13 PLANNING AND ACTING

In which planning systems must face up to the awful prospect of actually having to take their own advice.

The assumptions required for flawless planning and execution, given the algorithms in the previous chapters, are that the world be accessible, static, and deterministic—just as for our simple search methods. Furthermore, the action descriptions must be correct and complete, describing all the consequences exactly. We described a planning agent in this ideal case in Chapter 11; the resemblance to the simple problem-solving agent of Chapter 3 was no coincidence.

In real-world domains, agents have to deal with both incomplete and incorrect information. Incompleteness arises because the world is inaccessible; for example, in the shopping world, the agent may not know where the milk is kept unless it asks. Incorrectness arises because the world does not necessarily match the agent's model of it; for example, the price of milk may have doubled overnight, and the agent's wallet may have been pickpocketed.

There are two different ways to deal with the problems arising from incomplete and incorrect information:

CONDITIONAL PLANNING

CONTINGENCY

SENSING ACTIONS

◊ **Conditional planning:** Also known as **contingency planning,** conditional planning deals with incomplete information by constructing a conditional plan that accounts for each possible situation or **contingency** that could arise. The agent finds out which part of the plan to execute by including **sensing actions** in the plan to test for the appropriate conditions. For example, the shopping agent might want to include a sensing action in its shopping plan to check the price of some object in case it is too expensive. Conditional planning is discussed in Section 13.1.

EXECUTION MONITORING

REPLANNING

◊ **Execution monitoring:** The simple planning agent described in Chapter 11 executes its plan "with its eyes closed"—once it has a plan to execute, it does not use its percepts to select actions. Obviously this is a very fragile strategy when there is a possibility that the agent is using incorrect information about the world. By monitoring what is happening while it executes the plan, the agent can tell when things go wrong. It can then do **replanning** to find a way to achieve its goals from the new situation. For example, if the agent discovers that it does not have enough money to pay for all the items it has picked up, it can return some and replace them with cheaper versions. In Section 13.2 we look at a simple replanning agent that implements this strategy. Section 13.3 elaborates on this design to provide a full integration of planning and execution.

DEFERRING

Execution monitoring is related to conditional planning in the following way. An agent that builds a plan and then executes it while watching for errors is, in a sense, taking into account the possible conditions that constitute execution errors. Unlike a conditional planner, however, the execution monitoring agent is actually **deferring** the job of dealing with those conditions until they actually arise. The two approaches of course can be combined by planning for some contingencies and leaving others to be dealt with later if they occur. In Section 13.4, we will discuss when one might prefer to use one or the other approach.

13.1 CONDITIONAL PLANNING

We begin by looking at the nature of conditional plans and how an agent executes them. This will help to clarify the relationship between sensing actions in the plan and their effects on the agent's knowledge base. We then explain how to construct conditional plans.

The nature of conditional plans

Let us consider the problem of fixing a flat tire. Suppose we have the following three action schemata:

$Op($ACTION:$Remove(x),$
  PRECOND:$On(x),$
  EFFECT. $Off(x)$A $ClearHub(x)$ A $\neg On(x))$
$Op($ACTION:$PutOn(x),$
  PRECOND:$Off(x)$A $ClearHub(x),$
  EFFECT: $On(x)$ A $\neg ClearHub(x)$ A $\neg Off(x))$
$Op($ ACTION :$Inflate(x),$
  PRECOND:$Intact(x)$ A $Flat(x),$
  EFFECT: $Inflated(x)$A $\neg Flat(x))$

If our goal is to have an inflated tire on the wheel:

$On(x)$ A $Inflated(x)$

and the initial conditions are

$Inflated(Spare)$A $Intact(Spare)$ A $Off(Spare)$ A $On(Tire_1)$ A $Flat(Tire_1)$

then any of the standard planners described in the previous chapters would be able to come up with the following plan:

$[Remove(Tire_1), PutOn(Spare)]$

If the absence of $Intact(Tire_1)$ in the initial state really means that the tire is not intact (as the standard planners assume), then this is all well and good. But suppose we have incomplete knowledge of the world—the tire may be flat because it is punctured, or just because it has not been pumped up lately. Because changing a tire is a dirty and time-consuming business, it would be better if the agent could execute a conditional plan: if $Tire_1$ is intact, then inflate it. If not, remove it and put on the spare. To express this formally, we can extend our original notation for plan steps with a conditional step $If(<Condition>, <ThenPart>, <ElsePart>,)$. Thus, the tire-fixing plan now includes the step

$$If(Intact(Tire_1), [Inflate(Tire_1)], [Remove(Tire_1),\ PutOn(Spare)])$$

Thus, the conditional planning agent can sometimes do better than the standard planning agents described earlier. Furthermore, there are cases where a conditional plan is the only possible plan. If the agent does not know if its spare tire is flat or inflated, then the standard planner will fail, whereas the conditional planner can insert a second conditional step that inflates the spare if necessary. Lastly, if there is a possibility that both tires have holes, then neither planner can come up with a guaranteed plan. In this case, a conditional planner can plan for all the cases where success is possible, and insert a *Fail* action on those branches where no completion is possible.

Plans that include conditional steps are executed as follows. When the conditional step is executed, the agent first tests the condition against its knowledge base. It then continues executing either the then-part or the else-part, depending on whether the condition is true or false. The then-part and the else-part can themselves be plans, allowing arbitrary nesting of conditionals. The conditional planning agent design is shown in Figure 13.1. Notice that it deals with nested conditional steps by following the appropriate conditional branches until it finds a real action to do. (The conditional planning algorithm itself, CPOP, will be discussed later.)

The crucial part of executing conditional plans is that the agent must, at the time of execution of a conditional step, be able to decide the truth or falsehood of the condition—that is, *the condition must be known to the agent* at that point in the plan. If the agent does not know if $Tire_1$ is intact or not, it cannot execute the previously shown plan. What the agent knows at any point is of course determined by the sequence of percepts up to that point, the sequence of actions carried out, and the agent's initial knowledge. In this case, the initial conditions in the knowledge base do not say anything about $Intact(Tire_1)$. Furthermore, the agent may have no actions that cause $Intact(Tire_1)$ to become true.[1] *To ensure that a conditional plan is executable, the agent must insert actions that cause the relevant conditions to become known by the agent.*

Facts become known to the agent through its percepts, so what we mean by the previous remark is that the agent must act in such a way as to make sure it receives the appropriate percepts. For example, one way to come to know that a tire is intact is to put some air into it and place one's listening device in close proximity. A hissing percept then enables the agent to infer that the tire is not intact.[2] Suppose we use the name $CheckTire(x)$ to refer to an action that establishes the state of the tire x. This is an example of a **sensing action.**

Using the situation calculus description of sensing actions described in Chapter 8, we would write

$$\forall x, s \quad Tire(x) \quad \Rightarrow \quad KnowsWhether(\text{``}Intact(\underline{x})\text{''}, Result(CheckTire(x), s))$$

[1] Note that if, for example, a $Patch(Tire_1)$ action were available, then a standard plan could be constructed.

[2] Agents without sound percepts can wet the tire. A bubbling visual percept then suggests the tire is compromised.

```
function CONDITIONAL-PLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          p, a plan, initially NoPlan
          t, a counter, initially 0, indicating time
          G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if p = NoPlan then p ← CPOP(current, G, KB)
  if p = NoPlan or p is empty then action ← NoOp
  else
     action ← FIRST(p)
     while CONDITIONAL?(action) do
        if ASK(KB, CONDITION-PART[action]) then p ← APPEND(THEN-PART[action], REST(p))
        else p ← APPEND(ELSE-PART[action], REST(p))
        action ← FIRST(p)
     end
     p ← REST(p)
  TELL(KB, MAKE-ACTION-SENTENCE(action, 0)
  f ← t + 1
  return action
```

Figure 13.1 A conditional planning agent.

In our action schema format, we would write

Op(ACTION:$CheckTire(x)$,
  PRECOND:$Tire(x)$,
  EFFECT:$KnowsWhether$("$Intact(\underline{x})$"))

Notice that as well as having knowledge effects, a sensing action can have ordinary effects. For example, if the *CheckTire* action uses the water method, then the tire will become wet. Sensing actions can also have preconditions that need to be established. For example, we might need to fetch the pump in order to put some air in the tire in order to check it. *A conditional planner therefore will sometimes create plans that involve carrying out ordinary actions for the purpose of obtaining some needed information.*

An algorithm for generating conditional plans

CONTEXT

The process of generating conditional plans is much like the planning process described in Chapter 11. The main additional construct is the **context** of a step in the plan. A step's context is simply the union of the conditions that must hold in order for the step to be executed—essentially, it describes the "branch" on which the step lies. For example, the action $Inflate(Tire_1)$ in the earlier plan has a context *Intact(Tire\)*. Once it is established that a step has a certain context, then subsequent steps in the plan inherit that context. Because it cannot be the case that two steps with distinct contexts can both be executed, such steps cannot interfere with each other. Contexts are therefore essential for keeping track of which steps can establish or violate the preconditions of which other steps. An example will make this clear.

The flat-tire plan begins with the usual start and finish steps (Figure 13.2). Notice that the finish step has the context *True,* indicating that no assumptions have been made so far.

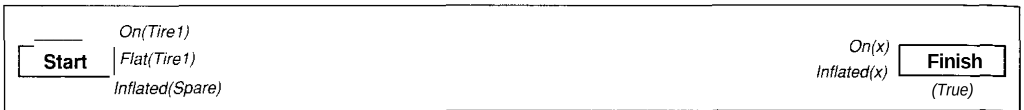

Figure 13.2 Initial plan state for the flat-tire problem.

There are two open conditions to be resolved: *On(x)* and *Inflated*(x). The first is satisfied by adding a link from the start step, with the unifier $\{x/Tire_1\}$. The second is satisfied by adding the step *Inflate*($Tire_1$), which has preconditions *Flat*($Tire_1$) and *Intact*($Tire_1$) (see Figure 13.3).

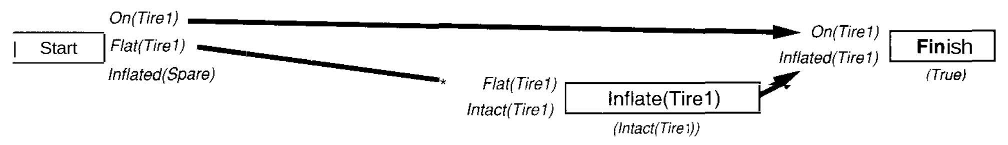

Figure 13.3 Plan after adding the *Inflate*($Tire_1$) step.

The open condition *Flat*($Tire_1$) is satisfied by adding a link from the start step. The interesting part is what to do with the *Intact*($Tire_1$) condition. In the statement of the problem there are no actions that can make the tire intact—that is, no action schema with the effect *Intact*(x). At this point a standard causal-link planner would abandon this plan and try another way to achieve the goal. There is, however, an action *CheckTire*(x) that allows one to *know* the truth value of a proposition that unifies with *Intact*($Tire_1$). If (and this is sometimes a big if) the outcome of checking the tire is that the tire is known to be intact, then the *Inflate*($Tire_1$) step can be achieved. We therefore add the *CheckTire* step to the plan with a **conditional link** (shown as a dotted arrow in Figure 13.4) to the *Inflate*($Tire_1$) step. The *CheckTire* step is called a **conditional step** because it will become a branch point in the final plan. The inflate step and the finish step now acquire a context label stating that they are assuming the outcome *Intact*($Tire_1$) rather than $\neg$*Intact*($Tire_1$). Because *CheckTire* has no preconditions in our simple formulation, the plan is complete *given the context of the finish step.*

CONDITIONAL LINK

CONDITIONAL STEP

Obviously, we cannot stop here. We need a plan that works in both cases. The conditional planner ensures this by adding a second copy of the original finish step, labelled with a context that is the negation of the existing context (see Figure 13.5).[3] In this way, the planner covers an

[3] If the solution of this new branch requires further context assumptions, then a third copy of the finish step will be added whose context is the negation of the disjunction of the existing finish steps. This continues until no more assumptions are needed.

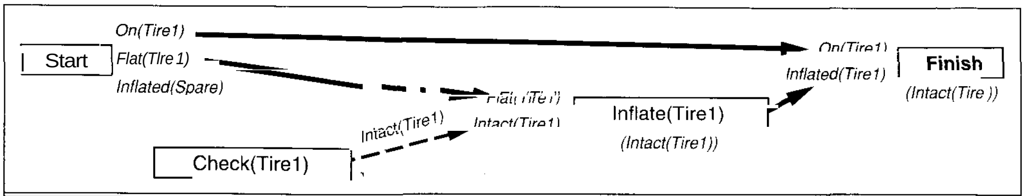

Figure 13.4 The plan to inflate $Tire_1$ will work provided that it is intact.

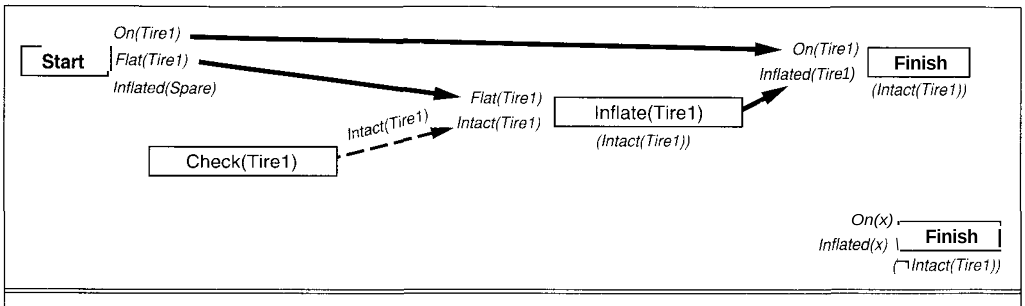

Figure 13.5 We must now plan for the case where $Tire_1$ is not intact.

exhaustive set of possibilities—for every possible outcome, there is a corresponding finish step, and a path to get to the finish step.

Now we need to solve the goal when $Tire_1$ has a hole in it. Here the context is very useful. If we were to try to add the step *Inflate*($Tire_1$) to the plan, we would immediately see that the precondition *Intact*($Tire_1$) is inconsistent with the context ¬*Intact*($Tire_1$). Thus, the only ways to satisfy the *Inflated*(x) condition are to link it to the start step with the unifier $\{x/spare\}$ or to add an *Inflate* step for the spare. Because the latter leads to a dead end (because the spare is not flat), we choose the former. This leads to the plan state in Figure 13.6.

The steps *Remove*($Tire_1$) and *PutOn(Spare)* are now added to the plan to satisfy the condition *On*(*Spare*), using standard causal-link addition. Initially, the steps would have a *True*

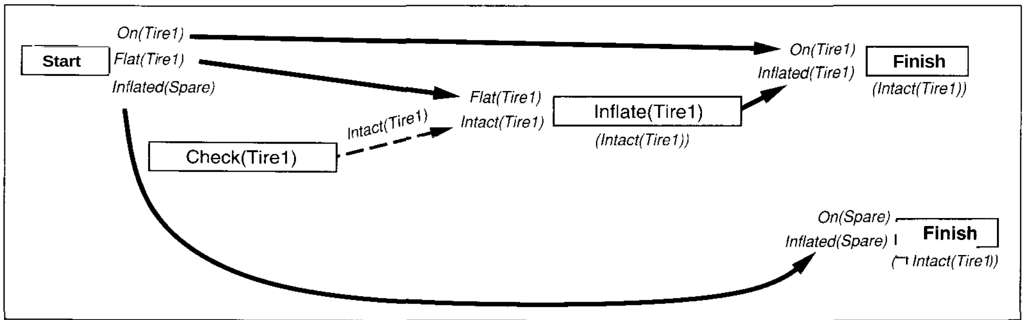

Figure 13.6 When $Tire_1$ is not intact, we must use the spare instead.

context, because it has not yet been established that they can only be executed under certain circumstances. This means that we have to check how the steps interact with other steps in the plan. In particular, the *Remove*($Tire_1$) step threatens the causal link protecting *On*($Tire_1$) in the first finish step (the one with the context (*Intact*($Tire_1$)). In a standard causal-link planner, the only solution would be to promote or demote the *Remove*($Tire_1$) step so that it cannot interfere.

CONDITIONING

In the conditional planner, we can also resolve the threat by **conditioning** the step so that its context becomes incompatible with the context of the step whose precondition it is threatening (in this case, the first finish step). Conditioning is achieved by finding a conditional step that has a possible outcome that would make the threatening step's context incompatible with the causal link's context. In this case, the *CheckTire* step has a possible outcome $\neg$*Intact*($Tire_1$). If we make a conditional link from the *CheckTire* step to the *Remove*($Tire_1$) step, then the remove step is no longer a threat. The new context is inherited by the *PutOn*(*Spare*)step, and the plan is now complete (Figure 13.7).

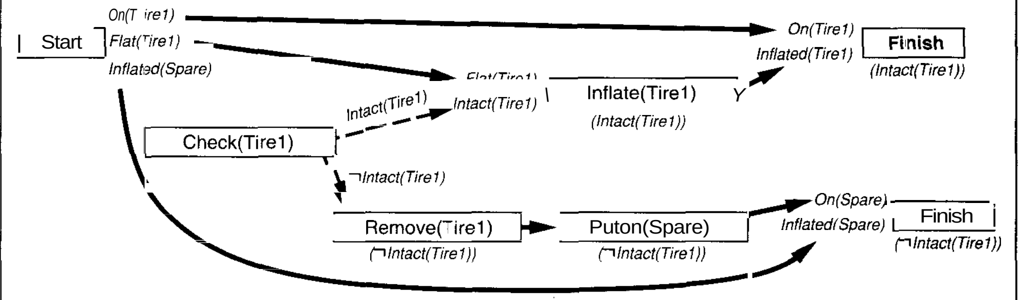

Figure 13.7 The complete plan for fixing the tire, showing causal and conditional links and contexts (in parentheses).

The algorithm is called CPOP (for Conditional Partial-Order Planner). It builds on the POP algorithm, and extends it by incorporating contexts, multiple finish steps and the conditioning process for resolving potential threats. It is shown in Figure 13.8.

Extending the plan language

The conditional steps we used in the previous section had only two possible outcomes. In some cases, however, a sensing action can have any number of outcomes. For example, checking the color of some object might result in a sentence of the form *Color*(x,c) being known for some value of c.

PARAMETERIZED PLANS

Sensing actions of this type can be used in **parameterized plans,** where the exact actions to be carried out will not be known until the plan is executed. For example, suppose *we* have a goal such as

$$Color(Chair,c) \wedge Color(Table,c)$$

```
function CPOP(initial, goals, operators) returns plan
  plan ← MAKE-PLAN(initial, goals)
  loop do
      Termination:
        if there are no unsatisfied preconditions
           and the contexts of the finish steps are exhaustive
             then return plan
      Alternative context generation:
        if the plans for existing finish steps are complete and have contexts $C_1 \ldots C_n$ then
             add a new finish step with a context $\neg (C_1 \lor \ldots \lor C_n)$
             this becomes the current context
      Subgoal selection and addition:
        find a plan step $S_{need}$ with an open precondition $c$
      Action selection:
        choose a step Sadd from operators or STEPS(plan) that adds $c$ or
                  knowledge of $c$ and has a context compatible with the current context
        if there is no such step
             then fail
        add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(plan)
        add $S_{add} < S_{need}$ to ORDERINGS(plan)
        if $S_{add}$ is a newly added step then
             add Sadd to STEPS(plan)
             add Start < Sadd < Finish to ORDERINGS(plan)
      Threat resolution:
        for each step $S_{threat}$ that potentially threatens any causal link $S_i \xrightarrow{c} 5\}$
                   with a compatible context do
             choose one of
                  Promotion: Add $S_{threat} < S_i$ to ORDERINGS(plan)
                  Demotion: Add $S_j < S_{threat}$ to ORDERINGS(plan)
                  Conditioning:
                      find a conditional step $S_{cond}$ possibly before both $S_{threat}$ and $S_j$, where
                           1. the context of $S_{cond}$ is compatible with the contexts of $S_{threat}$ and $S_j$;
                           2. the step has outcomes consistent with $S_{threat}$ and $S_j$, respectively
                      add conditioning links for the outcomes from $S_{cond}$ to $S_{threat}$ and $S_j$
                      augment and propagate the contexts of $S_{threat}$ and $S_j$
             if no choice is consistent
                      then fail
        end
  end
```

Figure 13.8 The CPOP algorithm for constructing conditional plans.

("the chair and table are the same color"). The chair is initially unpainted, and we have some paints and a paintbrush. Then we might use the plan

$$[SenseColor(Table), KnowsWhat(\text{“}Color(Table, \underline{c})\text{”}), GetPaint(c), Paint(Chair, c)]$$

The last two steps are parameterized, because until execution the agent will not know the value of c. We call c a **runtime variable,** as distinguished from normal planning variables whose values are known as soon as the plan is made. The step *SenseColor(Table)* will have the effect of providing percepts sufficient to allow the agent to deduce the color of the table. Then the action $KnowsWhat(\text{“}Color(Table, \underline{c})\text{”})$ is executed simply by querying the knowledge base to establish a value for the variable c. This value can be used in subsequent steps such as $GetPaint(c)$, or in conditional steps such as $If(c = Green, [\ldots], [\ldots])$. Sensing actions are defined just as in the binary-condition case:

RUNTIME VARIABLE

$$Op(\text{ACTION}: SenseColor(x),\\ \quad \text{EFFECT}: KnowsWhat(\text{“}Color(\underline{x}, \underline{c})\text{”}))$$

In situation calculus, the action would be described by

$$\forall x, s \quad \exists c \quad KnowsWhat(\text{“}Color(\underline{x}, \underline{c})\text{”}, Result(SenseColor(x), s))$$

The variables x and c are treated differently by the planner. Logically, c is existentially quantified in the situation calculus representation, and thus must be treated as a Skolem function of the object being sensed and the situation in which it is sensed. In the planner, runtime variables like c unify only with ordinary variables, and not with constants or with each other. This corresponds exactly to what would happen with Skolem functions.

When we have the ability to discover that certain facts *are* true, as well as the ability to cause them to *become* true, then we may wish to have some control over which facts are changed and which are preserved. For example, the goal of having the table and chair the same color can be achieved by painting them both black, regardless of what color the table is at the start. This can be prevented by protecting the table's color so that the agent has to sense it, rather than painting over it. A **maintenance goal** can be used to specify this:

MAINTENANCE GOAL

$$Color(Chair, c) \wedge Color(Table, c) \wedge Maintain(Color(Table, x))$$

The *Maintain* goal will ensure that no action is inserted in the plan that has an ordinary causal effect that changes the color of the table. This is done in a causal-link planner by adding a causal link from the start step to the finish step protecting the table's initial color.

Plans with conditionals start to look suspiciously like programs. Moreover, executing such plans starts to look rather like interpreting a program. The similarity becomes even stronger when we include loops in plans. A loop is like a conditional, except that when the condition holds, a portion of the plan is repeated. For example, we might include a looping step to make sure the chair is painted properly:

$$While(Knows(\text{“}UnevenColor(Chair)\text{”})[Paint(Chair, c), CheckColor(Chair)])$$

Techniques for generating plans with conditionals and loops are almost identical to those for generating programs from logical specifications (so-called **automatic programming).** Even a standard planner can do automatic programming of a simple kind if we encode as STRIPS operators the actions corresponding to assignment statements, procedure calls, printing, and so on.

AUTOMATIC PROGRAMMING

13.2 A SIMPLE REPLANNING AGENT

As long as the world behaves exactly as the action descriptions describe it, then executing a plan in the ideal or incomplete-information cases will always result in goal achievement. As each step is executed, the world state will be as predicted—as long as nothing goes wrong.

"Something going wrong" means that the world state after an action is not as predicted. More specifically, the remaining plan segment will fail if any of its preconditions is not met. The preconditions of a plan segment (as opposed to an individual step) are all those preconditions of the steps in the segment that are not established by other steps in the segment. It is straightforward to annotate a plan at each step with the preconditions required for successful completion of the remaining steps. In terms of the plan description adopted in Chapter 11, the required conditions are just the propositions protected by all the causal links beginning at or before the current step and ending at or after it. Then we can detect a potential failure by comparing the current preconditions with the state description generated from the percept sequence. This is the standard model of execution monitoring, first used by the original STRIPS planner. STRIPS also introduced the **triangle table,** an efficient representation for fully annotated plans.

TRIANGLE TABLE

A second approach is to check the preconditions of each action as it is executed, rather than checking the preconditions of the entire remaining plan. This is called **action monitoring.** As well as being simpler and avoiding the need for annotations, this method fits in well with realistic systems where an individual action failure can be recognized. For example, if a robot agent issues a command to the motor subsystem to move two meters forward, the subsystem can report a failure if the robot bumps into an obstacle that materialized unexpectedly. On the other hand, action monitoring is less effective than execution monitoring, because it does not look ahead to see that an unexpected current state will cause an action failure some time in the future. For example, the obstacle that the robot bumped into might have been knocked off the table by accident much earlier in the plan. An agent using execution monitoring could have realized the problem and picked it up again.

ACTION MONITORING

Action monitoring is also useful when a goal is serendipitously achieved. That is, if someone or something else has already changed the world so that the goal is achieved, action monitoring notices this and avoids wasting time by going through the rest of the plan.

These forms of monitoring require that the percepts provide enough information to tell if a plan or action is about to fail. In an inaccessible world where the relevant conditions are not perceivable, more complicated strategies are needed to cope with undetected but potentially serious deviations from expectations. This issue is beyond the scope of the current chapter.

We can divide the causes of plan failure into two kinds, depending on whether it is possible to anticipate the possible contingencies:

BOUNDED INDETERMINACY

0 **Bounded indeterminacy:** In this case, actions can have unexpected effects, but the possible effects can be enumerated and described as part of the action description axiom. For example, the result of opening a can of paint can be described as the disjunction of having paint available, having an empty can, or spilling the paint. Using a combination of CPOP and the "D" (disjunctive) part of POP-DUNC we can generate conditional plans to deal with this kind of indeterminacy.

UNBOUNDED INDETERMINACY

◇ **Unbounded indeterminacy:** In this case, the set of possible unexpected outcomes is too large to be completely enumerated. This would be the case in very complex and/or dynamic domains such as driving, economic planning, and military strategy. In such cases, we can plan for at most a limited number of contingencies, and must be able to *replan* when reality does not behave as expected.

The next subsection describes a simple method for replanning based on trying to get the plan "back on track" as quickly as possible. Section 13.3 describes a more comprehensive approach that deals with unexpected conditions as an integral part of the decision-making process.

Simple replanning with execution monitoring

One approach to replanning based on execution monitoring is shown in Figure 13.9. The simple planning agent is modified so that it keeps track of both the remaining plan segment p and the complete plan q. Before carrying out the first action of p, it checks to see whether the preconditions of the p are met. If not, it calls CHOOSE-BEST-CONTINUATION to choose some point in the complete plan q such that the plan p' from that point to the end of q is easiest to achieve from the current state. The new plan is to first achieve the preconditions of p' and then execute it.

Consider how REPLANNING-AGENT will perform the task of painting the chair to match the table. Suppose that the motor subsystem responsible for the painting action is imperfect and sometimes leaves small areas unpainted. Then after the *Paint*(*Chair*, *c*) action is done, the execution-monitoring part will check the preconditions for the rest of the plan; the preconditions

```
function REPLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          p, an annotated plan, initially NoPlan
          q, an annotated plan, initially NoPlan
          G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
     p ← PLANNER(current, G, KB)
     q ← p
     if p = NoPlan or p is empty then return NoOp
  if PRECONDITIONS(p) not currently true in KB then
     p' ← CHOOSE-BEST-CONTINUATION(current, q)
     p ← APPEND(PLANNER(current, PRECONDITIONS(p'), KB), p')
     q ← p
  action ← FIRST(p)
  p ← REST(p)
  return action
```

Figure 13.9 An agent that does execution monitoring and replanning.

are just the goal conditions because the remaining plan p is now empty, and the agent will detect that the chair is not all the same color as the table. Looking at the original plan q, the current state is identical to the precondition before the chair-painting step, so the agent will now try to paint over the bare spots. This behavior will cycle until the chair is completely painted.

Suppose instead that the agent runs out of paint during the painting process. This is not envisaged by the action description for *Paint,* but it will be detected because the chair will again not be completely painted. At this point, the current state matches the precondition of the plan beginning with *GetPaint*, so the agent will go off and get a new can of paint before continuing.

Consider again the agent's behavior in the first case, as it paints and repaints the chair. Notice that the *behavior* is identical to that of a conditional planning agent running the looping plan shown earlier. *The difference lies in the time at which the computation is done and the information is available to the computation process.* The conditional planning agent reasons explicitly about the possibility of uneven paint, and prepares for it even though it may not occur. The looping behavior results from a looping plan. The replanning agent assumes at planning time that painting succeeds, but during execution checks on the results and plans just for those contingencies that actually arise. The looping behavior results not from a looping plan but from the interaction between action failures and a persistent replanner.

We should mention the question of **learning** in response to failed expectations about the results of actions. Consider a plan for the painting agent that includes an action to open a door (perhaps to the paint store). If the door sticks a little, the replanning agent will try again until the door opens. But if the door is locked, the agent has a problem. Of course, if the agent already knows about locked doors, then *Unlocked* will be a precondition of opening the door, and the agent will have inserted a *ChecklfLocked* action that observes the state of the door, and perhaps a conditional branch to fetch the key. But if the agent does not know about locked doors, it will continue pulling on the door indefinitely. What we would like to happen is for the agent to learn that its action description is wrong; in this case, there is a missing precondition. We will see how this kind of learning can take place in Chapter 21.

13.3 FULLY INTEGRATED PLANNING AND EXECUTION

SITUATED PLANNING AGENT

In this section, we describe a more comprehensive approach to plan execution, in which the planning and execution processes are fully integrated. Rather than thinking of the planner and execution monitor as separate processes, one of which passes its results to the other, we can think of them as a single process in a **situated planning agent**.[4] The agent is thought of as always being *part of the way through* executing a plan—the grand plan of living its life. Its activities include executing some steps of the plan that are ready to be executed; refining the plan to resolve any of the standard deficiencies (open conditions, potential clobbering, and so on); refining the plan in the light of additional information obtained during execution; and fixing the

[4] The word "situated," which became popular in AI in the late 1980s, is intended to emphasize that the process of deliberation takes place in an agent that is directly connected to an environment. In this book all the agents are "situated," but the situated planning agent integrates deliberation and action to a greater extent than some of the other designs.

plan in the light of unexpected changes in the environment, which might include recovering from execution errors or removing steps that have been made redundant by serendipitous occurrences. Obviously, when it first gets a new goal the agent will have no actions ready to execute, so it will spend a while generating a partial plan. It is quite possible, however, for the agent to begin execution before the plan is complete, especially when it has independent subgoals to achieve. The situated agent continuously monitors the world, updating its world model from new percepts even if its deliberations are still continuing.

As in the discussion of the conditional planner, we will first go through an example and then give the planning algorithm. We will keep to the formulation of steps and plans used by the partial-order planner POP, rather than the more expressive languages used in Chapter 12. It is, of course, possible to incorporate more expressive languages, as well as conditional planning techniques, into a situated planner.

The example we will use is a version of the blocks world. The start state is shown in Figure 13.10(a), and the goal is *On(C,D)* A *On(D,B)*. The action we will need is *Move(x,y)*, which moves block *x* onto block *y*, provided both are clear. Its action schema is

Op(ACTION:*Move*(*x*, *y*),
  PRECOND:*Clear*(*x*) A *Clear*(*y*) A *On*(*x*, *z*),
  EFFECT:*On*(*x*, *y*) A *Clear*(*z*) A $\neg$*On*(*x*, *z*)$\wedge$ /*Clear*(*y*))

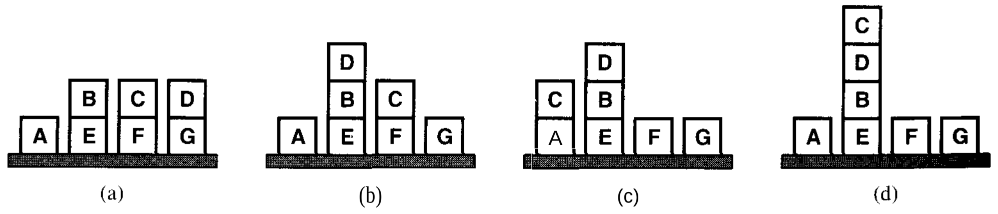

Figure 13.10 The sequence of states as the situated planning agent tries to reach the goal state *On(C, D)* A *On(D,B)* as shown in (d). The start state is (a). At (b), another agent has interfered, putting *D* on *B*. At (c), the agent has executed *Move(C,D)* but has failed, dropping C on A instead. It retries *Move(C,D)*, reaching the goal state (d).

The agent first constructs the plan shown in Figure 13.11. Notice that although the preconditions of both actions are satisfied by the initial state, there is an ordering constraint putting *Move(D,B)* before *Move(C,D)*. This is needed to protect the condition *Clear(D)* until *Move(D,B)* is completed.

At this point, the plan is ready to be executed, but nature intervenes. An external agent moves *D* onto *B* (perhaps the agent's teacher getting impatient), and the world is now in the state shown in Figure 13.10(b). Now *Clear(B)* and *On(D,G)* are no longer true in the initial state, which is updated from the new percept. The causal links that were supplying the preconditions *Clear(B)* and *On(D, G)* for the *Move(D,B)* action become invalid, and must be removed from the plan. The new plan is shown in Figure 13.12. Notice that two of the preconditions for *Move(D,B)*

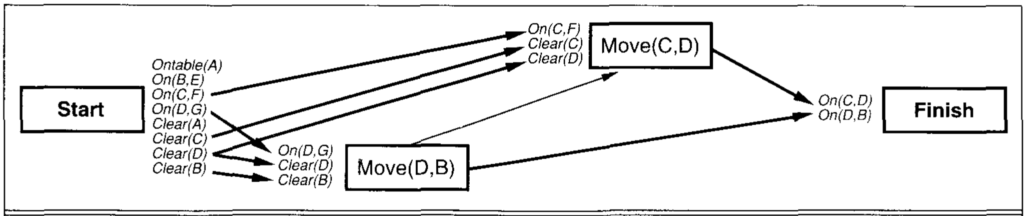

Figure 13.11 The initial plan constructed by the situated planning agent. The plan is indistinguishable, so far, from that produced by a normal partial-order planner.

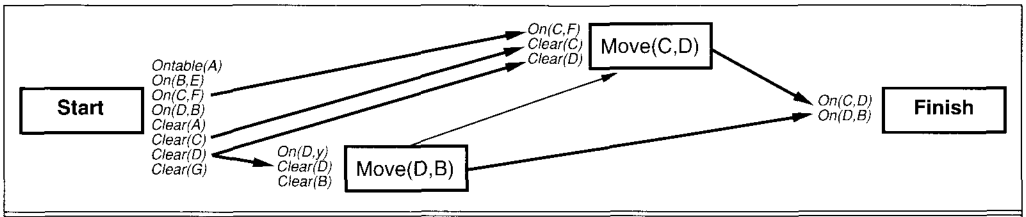

Figure 13.12 After someone else moves D onto B, the unsupported links protecting *Clear*(B) and *On*(D, G) are dropped, producing this plan.

are now open, and the precondition *On*(D, y)is now uninstantiated because there is no reason to assume the move will be from G any more.

Now the agent can take advantage of the "helpful" interference by noticing that the causal link protecting *On*(D,B) and supplied by *Move*(D,B) can be replaced by a direct link from START.

EXTENDING

This process is called **extending** a causal link, and is done whenever a condition can be supplied by an earlier step instead of a later one without causing a new threat.

REDUNDANT STEP

Once the old link from *Move*(D,B) is removed, the step no longer supplies any causal links at all. **It** is now a **redundant step.** All redundant steps are dropped from the plan, along with any links supplying them. This gives us the plan shown in Figure 13.13.

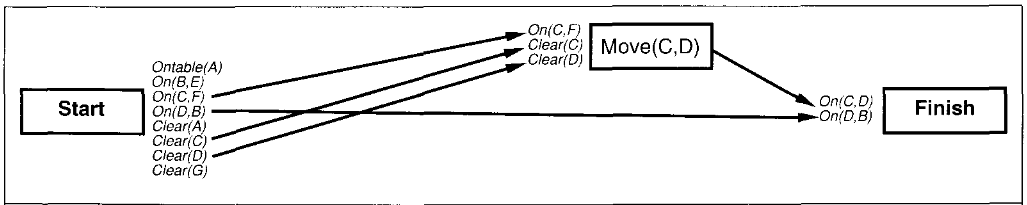

Figure 13.13 The link supplied by *Move*(D, B)has been replaced by one from START, and the now-redundant step *Move*(D,B) has been dropped.

Now the step *Move*(C,D) is ready to be executed, because all of its preconditions are satisfied by the START step, no other steps are necessarily before it, and it does not threaten any other link in the plan. The step is removed from the plan and executed. Unfortunately, the agent is clumsy and drops C onto A instead of D, giving the state shown in Figure 13.10(c). The new plan state is shown in Figure 13.14. Notice that although there are now no actions in the plan, there is still an open condition for the FINISH step.

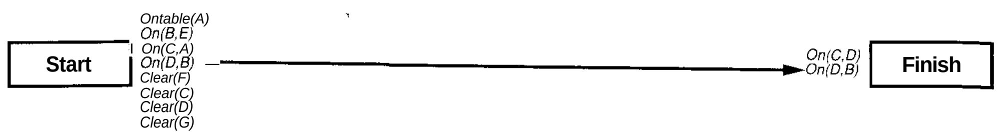

Figure 13.14 After *Move(C,D)* is executed and removed from the plan, the effects of the START step reflect the fact that *C* ended up on *A* instead of the intended *D*. The goal precondition *On(C, D)* is still open.

The agent now does the same planning operations as a normal planner, adding a new step to satisfy the open condition. Once again, *Move(C,D)* will satisfy the goal condition. Its preconditions are satisfied in turn by new causal links from the START step. The new plan appears in Figure 13.15.

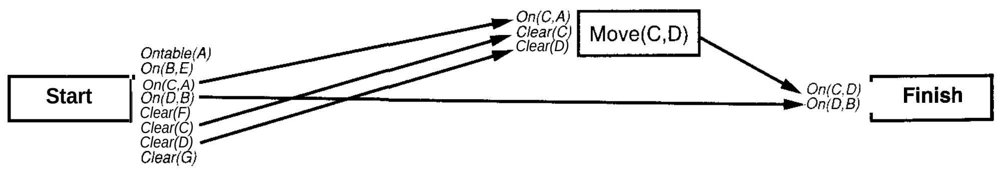

Figure **13.15** The open condition is resolved by adding *Move(C,D)* back in. Notice the new bindings for the preconditions.

Once again, *Move(C,D)* is ready for execution. This time it works, resulting in the goal state shown in Figure 13.10(d). Once the step is dropped from the plan, the goal condition *On(C, D)* becomes open again. Because the START step is updated to reflect the new world state, however, the goal condition can be satisfied immediately by a link from the START step. This is the normal course of events when an action is successful. The final plan state is shown in Figure 13.16. Because all the goal conditions are satisfied by the START step and there are no remaining actions, the agent resets the plan and looks for something else to do.

The complete agent design is shown in Figure 13.17 in much the same form as used for POP and CPOP, although we abbreviate the part in common (resolving standard flaws). One significant structural difference is that planning and acting are the same "loop" as implemented by the coupling between agent and environment. After each plan modification, an action is returned (even if it is a *NoOp)* and the world model is updated from the new percept. We assume that each action finishes executing before the next percept arrives. To allow for extended execution with a completion signal, an executed action must remain in the plan until it is completed.

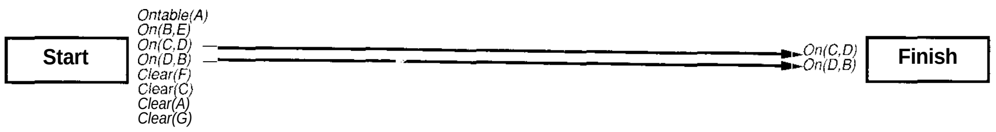

Figure 13.16 After $Move(C,D)$ is executed and dropped from the plan, the remaining open condition *On(C,D)* is resolved by adding a causal link from the new START step. The plan is now completed.

13.4 DISCUSSION AND EXTENSIONS

We have arrived at an agent design that addresses many of the issues arising in real domains:

- The agent can use explicit domain descriptions and goals to control its behavior.
- By using partial-order planning, it can take advantage of problem decomposition to deal with complex domains without necessarily suffering exponential complexity.
- By using the techniques described in Chapter 12, it can handle domains involving conditional effects, universally quantified effects, object creation and deletion, and ramifications. It can also use "canned plans" to achieve subgoals.
- It can deal with errors in its domain description, and, by incorporating conditional planning, it can plan to obtain information when more is needed.
- It can deal with a dynamically changing world by incrementally fixing its plan as it detects errors and unfulfilled preconditions.

Clearly, this is progress. It is, however, a good idea to examine each advance in capabilities and try to see where it breaks down.

Comparing conditional planning and replanning

Looking at conditional planning, we see that almost all actions in the real world have a variety of possible outcomes besides the expected outcome. The number of possible conditions that must be planned for grows exponentially with the number of steps in the plan. Given that only one set of conditions will actually occur, this seems rather wasteful as well as impractical. Many of the events being planned for have only an infinitesimal chance of occurring.

Looking at replanning, we see that the planner is basically assuming no action failures, and then fixing problems as they arise during execution. This too has its drawbacks. The planner may produce very "fragile" plans, which are very hard to fix if anything goes wrong. For example, the entire existence of "spare tires" is a result of conditional planning rather than replanning. If the agent does not plan for a puncture, then it will not see the need for a spare tire. Unfortunately, without a spare tire, even the most determined replanning agent might be faced with a long walk.

```
function SITUATED-PLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          p, a plan, initially NoPlan
          t, a counter, initially 0, indicating time
          G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  EFFECTS(START(p)) — current
  if p = NoPlan then
     G ← ASK(KB, MAKE-GOAL-QUERY(t))
     p ← MAKE-PLAN(current, G, KB)
  action ← NoOp (the default)

   Termination:
     if there are no open preconditions and p has no steps other than START and FTNISH then
        p ←NoPlan and skip remaining steps

   Resolving standard flaws:
     resolve any open condition by adding a causal link from any existing
        possibly prior step or a new step
     resolve potential threats by promotion or demotion

   Remove unsupported causal links:
     if there is a causal link START $\xrightarrow{c}$ S protecting a proposition c
       that no longer holds in START then
        remove the link and any associated bindings

   Extend causal links back to earliest possible step:
     if there is a causal link $S_j \xrightarrow{c} S_k$ such that
       another step $S_i$ exists with $S_i < S_j$ and the link $S_i \xrightarrow{c} S_k$ is safe then
        replace 5) $\xrightarrow{c}$ $S_k$ with $S_i \xrightarrow{c} S_k$

   Remove redundant actions:
     remove any step S that supplies no causal links

   Execute actions when ready for execution:
     if a step S in the plan other than FINISH satisfies the following:
        (a) all preconditions satisfied by START;
        (b) no other steps necessarily between START and S; and
        (c) S does not threaten any causal link in p then
           add ordering constraints to force all other steps after S
           remove S from p, and all causal links to and from S
           action ← the action in 5

  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t — t+ 1
  return action
```

Figure 13.17 A situated planning agent.

Conditional planning and replanning are really two extremes of a continuous spectrum. One way to construct intermediate systems is to specify disjunctive outcomes for actions where more than one outcome is reasonably likely. Then the agent can insert a sensing action to see which outcome occurred and construct a conditional plan accordingly. Other contingencies are dealt with by replanning. Although this approach has its merits, it requires the agent designer to decide which outcomes need tb be considered. This also means that the decision must be made once for each action schema, rather than depending on the particular context of the action. In the case of the provision of spare tires, for example, it is clear that the decision as to which contingencies to plan for depends not just on the likelihood of occurrence—after all, punctures are quite rare—but also on the cost of an action failure. An unlikely condition needs to be taken into account if it would result in catastrophe (e.g., a puncture when driving across a remote desert). Even if a conditional plan can be constructed, it might be better to plan around the suspect action altogether (e.g., by bringing two spare tires or crossing the desert by camel).

What all this suggests is that when faced with a complex domain and incomplete and incorrect information, the agent needs a way to assess the likelihoods and costs of various outcomes. Given this information, it should construct a plan that maximizes the probability of success and minimizes costs, while ignoring contingencies that are unlikely or are easy to deal with. Part V of this book deals with these issues in depth.

Coercion and abstraction

Although incomplete and incorrect information is the normal situation in real domains, there are techniques that still allow an agent to make quite complex, long-range plans without requiring the full apparatus of reasoning about likelihoods.

COERCION

The first method an agent can apply is **coercion,** which reduces uncertainty about the world by forcing it into a known state regardless of the initial state. A simple example is provided by the table-painting problem. Suppose that some aspects of the world are permanently inaccessible to the agent's senses—for example, it may have only a black and white camera. In this case, the agent can pick up a can of paint and paint both the chair and the table from the same can. This achieves the goal and reduces uncertainty. Furthermore, if the agent can read the label on the can, it will even know the color of the chair and table.

A second technique is **abstraction.** Although we have discussed abstraction as a tool for handling complexity (see Chapter 12), it also allows the agent to ignore details of a problem about which it may not have exact and complete knowledge. For example, if the agent is currently in London and plans to spend a week in Paris, it has a choice as to whether to plan the trip at an abstract level (fly out on Sunday, return the following Saturday) or a detailed level (take flight BA 216 and then taxi number 13471 via the Boulevard Peripherique). At the abstract level, the agent has actions such as *Fly*(*London*,*Paris)* that are reasonably certain to work. Even with delays, oversold flights, and so on, the agent will still get to Paris. At the detailed level, there is missing information (flight schedules, which taxi will turn up, Paris traffic conditions) and the possibility of unexpected situations developing that would lead to a particular flight being missed.

AGGREGATION

Aggregation is another useful form of abstraction for dealing with large numbers of objects. For example, in planning its cash flows, the U.S. Government assumes a certain number of taxpayers will send in their tax returns by any given date. At the level of individual taxpayers, there is almost complete uncertainty, but at the aggregate level, the system is reliable. Similarly, in trying to pour water from one bottle into another using a funnel, it would be hopeless to plan the path of each water molecule because of uncertainty as well as complexity, yet the pouring as an aggregated action is very reliable.

The discussion of abstraction leads naturally to the issue of the connection between plans and physical actions. Our agent designs have assumed that the actions returned by the planner (which are concrete instances of the actions described in the knowledge base) are directly executable in the environment. A more sophisticated design might allow planning at a higher level of abstraction and incorporate a "motor subsystem" that can take an action from the plan and generate a sequence of primitive actions to be carried out. The subsystem might, for example, generate a speech signal from an utterance description returned by the planner; or it might generate stepper-motor commands to turn the wheels of a robot to carry out a "move" action in a motion plan. We discuss the connection between planning and motor programs in more detail in Chapter 25. Note that the subsystem might itself be a planner of some sort; its goal is to find a sequence of lower-level actions that achieve the effects of the higher-level action specified in the plan. In any case, the actions generated by the higher-level planner must be capable of execution independently of each other, because the lower-level motor system cannot allow for interactions or interleavings among the subplans that implement different actions.

13.5 SUMMARY

The world is not a tidy place. When the unexpected or unknown occurs, an agent needs to do something to get back on track. This chapter shows how conditional planning and replanning can help an agent recover.

- Standard planning algorithms assume complete and correct information. Many domains violate this assumption.
- Incomplete information can be dealt with using sensing actions to obtain the information needed. **Conditional plans** include different subplans in different contexts, depending on the information obtained.
- Incorrect information results in unsatisfied preconditions for actions and plans. **Execution monitoring** detects violations of the preconditions for successful completion of the plan. **Action monitoring** detects actions that fail.
- A simple **replanning agent** uses execution monitoring and splices in subplans as needed.
- A more comprehensive approach to plan execution involves incremental modifications to the plan, including execution of steps, as conditions in the environment evolve.
- **Abstraction** and **coercion** can overcome the uncertainty inherent in most real domains.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Early planners, which lacked conditionals and loops, sometimes resorted to a coercive style in response to environmental uncertainty. Sacerdoti's NOAH used coercion in its solution to the "keys and boxes" problem, a planning challenge problem in which the planner knows little about the initial state. Mason (1993) argued that sensing often can and should be dispensed with in robotic planning, and describes 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.

WARPLAN-C (Warren, 1976), a variant of WARPLAN, was one of the earliest planners to use conditional actions. Olawski and Gini (1990) lay out the major issues involved in conditional planning. Recent systems for partial-order conditional planning include UWL (Etzioni *et al.*, 1992) and CNLP (Peot and Smith, 1992), on which CPOP is based). C-BURIDAN (Draper *et al.*, 1994) handles conditional planning for actions with probabilistic outcomes, thereby connecting to the work on Markov decision problems described in Chapter 17.

There is a close relation between conditional planning and automated program synthesis, for which there are a number of references in Chapter 10. The two fields have usually been pursued separately because of the enormous difference in typical cost between execution of machine instructions and execution of actions by robot vehicles or manipulators. Linden (1991) attempts explicit cross-fertilization between the two fields.

The earliest major treatment of execution monitoring was PLANEX (Fikes *et al.*, 1972), which worked with the STRIPS planner to control the robot Shakey. PLANEX used triangle tables to allow recovery from partial execution failure without complete replanning. Shakey's model of execution is discussed further in Chapter 25. 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. It used theorem proving to reason about these complex actions. IPEM (Integrated Planning, Execution, and Monitoring) (Ambros-Ingerson and Steel, 1988), which was the first system to smoothly integrate partial-order planning and planning execution, forms the basis for the discussion in this chapter.

A system that contains an explicitly represented agent function, whether implemented as a table or a set of condition-action rules, need not worry about unexpected developments in the environment. All it has to do is to execute whatever action its function recommends for the state in which it finds itself (or in the case of inaccessible environments, the percept sequence to date). The field of **reactive planning** aims to take advantage of this fact, thereby avoiding the complexities of planning in dynamic, inaccessible environments. "Universal plans" (Schoppers, 1987) were developed as a scheme for reactive planning, but turned out to be a rediscovery of the idea of **policies** in Markov decision processes. Brooks's (1986) subsumption architecture (also discussed in Chapter 25) uses a layered finite state machine to represent the agent function, and stresses the use of minimal internal state. Another important manifestation of the reactive planning paradigm is Pengi (Agre and Chapman, 1987), designed as a response to the criticism of classical AI planning in Chapman (1987). Ginsberg (1989) made a spirited attack on reactive planning, including intractability results for some formulations of the reactive planning problem. For an equally spirited response, see Schoppers (1989).

REACTIVE PLANNING

EXERCISES

13.1 Consider how one might use a planning system to play chess.

a. Write action schemata for legal moves. Make sure to include in the state description some way to indicate whose move it is. Will basic STRIPS actions suffice?
b. Explain how the opponent's moves can be handled by conditional steps.
c. Explain how the planner would represent and achieve the goal of winning the game.
d. How might we use the planner to do a finite-horizon lookahead and pick the best move, rather than planning for outright victory?
e. How would a replanning approach to chess work? What might be an appropriate way to combine conditional planning and replanning for chess?

13.2 Discuss the application of conditional planning and replanning techniques to the vacuum world and wumpus world.

13.3 Represent the actions for the flat-tire domain in the appropriate format, formulate the initial and goal state descriptions, and use the POP algorithm to solve the problem.

13.4 This exercise involves the use of POP to *actually* fix a flat tire (in simulation).

a. Build an environment simulator for the flat-tire world. Your simulator should be able to update the state of the environment according to the actions taken by the agent. The easiest way to do this is to take the postconditions directly from the operator descriptions and use TELL and RETRACT to update a logical knowledge base representing the world state.
b. Implement a planning agent for your environment, and show that it fixes the tire.

13.5 In this exercise, we will add nondeterminism to the environment from Exercise 13.4.

a. Modify your environment so that with probability 0.1, an action fails—that is, one of the effects does not occur. Show an example of a plan not working because of an action failure.
b. Modify your planning agent to include a simple replanning capability. It should call POP to construct a repair plan to get back to the desired state along the solution path, execute the repair plan (calling itself recursively, of course, if the repair plan fails), and then continue executing the original plan from there. (You may wish to start by having failed actions do nothing at all, so that this recursive repair method automatically results in a "loop-until-success" behavior; this will probably be easier to debug!)
c. Show that your agent can fix the tire in this new environment.

13.6 **Softbots** construct and execute plans in software environments. One typical task for softbots is to find copies of technical reports that have been published at some other institution. Suppose that the softbot is given the task "Get me the most recent report by X on topic Y." Relevant actions include logging on to a library information system and issuing queries, using an Internet directory to find X's institution, sending email to X; connecting to X's institution by `ftp`, and so on. Write down formal representations for a representative set of actions, and discuss what sort of planning and execution algorithms would be needed.