed by banana leaves (Jenniskens *et al.*, 1994). And in 2009, a German boy claimed to have been hit in the hand by a pea-sized meteorite. No serious injuries resulted from any of these incidents, suggesting that the need for preplanning against such contingencies is sometimes overstated.

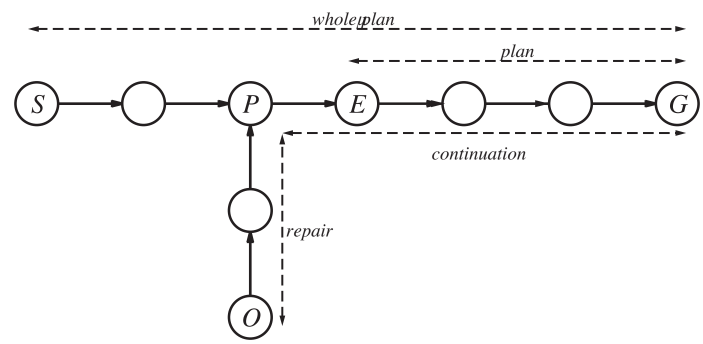

Figure 11.9 Before execution, the planner comes up with a plan, here called $whole\ plan$, to get from S to G. The agent executes steps of the plan until it expects to be in state E, but observes it is actually in O. The agent then replans for the minimal repair plus continuation to reach G.

MISSING PRECONDITION

MISSING EFFECT

MISSING STATE VARIABLE

EXOGENOUS EVENT

Replanning may also be needed if the agent's model of the world is incorrect. The model for an action may have a **missing precondition**—for example, the agent may not know that removing the lid of a paint can often requires a screwdriver; the model may have a **missing effect**—for example, painting an object may get paint on the floor as well; or the model may have a **missing state variable**—for example, the model given earlier has no notion of the amount of paint in a can, of how its actions affect this amount, or of the need for the amount to be nonzero. The model may also lack provision for **exogenous events** such as someone knocking over the paint can. Exogenous events can also include changes in the goal, such as the addition of the requirement that the table and chair not be painted black. Without the ability to monitor and replan, an agent's behavior is likely to be extremely fragile if it relies on absolute correctness of its model.

The online agent has a choice of how carefully to monitor the environment. We distinguish three levels:

ACTION MONITORING

PLAN MONITORING

GOAL MONITORING

- **Action monitoring**: before executing an action, the agent verifies that all the preconditions still hold.
- **Plan monitoring**: before executing an action, the agent verifies that the remaining plan will still succeed.
- **Goal monitoring**: before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

In Figure 11.9 we see a schematic of action monitoring. The agent keeps track of both its original plan, $wholeplan$, and the part of the plan that has not been executed yet, which is denoted by $plan$. After executing the first few steps of the plan, the agent expects to be in state E. But the agent observes it is actually in state O. It then needs to repair the plan by finding some point P on the original plan that it can get back to. (It may be that P is the goal state, G.) The agent tries to minimize the total cost of the plan: the repair part (from O to P) plus the continuation (from P to G).

Now let's return to the example problem of achieving a chair and table of matching color. Suppose the agent comes up with this plan:

$$
\begin{array}{l}
[LookAt(Table), LookAt(Chair), \\
\quad \textbf{if } Color(Table, c) \wedge Color(Chair, c) \textbf{ then } NoOp \\
\qquad \textbf{else } [RemoveLid(Can_1), LookAt(Can_1), \\
\qquad\qquad \textbf{if } Color(Table, c) \wedge Color(Can_1, c) \textbf{ then } Paint(Chair, Can_1) \\
\qquad\qquad \textbf{else } \text{REPLAN}]]\ .
\end{array}
$$

Now the agent is ready to execute the plan. Suppose the agent observes that the table and can of paint are white and the chair is black. It then executes $Paint(Chair, Can_1)$. At this point a classical planner would declare victory; the plan has been executed. But an online execution monitoring agent needs to check the preconditions of the remaining empty plan—that the table and chair are the same color. Suppose the agent perceives that they do not have the same color—in fact, the chair is now a mottled gray because the black paint is showing through. The agent then needs to figure out a position in $whole\ plan$ to aim for and a repair action sequence to get there. The agent notices that the current state is identical to the precondition before the $Paint(Chair, Can_1)$ action, so the agent chooses the empty sequence for $repair$ and makes its $plan$ be the same $[Paint]$ sequence that it just attempted. With this new plan in place, execution monitoring resumes, and the $Paint$ action is retried. This behavior will loop until the chair is perceived to be completely painted. But notice that the loop is created by a process of plan–execute–replan, rather than by an explicit loop in a plan. Note also that the original plan need not cover every contingency. If the agent reaches the step marked REPLAN, it can then generate a new plan (perhaps involving Can_2).

Action monitoring is a simple method of execution monitoring, but it can sometimes lead to less than intelligent behavior. For example, suppose there is no black or white paint, and the agent constructs a plan to solve the painting problem by painting both the chair and table red. Suppose that there is only enough red paint for the chair. With action monitoring, the agent would go ahead and paint the chair red, then notice that it is out of paint and cannot paint the table, at which point it would replan a repair—perhaps painting both chair and table green. A plan-monitoring agent can detect failure whenever the current state is such that the remaining plan no longer works. Thus, it would not waste time painting the chair red. Plan monitoring achieves this by checking the preconditions for success of the entire remaining plan—that is, the preconditions of each step in the plan, except those preconditions that are achieved by another step in the remaining plan. Plan monitoring cuts off execution of a doomed plan as soon as possible, rather than continuing until the failure actually occurs.[4] Plan monitoring also allows for **serendipity**—accidental success. If someone comes along and paints the table red at the same time that the agent is painting the chair red, then the final plan preconditions are satisfied (the goal has been achieved), and the agent can go home early.

It is straightforward to modify a planning algorithm so that each action in the plan is annotated with the action's preconditions, thus enabling action monitoring. It is slightly

[4] Plan monitoring means that finally, after 424 pages, we have an agent that is smarter than a dung beetle (see page 39). A plan-monitoring agent would notice that the dung ball was missing from its grasp and would replan to get another ball and plug its hole.

more complex to enable plan monitoring. Partial-order and planning-graph planners have the advantage that they have already built up structures that contain the relations necessary for plan monitoring. Augmenting state-space planners with the necessary annotations can be done by careful bookkeeping as the goal fluents are regressed through the plan.

Now that we have described a method for monitoring and replanning, we need to ask, "Does it work?" This is a surprisingly tricky question. If we mean, "Can we guarantee that the agent will always achieve the goal?" then the answer is no, because the agent could inadvertently arrive at a dead end from which there is no repair. For example, the vacuum agent might have a faulty model of itself and not know that its batteries can run out. Once they do, it cannot repair any plans. If we rule out dead ends—assume that there exists a plan to reach the goal from *any* state in the environment—and assume that the environment is really nondeterministic, in the sense that such a plan always has *some* chance of success on any given execution attempt, then the agent will eventually reach the goal.

Trouble occurs when an action is actually not nondeterministic, but rather depends on some precondition that the agent does not know about. For example, sometimes a paint can may be empty, so painting from that can has no effect. No amount of retrying is going to change this.[5] One solution is to choose randomly from among the set of possible repair plans, rather than to try the same one each time. In this case, the repair plan of opening another can might work. A better approach is to **learn** a better model. Every prediction failure is an opportunity for learning; an agent should be able to modify its model of the world to accord with its percepts. From then on, the replanner will be able to come up with a repair that gets at the root problem, rather than relying on luck to choose a good repair. This kind of learning is described in Chapters 18 and 19.

11.4 MULTIAGENT PLANNING

So far, we have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a **multiagent planning problem** in which it tries to achieve its own goals with the help or hindrance of others.

MULTIAGENT PLANNING PROBLEM

Between the purely single-agent and truly multiagent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do **multieffector planning** to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory—multieffector planning becomes **multibody planning**. A multibody problem is still a "standard" single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body. When communication constraints make this impossible, we have

MULTIEFFECTOR PLANNING

MULTIBODY PLANNING

[5] Futile repetition of a plan repair is exactly the behavior exhibited by the sphex wasp (page 39).

DECENTRALIZED PLANNING

what is sometimes called a **decentralized planning** problem; this is perhaps a misnomer, because the planning phase is centralized but the execution phase is at least partially decoupled. In this case, the subplan constructed for each body may need to include explicit communicative actions with other bodies. For example, multiple reconnaissance robots covering a wide area may often be out of radio contact with each other and should share their findings during times when communication is feasible.

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multiagent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multiagent doubles team, on the other hand, each agent decides what to do; without some method for **coordination**, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

COORDINATION

The clearest case of a multiagent problem, of course, is when the agents have different goals. In tennis, the goals of two opposing teams are in direct conflict, leading to the zero-sum situation of Chapter 5. Spectators could be viewed as agents if their support or disdain is a significant factor and can be influenced by the players' conduct; otherwise, they can be treated as an aspect of nature—just like the weather—that is assumed to be indifferent to the players' intentions.[6]

Finally, some systems are a mixture of centralized and multiagent planning. For example, a delivery company may do centralized, offline planning for the routes of its trucks and planes each day, but leave some aspects open for autonomous decisions by drivers and pilots who can respond individually to traffic and weather situations. Also, the goals of the company and its employees are brought into alignment, to some extent, by the payment of **incentives** (salaries and bonuses)—a sure sign that this is a true multiagent system.

INCENTIVE

The issues involved in multiagent planning can be divided roughly into two sets. The first, covered in Section 11.4.1, involves issues of representing and planning for multiple simultaneous actions; these issues occur in all settings from multieffector to multiagent planning. The second, covered in Section 11.4.2, involves issues of cooperation, coordination, and competition arising in true multiagent settings.

11.4.1 Planning with multiple simultaneous actions

MULTIACTOR

ACTOR

For the time being, we will treat the multieffector, multibody, and multiagent settings in the same way, labeling them generically as **multiactor** settings, using the generic term **actor** to cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multiactor setting. A correct plan is one that, if executed by the actors, achieves the goal. (In the true multiagent setting, of course, the agents may not agree to execute any particular plan, but at least they

[6] We apologize to residents of the United Kingdom, where the mere act of contemplating a game of tennis guarantees rain.

$$
\begin{aligned}
&Actors(A, B)\\
&Init(At(A, LeftBaseline) \ \wedge\ At(B, RightNet) \ \wedge\\
&\qquad Approaching(Ball, RightBaseline)) \ \wedge\ Partner(A, B) \ \wedge\ Partner(B, A)\\
&Goal(Returned(Ball) \ \wedge\ (At(a, RightNet) \ \vee\ At(a, LeftNet))\\
&Action(Hit(actor, Ball),\\
&\qquad \text{PRECOND:} Approaching(Ball, loc) \ \wedge\ At(actor, loc)\\
&\qquad \text{EFFECT:} Returned(Ball))\\
&Action(Go(actor, to),\\
&\qquad \text{PRECOND:} At(actor, loc) \ \wedge\ to \neq loc,\\
&\qquad \text{EFFECT:} At(actor, to) \ \wedge\ \neg\ At(actor, loc))
\end{aligned}
$$

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: $LeftBaseline$, $RightBaseline$, $LeftNet$, and $RightNet$. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

will know what plans *would* work if they *did* agree to execute them.) For simplicity, we assume perfect **synchronization**: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

SYNCHRONIZATION

We begin with the transition model; for the deterministic case, this is the function RESULT(s, a). In the single-agent setting, there might be b different choices for the action; b can be quite large, especially for first-order representations with many objects to act on, but action schemas provide a concise representation nonetheless. In the multiactor setting with n actors, the single action a is replaced by a **joint action** $\langle a_1, \ldots, a_n \rangle$, where a_i is the action taken by the ith actor. Immediately, we see two problems: first, we have to describe the transition model for b^n different joint actions; second, we have a joint planning problem with a branching factor of b^n.

JOINT ACTION

Having put the actors together into a multiactor system with a huge branching factor, the principal focus of research on multiactor planning has been to *decouple* the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are **loosely coupled**, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI. We have seen it explicitly in the context of CSPs, where "tree like" constraint graphs yielded efficient solution methods (see page 225), as well as in the context of disjoint pattern databases (page 106) and additive heuristics for planning (page 378).

LOOSELY COUPLED

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let's see how this works for the doubles tennis problem. Let's suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

JOINT PLAN

A first pass at a multiactor definition might look like Figure 11.10. With this definition, it is easy to see that the following **joint plan** plan works:

PLAN 1:

$$A: \ [Go(A, RightBaseline), Hit(A, Ball)]$$
$$B: \ [NoOp(B), NoOp(B)]\ .$$

CONCURRENT ACTION LIST

Problems arise, however, when a plan has both agents hitting the ball at the same time. In the real world, this won't work, but the action schema for *Hit* says that the ball will be returned successfully. Technically, the difficulty is that preconditions constrain the *state* in which an action can be executed successfully, but do not constrain other actions that might mess it up. We solve this by augmenting action schemas with one new feature: a **concurrent action list** stating which actions must or must not be executed concurrently. For example, the Hit action could be described as follows:

$$\begin{array}{l} Action(Hit(a, Ball), \\ \quad \text{CONCURRENT:} b \neq a \ \Rightarrow \ \neg Hit(b, Ball) \\ \quad \text{PRECOND:} Approaching(Ball, loc) \wedge At(a, loc) \\ \quad \text{EFFECT:} Returned(Ball))\ . \end{array}$$

In other words, the Hit action has its stated effect only if no other Hit action by another agent occurs at the same time. (In the SATPLAN approach, this would be handled by a partial **action exclusion axiom**.) For some actions, the desired effect is achieved *only* when another action occurs concurrently. For example, two agents are needed to carry a cooler full of beverages to the tennis court:

$$\begin{array}{l} Action(Carry(a, cooler, here, there), \\ \quad \text{CONCURRENT:} b \neq a \wedge Carry(b, cooler, here, there) \\ \quad \text{PRECOND:} At(a, here) \wedge At(cooler, here) \wedge Cooler(cooler) \\ \quad \text{EFFECT:} At(a, there) \wedge At(cooler, there) \wedge \neg At(a, here) \wedge \neg At(cooler, here)). \end{array}$$

With these kinds of action schemas, any of the planning algorithms described in Chapter 10 can be adapted with only minor modifications to generate multiactor plans. To the extent that the coupling among subplans is loose—meaning that concurrency constraints come into play only rarely during plan search—one would expect the various heuristics derived for single-agent planning to also be effective in the multiactor context. We could extend this approach with the refinements of the last two chapters—HTNs, partial observability, conditionals, execution monitoring, and replanning—but that is beyond the scope of this book.

11.4.2 Planning with multiple agents: Cooperation and coordination

Now let us consider the true multiagent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the "*the*" in "*the joint solution*" is misleading. For our doubles team, more than one joint solution exists:

PLAN 2:

$$A: \ [Go(A, LeftNet), NoOp(A)]$$
$$B: \ [Go(B, RightBaseline), Hit(B, Ball)]\ .$$

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if A chooses plan 2 and B chooses plan 1, then nobody will return the ball. Conversely, if A chooses 1 and B chooses 2, then they will both try to hit the ball. The agents may realize this, but how can they coordinate to make sure they agree on the plan?

CONVENTION

One option is to adopt a **convention** before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention "stick to your side of the court" would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention "stay on the right side of the road" in most countries; the alternative, "stay on the left side," works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called **social laws**.

SOCIAL LAWS

In the absence of a convention, agents can use **communication** to achieve common knowledge of a feasible joint plan. For example, a tennis player could shout "Mine!" or "Yours!" to indicate a preferred joint plan. We cover mechanisms for communication in more depth in Chapter 22, where we observe that communication does not necessarily involve a verbal exchange. For example, one player can communicate a preferred joint plan to the other simply by executing the first part of it. If agent A heads for the net, then agent B is obliged to go back to the baseline to hit the ball, because plan 2 is the only joint plan that begins with A's heading for the net. This approach to coordination, sometimes called **plan recognition**, works when a single action (or short sequence of actions) is enough to determine a joint plan unambiguously. Note that communication can work as well with competitive agents as with cooperative ones.

PLAN RECOGNITION

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen's job is to reproduce, not to do centralized planning—and with very limited computation, communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role according to the local conditions it observes. For example, foragers travel away from the nest, search for a seed, and when they find one, bring it back immediately. Thus, the rate at which foragers return to the nest is an approximation of the availability of food today. When the rate is high, other ants abandon their current role and take on the role of scavenger. The ants appear to have a convention on the importance of roles—foraging is the most important—and ants will easily switch into the more important roles, but not into the less important. There is some learning mechanism: a colony learns to make more successful and prudent actions over the course of its decades-long life, even though individual ants live only about a year.

One final example of cooperative multiagent behavior appears in the flocking behavior of birds. We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a **boid**) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components:

BOID

(a) (b) (c)

Figure 11.11 (a) A simulated flock of birds, using Reynold's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors

EMERGENT BEHAVIOR

If all the boids execute this policy, the flock exhibits the **emergent behavior** of flying as a pseudorigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents.

The most difficult multiagent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control. We see this in games such as robotic soccer or the NERO game shown in Figure 11.11(c), in which two teams of software agents compete to capture the control towers. As yet, methods for efficient planning in these kinds of environments—for example, taking advantage of loose coupling—are in their infancy.

11.5 SUMMARY

This chapter has addressed some of the complications of planning and acting in the real world. The main points:

- Many actions consume **resources**, such as money, gas, or raw materials. It is convenient to treat these resources as numeric measures in a pool rather than try to reason about, say, each individual coin and bill in the world. Actions can generate and consume resources, and it is usually cheap and effective to check partial plans for satisfaction of resource constraints before attempting further refinements.
- Time is one of the most important resources. It can be handled by specialized scheduling algorithms, or scheduling can be integrated with planning.

- **Hierarchical task network** (HTN) planning allows the agent to take advice from the domain designer in the form of **high-level actions** (HLAs) that can be implemented in various ways by lower-level action sequences. The effects of HLAs can be defined with **angelic semantics**, allowing provably correct high-level plans to be derived without consideration of lower-level implementations. HTN methods can create the very large plans required by many real-world applications.
- Standard planning algorithms assume complete and correct information and deterministic, fully observable environments. Many domains violate this assumption.
- **Contingent plans** allow the agent to sense the world during execution to decide what branch of the plan to follow. In some cases, **sensorless** or **conformant planning** can be used to construct a plan that works without the need for perception. Both conformant and contingent plans can be constructed by search in the space of **belief states**. Efficient representation or computation of belief states is a key problem.
- An **online planning agent** uses execution monitoring and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment.
- **Multiagent** planning is necessary when there are other agents in the environment with which to cooperate or compete. Joint plans can be constructed, but must be augmented with some form of coordination if two agents are to agree on which joint plan to execute.
- This chapter extends classic planning to cover nondeterministic environments (where outcomes of actions are uncertain), but it is not the last word on planning. Chapter 17 describes techniques for stochastic environments (in which outcomes of actions have probabilities associated with them): Markov decision processes, partially observable Markov decision processes, and game theory. In Chapter 21 we show that reinforcement learning allows an agent to learn how to behave from past successes and failures.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Planning with time constraints was first dealt with by DEVISER (Vere, 1983). The representation of time in plans was addressed by Allen (1984) and by Dean *et al.* (1990) in the FORBIN system. NONLIN+ (Tate and Whiter, 1984) and SIPE (Wilkins, 1988, 1990) could reason about the allocation of limited resources to various plan steps. O-PLAN (Bell and Tate, 1985), an HTN planner, had a uniform, general representation for constraints on time and resources. In addition to the Hitachi application mentioned in the text, O-PLAN has been applied to software procurement planning at Price Waterhouse and back-axle assembly planning at Jaguar Cars.

The two planners SAPA (Do and Kambhampati, 2001) and T4 (Haslum and Geffner, 2001) both used forward state-space search with sophisticated heuristics to handle actions with durations and resources. An alternative is to use very expressive action languages, but guide them by human-written domain-specific heuristics, as is done by ASPEN (Fukunaga *et al.*, 1997), HSTS (Jonsson *et al.*, 2000), and IxTeT (Ghallab and Laruelle, 1994).

A number of hybrid planning-and-scheduling systems have been deployed: ISIS (Fox *et al.*, 1982; Fox, 1990) has been used for job shop scheduling at Westinghouse, GARI (Descotte and Latombe, 1985) planned the machining and construction of mechanical parts, FORBIN was used for factory control, and NONLIN+ was used for naval logistics planning. We chose to present planning and scheduling as two separate problems; (Cushing *et al.*, 2007) show that this can lead to incompleteness on certain problems. There is a long history of scheduling in aerospace. T-SCHED (Drabble, 1990) was used to schedule mission-command sequences for the UOSAT-II satellite. OPTIMUM-AIV (Aarup *et al.*, 1994) and PLAN-ERS1 (Fuchs *et al.*, 1990), both based on O-PLAN, were used for spacecraft assembly and observation planning, respectively, at the European Space Agency. SPIKE (Johnston and Adorf, 1992) was used for observation planning at NASA for the Hubble Space Telescope, while the Space Shuttle Ground Processing Scheduling System (Deale *et al.*, 1994) does job-shop scheduling of up to 16,000 worker-shifts. Remote Agent (Muscettola *et al.*, 1998) became the first autonomous planner–scheduler to control a spacecraft when it flew onboard the Deep Space One probe in 1999. Space applications have driven the development of algorithms for resource allocations; see Laborie (2003) and Muscettola (2002). The literature on scheduling is presented in a classic survey article (Lawler *et al.*, 1993), a recent book (Pinedo, 2008), and an edited handbook (Blazewicz *et al.*, 2007).

MACROPS

The facility in the STRIPS program for learning **macrops**—“macro-operators” consisting of a sequence of primitive steps—could be considered the first mechanism for hierarchical planning (Fikes *et al.*, 1972). Hierarchy was also used in the LAWALY system (Siklossy and Dreussi, 1973). The ABSTRIPS system (Sacerdoti, 1974) introduced the idea of an **abstraction hierarchy**, whereby planning at higher levels was permitted to ignore lower-level preconditions of actions in order to derive the general structure of a working plan. Austin Tate’s Ph.D. thesis (1975b) and work by Earl Sacerdoti (1977) developed the basic ideas of HTN planning in its modern form. Many practical planners, including O-PLAN and SIPE, are HTN planners. Yang (1990) discusses properties of actions that make HTN planning efficient. Erol, Hendler, and Nau (1994, 1996) present a complete hierarchical decomposition planner as well as a range of complexity results for pure HTN planners. Our presentation of HLAs and angelic semantics is due to Marthi *et al.* (2007, 2008). Kambhampati *et al.* (1998) have proposed an approach in which decompositions are just another form of plan refinement, similar to the refinements for non-hierarchical partial-order planning.

ABSTRACTION HIERARCHY

Beginning with the work on macro-operators in STRIPS, one of the goals of hierarchical planning has been the reuse of previous planning experience in the form of generalized plans. The technique of **explanation-based learning**, described in depth in Chapter 19, has been applied in several systems as a means of generalizing previously computed plans, including SOAR (Laird *et al.*, 1986) and PRODIGY (Carbonell *et al.*, 1989). An alternative approach is to store previously computed plans in their original form and then reuse them to solve new, similar problems by analogy to the original problem. This is the approach taken by the field called **case-based planning** (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) argues that case-based planning should be analyzed as a form of refinement planning and provides a formal foundation for case-based partial-order planning.

CASE-BASED PLANNING

Early planners lacked conditionals and loops, but some could use coercion to form conformant plans. Sacerdoti's NOAH solved the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state, using coercion. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and described a sensorless plan that can move a tool into a specific position on a table by a sequence of tilting actions, *regardless* of the initial position.

Goldman and Boddy (1996) introduced the term **conformant planning**, noting that sensorless plans are often effective even if the agent has sensors. The first moderately efficient conformant planner was Smith and Weld's (1998) Conformant Graphplan or CGP. Ferraris and Giunchiglia (2000) and Rintanen (1999) independently developed SATPLAN-based conformant planners. Bonet and Geffner (2000) describe a conformant planner based on heuristic search in the space of belief states, drawing on ideas first developed in the 1960s for partially observable Markov decision processes, or POMDPs (see Chapter 17).

Currently, there are three main approaches to conformant planning. The first two use heuristic search in belief-state space: HSCP (Bertoli *et al.*, 2001a) uses binary decision diagrams (BDDs) to represent belief states, whereas Hoffmann and Brafman (2006) adopt the lazy approach of computing precondition and goal tests on demand using a SAT solver. The third approach, championed primarily by Jussi Rintanen (2007), formulates the entire sensorless planning problem as a quantified Boolean formula (QBF) and solves it using a general-purpose QBF solver. Current conformant planners are five orders of magnitude faster than CGP. The winner of the 2006 conformant-planning track at the International Planning Competition was T_0 (Palacios and Geffner, 2007), which uses heuristic search in belief-state space while keeping the belief-state representation simple by defining derived literals that cover conditional effects. Bryce and Kambhampati (2007) discuss how a planning graph can be generalized to generate good heuristics for conformant and contingent planning.

There has been some confusion in the literature between the terms "conditional" and "contingent" planning. Following Majercik and Littman (2003), we use "conditional" to mean a plan (or action) that has different effects depending on the actual state of the world, and "contingent" to mean a plan in which the agent can choose different actions depending on the results of sensing. The problem of contingent planning received more attention after the publication of Drew McDermott's (1978a) influential article, *Planning and Acting*.

The contingent-planning approach described in the chapter is based on Hoffmann and Brafman (2005), and was influenced by the efficient search algorithms for cyclic AND–OR graphs developed by Jimenez and Torras (2000) and Hansen and Zilberstein (2001). Bertoli *et al.* (2001b) describe MBP (Model-Based Planner), which uses binary decision diagrams to do conformant and contingent planning.

In retrospect, it is now possible to see how the major classical planning algorithms led to extended versions for uncertain domains. Fast-forward heuristic search through state space led to forward search in belief space (Bonet and Geffner, 2000; Hoffmann and Brafman, 2005); SATPLAN led to stochastic SATPLAN (Majercik and Littman, 2003) and to planning with quantified Boolean logic (Rintanen, 2007); partial order planning led to UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992); GRAPHPLAN led to Sensory Graphplan or SGP (Weld *et al.*, 1998).

The first online planner with execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. The NASL planner (McDermott, 1978a) treated a planning problem simply as a specification for carrying out a complex action, so that execution and planning were completely unified. SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988, 1990) was the first planner to deal systematically with the problem of replanning. It has been used in demonstration projects in several domains, including planning operations on the flight deck of an aircraft carrier, job-shop scheduling for an Australian beer factory, and planning the construction of multistory buildings (Kartam and Levitt, 1990).

REACTIVE PLANNING

In the mid-1980s, pessimism about the slow run times of planning systems led to the proposal of reflex agents called **reactive planning** systems (Brooks, 1986; Agre and Chapman, 1987). PENGI (Agre and Chapman, 1987) could play a (fully observable) video game by using Boolean circuits combined with a "visual" representation of current goals and the agent's internal state. "Universal plans" (Schoppers, 1987, 1989) were developed as a lookup-table method for reactive planning, but turned out to be a rediscovery of the idea of **policies** that had long been used in Markov decision processes (see Chapter 17). A universal plan (or a policy) contains a mapping from any state to the action that should be taken in that state. Koenig (2001) surveys online planning techniques, under the name *Agent-Centered Search*.

POLICY

Multiagent planning has leaped in popularity in recent years, although it does have a long history. Konolige (1982) formalizes multiagent planning in first-order logic, while Pednault (1986) gives a STRIPS-style description. The notion of joint intention, which is essential if agents are to execute a joint plan, comes from work on communicative acts (Cohen and Levesque, 1990; Cohen *et al.*, 1990). Boutilier and Brafman (2001) show how to adapt partial-order planning to a multiactor setting. Brafman and Domshlak (2008) devise a multiactor planning algorithm whose complexity grows only linearly with the number of actors, provided that the degree of coupling (measured partly by the **tree width** of the graph of interactions among agents) is bounded. Petrik and Zilberstein (2009) show that an approach based on bilinear programming outperforms the cover-set approach we outlined in the chapter.

We have barely skimmed the surface of work on negotiation in multiagent planning. Durfee and Lesser (1989) discuss how tasks can be shared out among agents by negotiation. Kraus *et al.* (1991) describe a system for playing Diplomacy, a board game requiring negotiation, coalition formation, and dishonesty. Stone (2000) shows how agents can cooperate as teammates in the competitive, dynamic, partially observable environment of robotic soccer. In a later article, Stone (2003) analyzes two competitive multiagent environments—RoboCup, a robotic soccer competition, and TAC, the auction-based Trading Agents Competition—and finds that the computational intractability of our current theoretically well-founded approaches has led to many multiagent systems being designed by *ad hoc* methods.

In his highly influential *Society of Mind* theory, Marvin Minsky (1986, 2007) proposes that human minds are constructed from an ensemble of agents. Livnat and Pippenger (2006) prove that, for the problem of optimal path-finding, and given a limitation on the total amount of computing resources, the best architecture for an agent is an ensemble of subagents, each of which tries to optimize its own objective, and all of which are in conflict with one another.

The boid model on page 429 is due to Reynolds (1987), who won an Academy Award for its application to swarms of penguins in *Batman Returns*. The NERO game and the methods for learning strategies are described by Bryant and Miikkulainen (2007).

Recent book on multiagent systems include those by Weiss (2000a), Young (2004), Vlassis (2008), and Shoham and Leyton-Brown (2009). There is an annual conference on autonomous agents and multiagent systems (AAMAS).

EXERCISES

11.1 The goals we have considered so far all ask the planner to make the world satisfy the goal at just one time step. Not all goals can be expressed this way: you do not achieve the goal of suspending a chandelier above the ground by throwing it in the air. More seriously, you wouldn't want your spacecraft life-support system to supply oxygen one day but not the next. A *maintenance goal* is achieved when the agent's plan causes a condition to hold continuously from a given state onward. Describe how to extend the formalism of this chapter to support maintenance goals.

11.2 You have a number of trucks with which to deliver a set of packages. Each package starts at some location on a grid map, and has a destination somewhere else. Each truck is directly controlled by moving forward and turning. Construct a hierarchy of high-level actions for this problem. What knowledge about the solution does your hierarchy encode?

11.3 Suppose that a high-level action has exactly one implementation as a sequence of primitive actions. Give an algorithm for computing its preconditions and effects, given the complete refinement hierarchy and schemas for the primitive actions.

11.4 Suppose that the optimistic reachable set of a high-level plan is a superset of the goal set; can anything be concluded about whether the plan achieves the goal? What if the pessimistic reachable set doesn't intersect the goal set? Explain.

11.5 Write an algorithm that takes an initial state (specified by a set of propositional literals) and a sequence of HLAs (each defined by preconditions and angelic specifications of optimistic and pessimistic reachable sets) and computes optimistic and pessimistic descriptions of the reachable set of the sequence.

11.6 In Figure 11.2 we showed how to describe actions in a scheduling problem by using separate fields for DURATION, USE, and CONSUME. Now suppose we wanted to combine scheduling with nondeterministic planning, which requires nondeterministic and conditional effects. Consider each of the three fields and explain if they should remain separate fields, or if they should become effects of the action. Give an example for each of the three.

11.7 Some of the operations in standard programming languages can be modeled as actions that change the state of the world. For example, the assignment operation changes the contents of a memory location, and the print operation changes the state of the output stream. A program consisting of these operations can also be considered as a plan, whose goal is given by the specification of the program. Therefore, planning algorithms can be used to construct programs that achieve a given specification.

a. Write an action schema for the assignment operator (assigning the value of one variable to another). Remember that the original value will be overwritten!

b. Show how object creation can be used by a planner to produce a plan for exchanging the values of two variables by using a temporary variable.

11.8 Suppose the $Flip$ action always changes the truth value of variable L. Show how to define its effects by using an action schema with conditional effects. Show that, despite the use of conditional effects, a 1-CNF belief state representation remains in 1-CNF after a $Flip$.

11.9 In the blocks world we were forced to introduce two action schemas, $Move$ and $MoveToTable$, in order to maintain the $Clear$ predicate properly. Show how conditional effects can be used to represent both of these cases with a single action.

11.10 Conditional effects were illustrated for the $Suck$ action in the vacuum world—which square becomes clean depends on which square the robot is in. Can you think of a new set of propositional variables to define states of the vacuum world, such that $Suck$ has an *unconditional* description? Write out the descriptions of $Suck$, $Left$, and $Right$, using your propositions, and demonstrate that they suffice to describe all possible states of the world.

11.11 Find a suitably dirty carpet, free of obstacles, and vacuum it. Draw the path taken by the vacuum cleaner as accurately as you can. Explain it, with reference to the forms of planning discussed in this chapter.

11.12 To the medication problem in the previous exercise, add a $Test$ action that has the conditional effect $CultureGrowth$ when $Disease$ is true and in any case has the perceptual effect $Known(CultureGrowth)$. Diagram a conditional plan that solves the problem and minimizes the use of the $Medicate$ action.