Chapter 17
Planning Based on Model Checking

17.1 Introduction

Planning by model checking is an approach to planning under uncertainty that deals with nondeterminism, partial observability, and extended goals. Its key idea is to solve planning problems model-theoretically. It is based on the following conventions.

- A planning domain is a nondeterministic state-transition system, where an action may lead from the same state to many different states. The planner does not know which of the outcomes will actually take place when the action will be executed.
- Formulas in temporal logic express reachability goals, i.e., a set of final desired states, as well as temporal goals with conditions on the entire plan execution paths. They can express requirements of different strengths that take into account nondeterminism.
- Plans result in conditional and iterative behaviors,[1] and in general they are strictly more expressive than plans that simply map states to actions to be executed.
- Given a state-transition system and a temporal formula, planning by model checking generates plans that "control" the evolution of the system so that all of the system's behaviors make the temporal formula true. Plan validation can be formulated as a model checking problem (see Appendix C).
- Planning algorithms can use symbolic model checking techniques (see Appendix C). In particular, sets of states are represented as propositional formulas, and searching through the state space is performed by doing logical transformations over propositional formulas. The algorithms can be

1. They are similar to programs with conditional statements and loops.

implemented by using symbolic techniques taking advantage of ordered binary decision diagrams (BDDs; see Appendix C), which allow for the compact representation and effective manipulation of propositional formulas.

The main advantage of planning by model checking is the ability to plan under uncertainty in a practical way. Nondeterminism leads to the need to deal with different action outcomes, i.e., different possible transitions. Partial observability leads to the need to deal with observations that correspond to more than a single state. Planning by model checking searches sets of states and sets of transitions at once, rather than single states. These sets are represented and manipulated symbolically: extremely large sets of states or huge sets of transitions are often represented very compactly, and in some cases they are manipulated at a low computational cost. Most often indeed, algorithms for planning by model checking do not degrade performances at increasing uncertainty; on the contrary, they are faster the more uncertain the domain is. Last but not least, the approach can be extended to deal with complex goals, expressing temporal conditions and requirements of different strengths. Even in this case, the idea of working on sets of states seems to be feasible and practical: symbolic algorithms for temporally extended goals do not degrade significantly with respect to those for reachability goals.

In conclusion, planning by model checking has great potential to deal with uncertainty in a general, well-founded, and practical way. As a consequence, the approach is in principle good for all applications where uncertainty is critical and nonnominal behaviors are relevant, such as safety-critical applications. The approach seems promising also in those cases where models cannot avoid uncertainty, for instance, the case of planning for web services.

This chapter describes planning by model checking under the assumption of full observability for reachability goals (Section 17.2) and for extended goals (Section 17.3), and planning under partial observability (Section 17.4). In Section 17.5 we discuss planning based on model checking versus MDPs. The chapter ends with discussion and exercises.

17.2 Planning for Reachability Goals

Reachability goals intuitively express conditions on the final state of the execution of a plan: we want a plan that, when executed, reaches a state that satisfies some condition, i.e., the final state is a goal state. The notion of solution for a reachability goal in classical planning, where domains are deterministic, is clear because the execution of a plan corresponds to a unique sequence of states: the final state must be a goal state. In the case of nondeterminism, the execution of a given plan may result, in general, in more than one sequence of states. Therefore, the solution to a reachability goal should be characterized with respect to the many possible executions of a plan, e.g., all the executions or just some of them can reach a goal state. In Section 17.2.1, we define precisely the three possible notions of solutions to a planning problem with a reachability goal, while in Section 17.2.2 we describe some planning algorithms that generate plans for these three different kinds of planning problems.

17.2.1 Domains, Plans, and Planning Problems

A *planning domain* is a nondeterministic state-transition system $\Sigma = (S, A, \gamma)$, where:

- S is a finite set of states.
- A is a finite set of actions.
- $\gamma : S \times A \rightarrow 2^S$ is the state-transition function.

Nondeterminism is modeled by γ: given a state s and an action a, $\gamma(s, a)$ is a set of states. We say that an action a is applicable in a state s if $\gamma(s, a)$ is not empty. The set of actions that are applicable in state s is $A(s) = \{a : \exists s' \in \gamma(s, a)\}$.

Example 17.1 Figure 17.1 shows a nondeterministic state-transition system for a simplified DWR domain. It is the same example as the one in Figure 16.1 but without probabilities. In this example, a robot can move among the five different locations

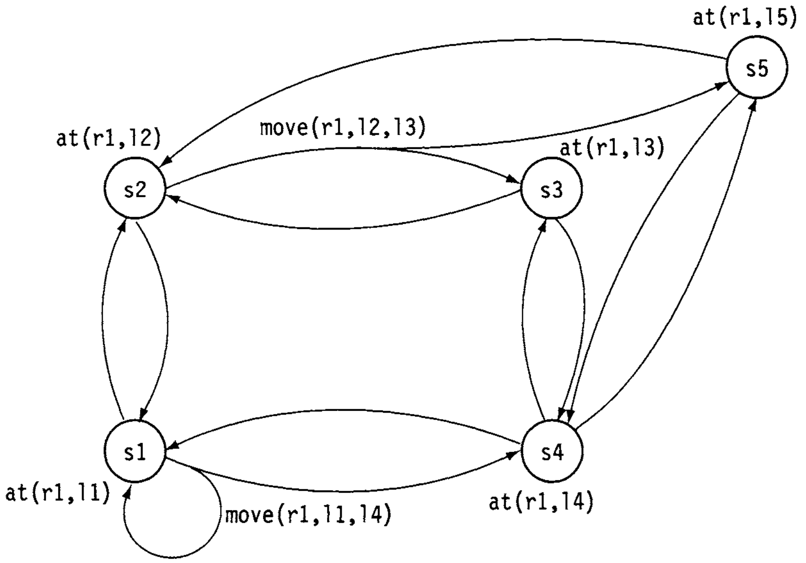

Figure 17.1 A nondeterministic state-transition system. There are five states (s1, s2, s3, s4, s5), one for each location (l1, l2, l3, l4, l5). An action, e.g., move(r1,l1,l2) labels the arc, e.g., s1 to s2. There are two nondeterministic actions, move(r1,l2,l3) and move(r1,l1,l4).

l1, l2, l3, l4, and l5, corresponding to the states s1, s2, s3, s4, and s5, respectively. There are two sources of nondeterminism: γ(s2, move(r1,l2,l3)) = {s3, s5} and γ(s1, move(r1,l1,l4)) = {s1, s4}. ■

As in MDP planning, we need to generate plans that encode conditional and iterative behaviors. Here plans are policies that are similar to MDP policies. A *policy* π for a planning domain $\Sigma = (S, A, \gamma)$ is a set of pairs (s, a) such that $s \in S$ and $a \in A(s)$. We require that for any state s there is at most one action a such that $(s, a) \in \pi$. The set of states of a policy is $S_\pi = \{s \mid (s, a) \in \pi\}$.

Similarly to MDPs, a controller uses a reactive loop to execute a policy (see Figure 17.2). The differences compared with MDP policies are that here policies are not necessarily defined over all S ($S_\pi \subseteq S$). In the following discussion, we use interchangeably the terms *policy* and *plan.*

Example 17.2 Consider the following policies for the domain in Figure 17.1.

$$\pi_1 = \{(\mathsf{s1}, \mathsf{move(r1, l1, l2)}) \quad (\mathsf{s2}, \mathsf{move(r1, l2, l3)}) \quad (\mathsf{s3}, \mathsf{move(r1, l3, l4)})\}$$

$$\pi_2 = \{(\mathsf{s1}, \mathsf{move(r1, l1, l2)}) \quad (\mathsf{s2}, \mathsf{move(r1, l2, l3)}) \quad (\mathsf{s3}, \mathsf{move(r1, l3, l4)}) \quad (\mathsf{s5}, \mathsf{move(r1, l5, l4)})\}$$

$$\pi_3 = \{(\mathsf{s1}, \mathsf{move(r1, l1, l4)})\}$$

Policies π_1, π_2, and π_3 are different strategies for going from state s1 to s4. The main difference between π_1 and π_2 is that the latter takes into account the possibility that moving from location l2 to l3 may lead the robot to l5. If the robot goes to l5, then π_1 does not say what to do, but π_2 says to move the robot from l5 to l4. π_3 is defined only in the state where the robot is at location l1 and tries to move the robot to l4. This action may leave the robot at location l1, in which case π_3 repeats action move(r1,l1,l4), thus encoding an iterative plan. While Execute-Policy(π_1) and Execute-Policy(π_2) are guaranteed to terminate, Execute-Policy(π_3) may not terminate if moving the robot never leads to location l4. ■

```
Execute-Policy($\pi$)
    observe the current state $s$
    while $s \in S_\pi$ do
        select an action $a$ such that $(s, a) \in \pi$
        execute action $a$
        observe the current state $s$
end
```

Figure 17.2 Policy execution.

We represent the execution of a policy in a planning domain with an *execution structure*, i.e., a directed graph in which the nodes are all of the states of the domain that can be reached by executing actions in the policy, and the arcs represent possible state transitions caused by actions in the policy. Let π be a policy of a planning domain $\Sigma = (S, A, \gamma)$. The *execution structure* induced by π from the set of initial states $S_0 \subseteq S$ is a pair $\Sigma_\pi = (Q, T)$, with $Q \subseteq S$, $T \subseteq S \times S$, such that $S_0 \subseteq Q$, and for every $s \in Q$ if there exists an action a such that $(s, a) \in \pi$, then for all $s' \in \gamma(s, a)$, $s' \in Q$ and $T(s, s')$. A state $s \in Q$ is a *terminal state* of Σ_π if there is no $s' \in Q$ such that $T(s, s')$.

Let $\Sigma_\pi = (Q, T)$ be the execution structure induced by a policy π from S_0. An *execution path* of Σ_π from $s_0 \in S_0$ is a possibly infinite sequence $s_0, s_1, s_2, \ldots$ of states in Q such that, for every state s_i in the sequence, either s_i is the last state of the sequence (in which case s_i is a terminal state of Σ_π) or $T(s_i, s_{i+1})$ holds. We say that a state s' is *reachable from* a state s if there is a path from s to s'.

Example 17.3 Figure 17.3 shows the execution structures induced by π_1, π_2, and π_3 from $S_0 = \{\mathtt{s1}\}$ in the planning domain of Figure 17.1. The execution paths of π_1 are $\langle \mathtt{s1}, \mathtt{s2}, \mathtt{s3}, \mathtt{s4} \rangle$ and $\langle \mathtt{s1}, \mathtt{s2}, \mathtt{s5} \rangle$, and those of π_2 are $\langle \mathtt{s1}, \mathtt{s2}, \mathtt{s3}, \mathtt{s4} \rangle$ and $\langle \mathtt{s1}, \mathtt{s2}, \mathtt{s5}, \mathtt{s4} \rangle$. Some of the infinitely many execution paths of π_3 are $\langle \mathtt{s1}, \mathtt{s4} \rangle$, $\langle \mathtt{s1}, \mathtt{s1}, \mathtt{s4} \rangle$, $\langle \mathtt{s1}, \mathtt{s1}, \mathtt{s1}, \mathtt{s4} \rangle$, and $\langle \mathtt{s1}, \mathtt{s1}, \mathtt{s1}, \ldots \rangle$. ■

A reachability goal is similar to a goal in classical planning in that a plan succeeds if it reaches a state that satisfies the goal. However, because the execution of a plan may produce more than one possible path, the definition of a solution to a planning problem is more complicated than in classical planning. A *planning problem* is a triple (Σ, S_0, S_g), where $\Sigma = (S, A, \gamma)$ is a planning domain, $S_0 \subseteq S$ is a set of initial states, and $S_g \subseteq S$ is a set of goal states. Let π be a policy for Σ. Let $\Sigma_\pi = (Q, T)$ be the execution structure induced by π from S_0. Then we distinguish among three kinds of solutions.

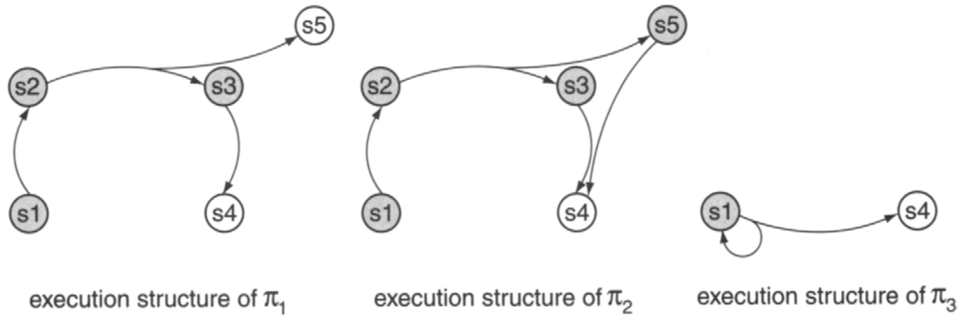
s1
s2
s3
s4
s5
execution structure of π_1
s1
s2
s3
s4
s5
execution structure of π_2
s1
s4
execution structure of π_3

Figure 17.3 The execution structures of the policies listed in Example 17.2.

1. *Weak solutions* are plans that may achieve the goal but are not guaranteed to do so. A plan is a weak solution if there is at least one finite path that reaches the goal. Formally π is a weak solution to P if and only if for each state in S_0 there exists a state in S_g that is a terminal state of Σ_π.
2. *Strong solutions* are plans that are guaranteed to achieve the goal in spite of nondeterminism: all the paths are finite and reach the goal. Formally, π is a strong solution to P if and only if Σ_π has no infinite paths, i.e., it is acyclic, and all of the terminal states of Σ_π are in S_g.
3. *Strong cyclic solutions* are guaranteed to reach the goal under a "fairness" assumption, i.e., the assumption that execution will eventually exit the loop. These solutions are such that all their partial execution paths can be extended to a finite execution path whose terminal state is a goal state. Formally, π is a strong cyclic solution to P if and only if from each state in Q there exists a terminal state of Σ_π that is reachable and all the terminal states of Σ_π are in S_g.

Weak and strong solutions correspond to the two extreme requirements for satisfying reachability goals. Intuitively, weak solutions correspond to "optimistic plans." Strong solutions correspond to "safe plans." However, there might be cases in which weak solutions are not acceptable and strong solutions do not exist. In such cases, strong cyclic solutions may be a viable alternative.

The set of strong solutions to a planning problem is a subset of the set of strong cyclic solutions, which in turn is a subset of the set of weak solutions.

Example 17.4 In Figure 17.1, let the initial state and goal state be s1 and s4, respectively. π_1 is a weak solution, π_2 is a strong solution, and π_3 is a strong cyclic solution. π_3 can be described as the iterative trial-and-error strategy "move the robot to location l4 until it succeeds," which repeats the execution of the action. This solution is much stronger than a weak solution: if at least one of the executions of the action succeeds, then we reach the goal. The only execution that won't reach the goal is the "unfair" one in which the move action fails forever. ■

17.2.2 Planning Algorithms

In this section, we describe some algorithms that generate strong, weak, and strong cyclic solutions.

Strong Planning. Figure 17.4 shows the algorithm Strong-Plan which, given a planning problem $P = (\Sigma, S_0, S_g)$ as input, returns either a policy that is a strong solution or failure if a strong solution does not exist. The algorithm is based on a breadth-first search proceeding backward from the goal states toward the initial states. It iteratively applies a subroutine StrongPreImg(S) that returns:

$$\text{StrongPreImg}(S) = \{(s, a) \mid \gamma(s, a) \neq \emptyset \text{ and } \gamma(s, a) \subseteq S\}$$

```
Strong-Plan($P$)
    $\pi \leftarrow$ failure; $\pi' \leftarrow \emptyset$
    While $\pi' \neq \pi$ and $S_0 \not\subseteq (S_g \cup S_{\pi'})$ do
        $PreImage \leftarrow$ StrongPreImg($S_g \cup S_{\pi'}$)
        $\pi'' \leftarrow$ PruneStates($PreImage$, $S_g \cup S_{\pi'}$)
        $\pi \leftarrow \pi'$
        $\pi' \leftarrow \pi' \cup \pi''$
    if $S_0 \subseteq (S_g \cup S_{\pi'})$ then return(MkDet($\pi'$))
        else return(failure)
end
```

Figure 17.4 Strong planning algorithm.

PreImage is therefore the set of pairs (s, a) such that a is guaranteed to lead to states in S_g or to states in $S_{\pi'}$ for which a solution is already known,[2] where $S_{\pi'}$ is the set of states of π'. This set is then pruned by removing pairs (s, a) such that a solution is already known for s:

$$\text{PruneStates}(\pi, S) = \{(s, a) \in \pi \mid s \notin S\}$$

The algorithm terminates if the initial states are included in the set of accumulated states (i.e., $S_g \cup S_{\pi'}$) or if a fixed point has been reached from which no more states can be added to the policy π'. If there is no strong solution, then at some iteration StrongPreImg returns an empty set, and we get $\pi = \pi'$. In the first case, the returned policy is a solution to the planning problem. Notice, however, that π' may have more than one action for a given state. MkDet(π') returns a policy $\pi \subseteq \pi'$ such that $S_\pi = S_{\pi'}$, and π satisfies the requirement that one state has only one corresponding action. In the second case, no solution exists: indeed, there is some initial state from which the problem is not solvable.

Strong-Plan corresponds to the computation of a least fixed point where state–action pairs are incrementally added to the current policy until either the set of initial states is included or a fixed point is reached.

Example 17.5 We apply Strong-Plan to the planning domain in Figure 17.1 with $S_0 = \{$s1$\}$ and $S_g = \{$s4$\}$.

At the first iteration, *PreImage* is {(s3,move(r1,l3,l4)), (s5,move(r1,l5,l4))} while PruneStates has no effects. Therefore π' is the same as *PreImage*.

At the second iteration, *PreImage* becomes {(s3, move(r1,l3,l4)), (s5, move(r1,l5, l4)), (s2, move(r1,l2,l3)), (s4, move(r1,l4,l3)), (s4, move(r1,l4,l5))}. PruneStates eliminates (s3, move(r1,l3,l4)), (s5, move(r1,l5,l4)), (s4, move(r1,l4,l3)), and

2. Compare this with Γ and Γ^{-1} as defined in Chapter 2.

(s4, move(r1,l4,l5)). π' is therefore {(s3, move(r1,l3,l4)), (s5, move(r1,l5,l4)), (s2, move(r1,l2,l3))}.

At the next iteration, we get $\pi' = $ {(s3, move(r1,l3,l4)), (s5, move(r1,l5,l4)), (s2, move(r1,l2,l3)), (s1, move(r1,l1,l2))}. At this point, the termination test $S_g \cup S_{\pi'}$ is satisfied, so Strong-Plan returns π_2. ■

Strong-Plan is guaranteed to terminate. It is sound, i.e., the returned policies are strong solutions, and it is complete, i.e., if it returns failure, then there exists no strong solution. Moreover, it returns policies that are optimal in the following sense. A policy results in a set of paths. Consider the longest path of the policy, and let us call it the *worst path*. Then the solution returned by Strong-Plan has a minimal worst path among all possible solutions. For the formal proofs, see Cimatti *et al.* [128].

Weak Planning. The algorithm Weak-Plan for weak planning is identical to the algorithm for strong planning, except that StrongPreImg is replaced by WeakPreImg:

$$\text{WeakPreImg}(S) = \{(s, a) : \gamma(s, a) \cap S \neq \emptyset\}$$

Example 17.6 In the planning problem described in Example 17.5, Weak-Plan stops at the first iteration and returns the policy $\pi = $ {(s1, move(r1,l1,l4)), (s3, move(r1,l3,l4)), (s5, move(r1,l5,l4))}. Some remarks are in order. First, π is a strong cyclic solution, but this is fine because strong cyclic solutions also are weak solutions. Second, the pairs (s3, move(r1, l3, l4)) and (s5, move(r1, l5, l4)) are uneeded, and if we removed them from π, π still would be a weak solution. However, the presence of such pairs causes no harm, and it is very easy to build a procedure that removes them. Third, Weak-Plan returns not all the possible weak solutions but one that has the shortest path to the goal. ■

We can easily compare strong planning and weak planning in nondeterministic domains with classical planning in deterministic domains. Let $\mathcal{P} = (\Sigma, s_0, S_g)$ be a classical planning problem. Then $\mathcal{P}$ can be viewed as a nondeterministic planning problem in which every action happens to be deterministic. Because every action is deterministic, every solution policy for $\mathcal{P}$ is both a weak solution *and* a strong solution.

Suppose we invoke the strong planning algorithm (or the weak planning one) on such a planning problem. Then both algorithms will behave identically. In both cases, the algorithm will do a breadth-first search going backward from the goal states, ending when the algorithm has visited every node of Σ for which there is a path to a goal.

Let π be the solution policy returned by the planning algorithm. One way to visualize π is as a set of trees, where each tree is rooted at a different goal state. If S is the set of all states in Σ from which the goal is reachable, then the node sets of these trees are a partition of S.

```
Strong-Cyclic-Plan($S_0$,$S_g$)
    $\pi \leftarrow \emptyset$; $\pi' \leftarrow UnivPol$
    while $\pi' \neq \pi$ do
        $\pi \leftarrow \pi'$
        $\pi' \leftarrow$ PruneUnconnected(PruneOutGoing($\pi'$,$S_g$),$S_g$)
    if $S_0 \subseteq (S_g \cup S_{\pi'})$
        then return(MkDet(RemoveNonProgress($\pi'$,$S_g$)))
        else return(failure)
end
```

Figure 17.5 Strong cyclic planning algorithm: main routine. It repeatedly eliminates state–action pairs by calling the subroutines PruneOutgoing and PruneUnconnected till a greatest fixed point is reached. It then calls the subroutine RemoveNonProgress and checks whether a solution exists.

If s is any state from which it is possible to reach a goal state, then π gives us a path $\langle s, \pi(s), \pi(\pi(s)), \ldots \rangle$ to a goal state. For the case where $s = s_0$, this path is an irredundant solution plan for the classical planning problem.

Strong Cyclic Planning. The algorithm Strong-Cyclic-Plan is presented in Figure 17.5. It starts with the universal policy $UnivPol = \{(s, a) \mid a \in A(s)\}$ that contains all state–action pairs. It iteratively eliminates state–action pairs from the universal policy. This elimination phase, where state–action pairs leading to states out of the states of *UnivPol* are discarded,[3] corresponds to the while loop of Strong-Cyclic-Plan. It is based on the repeated application of PruneOutgoing and PruneUnconnected. (Figure 17.6 shows the three subroutines for Strong-Cyclic-Plan.) The role of PruneOutgoing is to remove every state–action pair that leads out of $S_g \cup S_\pi$. Because of the application of PruneOutgoing, from certain states it may become impossible to reach the set of goal states. The role of PruneUnconnected is to identify and remove such states. Due to this removal, the need may arise to eliminate further outgoing transitions, and so on. The elimination loop terminates when convergence is reached. Strong-Cyclic-Plan then checks whether the computed policy π tells what to do in every initial state, i.e., $S_0 \subseteq S_g \cup S_{\pi'}$. If this is not the case, then a failure is returned. The following example explains the need for the RemoveNonProgress subroutine.

Example 17.7 Figure 17.7 shows a variation of the domain in Figure 17.1. Consider the planning problem of going from location l1 (initial state s1) to location l6 (goal state s6). The action move(r1,l4,l1) in state s4 is "safe": if executed, it leads to

3. Note that *UnivPol* contains all the state–action pairs where the action is applicable to the state, and therefore there may be states that are not in the states of *UnivPol*.

```
PruneOutgoing($\pi$,$S_g$) ;; removes outgoing state-action pairs
    $\pi' \leftarrow \pi$ − ComputeOutgoing($\pi, S_g \cup S_\pi$)
    return($\pi'$)
end

PruneUnconnected($\pi$,$S_g$) ;; removes unconnected state-action pairs
    $\pi' \leftarrow \emptyset$
    repeat
        $\pi'' \leftarrow \pi'$
        $\pi' \leftarrow \pi \cap$ WeakPreImg($S_g \cup S_{\pi'}$)
    until $\pi'' = \pi'$
    return($\pi'$) end

RemoveNonProgress($\pi$,$S_g$) ;; remove state-action pairs that
                              ;; do not lead toward the goal
    $\pi^* \leftarrow \emptyset$
    repeat
        $PreImage \leftarrow \pi \cap$ WeakPreImg($S_g \cup S_{\pi^*}$)
        $\pi^*_{old} \leftarrow \pi^*$
        $\pi^* \leftarrow \pi^* \cup$ PruneStates($PreImage, S_g \cup S_{\pi^*}$)
    until $\pi^*_{old} = \pi^*$
    return($\pi^*$)
end
```

Figure 17.6 Strong cyclic planning algorithm: subroutines. PruneOutgoing removes every states–action pair that leads out of the current set of states $S_g \cup S_\pi$. PruneUnconnected removes every edges that are unconnected, i.e., do not lead to the goal. RemoveNonProgress removes pairs that do not lead toward the goal.

state s1, where the goal is still reachable. However, this action does not contribute to reaching the goal. On the contrary, it leads back to the initial state, from which it would be necessary to move again to state s4. Moreover, if the action move(r1,l4,l1) is performed whenever the execution is in state s4, then the goal will never be reached. ■

After the elimination loop, we may have state–action pairs like (s4, move(r1, l4, l1)) that, while preserving the reachability of the goal, still do not make any progress toward it. RemoveNonProgress takes care of removing all those state–action pairs that cause this kind of problem. It is very similar to the weak planning algorithm: it iteratively extends the policy backward from the goal. In this case, however, the weak preimage computed at any iteration step is restricted to the

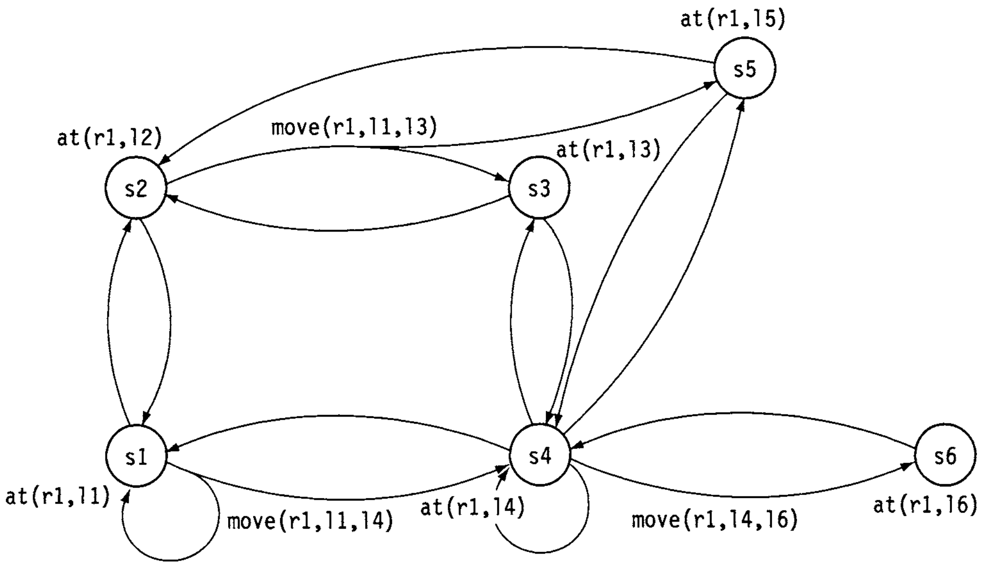

Figure 17.7 A nondeterministic state-transition system. It is a variation of the example in Figure 17.1. There is one additional state, state s6, and the two additional actions move(r1,l4,l6) and move(r1,l6,l4), which move the robot between locations l4 and l6.

state–action pairs that appear in the input policy and hence that are "safe" according to the elimination phase.

The subroutines PruneOutgoing, PruneUnconnected, and RemoveNonProgress, as shown in Figure 17.6, are based on the same routines WeakPreImg and PruneStates, which we already defined for the strong and weak planning algorithms (see Figure 17.4 and discussion), and on the primitive ComputeOutgoing, which takes as input a policy π and a set of states S and returns all state–action pairs that are not guaranteed to result in states in S:

$$\text{ComputeOutgoing}(\pi, S) = \{(s, a) \in \pi \mid \gamma(s, a) \not\subseteq S\}$$

Example 17.8 In Example 17.7, neither PruneOutgoing nor PruneUnconnected prune any pair from the universal policy, so the loop ends after the first iteration. RemoveNonProgress eliminates (s4, move(r1, l4, l1)), (s4, move(r1, l4, l3)), (s4, move(r1, l4, l5)), (s3, move(r1, l3, l2)), (s5, move(r1, l5, l2)), and (s2, move(r1, l2,l1)). MkDet returns either the policy that moves the robot from l1 to l6 through l4 or the policy that moves through l2 first and then to l3 or l5, and finally to l4, prior to reaching l6. ■

Strong-Cyclic-Plan is guaranteed to terminate, it is sound (i.e., the returned policies are strong cyclic solutions), and it is complete (i.e., if it returns failure, then there exists no strong cyclic solution). It corresponds to the computation of a greatest fixed point where states are removed when not needed (while strong and weak planning correspond to the computation of a least fixed point).

We conclude this section with two remarks. First, for the sake of simplicity, we have described weak, strong, and strong cyclic solutions in terms of properties of a graph representing the executions of a plan. In Section 17.3, we will show how these solutions can be expressed as formulas in temporal logic and how plan validation can be seen as a model checking problem. Second, notice that the algorithms we have presented are designed to work on sets of states and on sets of state–action pairs, thus taking advantage of the BDD-based symbolic model checking approach described in Appendix C . The basic building blocks of the algorithms, e.g., StrongPreImg and WeakPreImg, are minor variations of standard symbolic model checking routines (again see Appendix C).

17.3 Planning for Extended Goals

In this section we relax the basic assumption that goals are sets of final desired states. We extend the framework in two directions.

1. We allow for temporal goals, i.e., goals that state conditions on the whole execution path, rather than on its final state. This extension allows for expressing goals such as "generate a plan such that the robot keeps surveilling some rooms."
2. We allow for goals that take into account the uncertainty of the domain and express conditions of different strengths, such as the requirement that a plan should *guarantee* that the robot keeps visiting some rooms versus a plan that *does its best* to make the robot keep visiting the rooms.

We first formalize the planning problem through the use of temporal logic (Sections 17.3.1 and 17.3.2), and then we discuss a different approach to the formalization of extended goals (Section 17.3.3).

17.3.1 Domains, Plans, and Planning Problems

A planning domain is a state-transition system $\Sigma = (S, A, \gamma)$, where S, A, and γ are the same as in previous sections, except for one minor difference. In order to simplify the definitions, we require γ to be total, i.e., $\forall s\ A(s) \neq \emptyset$.

Policies (see Section 17.2.1) map states to actions to be executed. In the case of extended goals, plans as policies are not enough. Different actions may need to be executed in the same state depending on the previous states of the execution path.

Example 17.9 Consider the situation shown in Figure 17.1. Suppose the robot is in state s4 and the goal is "move to s3 first, and then to s5." There exists no policy that satisfies this goal because in s4 we have to execute two different actions, depending on whether we have already visited s3. ∎

A similar but more interesting example is the following: "starting from s4, keep moving back and forth between locations l3 and l5." Every time in s4, the robot has to take into account which state, s3 or s5, has just been visited and then move to the other one. In other words, plans have to take into account the *context of execution*, i.e., the internal state of the controller. A *plan* for a domain Σ is a tuple $(C, c_0, act, ctxt)$, where:

- C is a set of execution contexts.
- $c_0 \in C$ is the initial context.
- *act*: $S \times C \times A$ is the action function.
- *ctxt*: $S \times C \times S \times C$ is the context function.

If we are in state s and in execution context c, then $act(s, c)$ returns the action to be executed by the plan, while $ctxt(s, c, s')$ associates with each reached state s' the new execution context. Functions *act* and *ctxt* are partial because some state-context pairs are never reached in the execution of the plan.

In the following discussion, for the sake of readability, we write plans as tables whose columns describe the current state s, the current context c, the action to be performed in the current state and context (i.e., $act(s, c)$), the next state that may result from the execution of the action (i.e., $s' \in \gamma(s, act(s,c))$), and the next context $ctxt(s, c, s')$.

In Figure 17.1, suppose the goal is to move the robot from location l4 to l3 and then to l5. Table 17.1 shows a plan π_4 that satisfies the goal.

Now consider the goal "keep moving back and forth between locations l2 and l4, and never pass through l5." Notice that there is no plan that, at each round, can *guarantee* that the robot visits l2 and l4 without passing through l5. The requirement is too strong for the kind of uncertainty we have to deal with. However, in many cases, a weaker requirement is enough. For instance, we can require that at each round the robot *does* visit l2, *does not* pass through l5, and, *just if possible*, visits l4.

Table 17.1 An example of a plan for extended goals.

State	*Context*	*Action*	*Next state*	*Next context*
s4	c1	move(r1,l4,l3)	s3	c2
s3	c2	move(r1,l3,l4)	s4	c2
s4	c2	move(r1,l4,l5)	s5	c2

Table 17.2 A plan for goal "keep moving back and forth between two locations."

State	*Context*	*Action*	*Next state*	*Next context*
s1	c1	move(r1,l1,l2)	s2	c2
s1	c2	move(r1,l1,l4)	s1	c1
s1	c2	move(r1,l1,l4)	s4	c1
s2	c2	move(r1,l2,l1)	s1	c2
s4	c1	move(r1,l4,l1)	s1	c1

This goal expresses a different strength on the need to visit (or not visit) different locations. We can synthesize this goal with the following statement: "keep moving back and forth between locations l2 and, if possible, l4, and never pass through l5." A plan π_5 that satisfies this goal for the domain in Figure 17.1 is shown in Table 17.2.

We say that plan π is *executable* if, whenever $act(s, c) = a$ and $ctxt(s, c, s') = c'$, then $s' \in \gamma(s, a)$. We say that π is *complete* if, whenever $act(s, c) = a$ and $s' \in \gamma(s, a)$, then there is some context c' such that $ctxt(s, c, s') = c'$ and $act(s', c')$ is defined. Intuitively, a complete plan always specifies how to proceed for all the possible outcomes of any action in the plan. In the following discussion, we consider only plans that are *executable and complete.*

The execution of a plan results in a change in the current state and in the current context. It can therefore be described in terms of transitions from one state–context pair to another. Formally, given a domain Σ and a plan π, a transition of plan π in Σ is a tuple $(s, c) \xrightarrow{a} (s', c')$ such that $s' \in \gamma(s, a)$, $a = act(s, c)$, and $c' = ctxt(s, c, s')$. A *run* of plan π from state s_0 is an infinite sequence $(s_0, c_0) \xrightarrow{a_0} (s_1, c_1) \xrightarrow{a_1} (s_2, c_2) \xrightarrow{a_2} (s_3, c_3) \cdots$, where $(s_i, c_i) \xrightarrow{a_i} (s_{i+1}, c_{i+1})$ are transitions.

The *execution structure* of plan π in a domain Σ from state s_0 is the structure[4] $\Sigma_\pi = (Q, T, L)$, where:

- $Q = \{(s, c) \mid act(s, c) \text{ is defined}\}$
- $((s, c), (s', c')) \in T$ if $(s, c) \xrightarrow{a} (s', c')$ for some a

Example 17.10 Figure 17.8 (b) shows the execution structure of plan π_4, which moves the robot from location l4 to l3 and then to l5. ■

The goals we have considered so far can be expressed in computation tree logic (CTL) [171]. For instance, the goal "move the robot from location l4 to l3 and then to l5" is formalized as:

$$\text{at(r1, l4)} \rightarrow \text{AF}\,(\text{at(r1, l3)} \wedge \text{AF at(r1, l5)}) \tag{17.1}$$

4. Σ_π is a Kripke structure; see Appendix C.

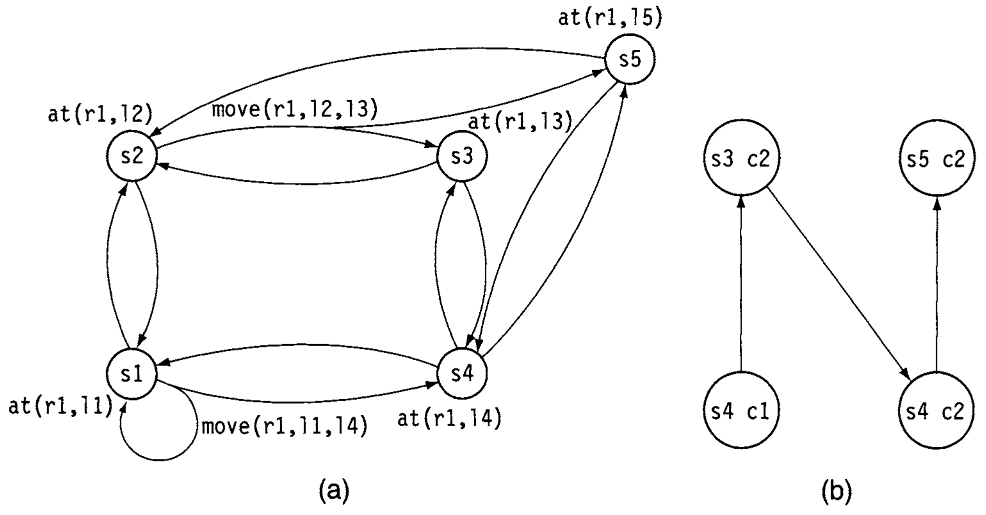

Figure 17.8 An example of a planning domain (a) and an execution structure (b).

AF is a CTL operator that is composed of two parts: a path quantifier A, which states that the formula should hold for all possible execution paths, a temporal operator F, which means "in the future" or "eventually." Therefore, AF at(r1, l5) means that for all possible execution paths, eventually there will be a state in the future where at(r1, l5) holds. The formula AF (at(r1, l3) $\wedge$ AF at(r1, l5)) means therefore that eventually in the future at(r1, l3) $\wedge$ AF at(r1, l5) will hold, i.e., there is a state in the future where at(r1, l3) holds, and from then on, at(r1, l5) holds. In general, AF $(p \wedge \text{AF } q)$ can be used to express the goal that p must become true and, after that p becomes true, then q becomes true.

Consider now the goal "keep moving back and forth between locations l2 and l4, and never pass through l5." It can be formalized in CTL with the formula:

$$\text{AG}\,(\text{AF}\,\texttt{at(r1,l2)} \wedge \text{AF}\,\texttt{at(r1,l4)}) \wedge \text{AG}\ \neg\texttt{at(r1,l5)} \qquad (17.2)$$

The CTL formula AG p states that p must hold "globally" or "always" in time, i.e., in all the future states. The formula AG (AF p $\wedge$ AF q) states that we should "keep to reach a state where p holds and keep moving to a state where q holds". As we know, the goal expressed in Formula 17.2 cannot be satisfied by any plan. The weaker goal "keep moving back and forth between locations l2 and, *if possible*, l4, and never pass through l5" is represented by:

$$\text{AG}\,(\text{AF}\,\texttt{at(r1,l2)} \wedge \text{EF}\,\texttt{at(r1,l4)}) \wedge \text{AG}\ \neg\texttt{at(r1,l5)} \qquad (17.3)$$

E is the CTL existential path quantifier, which means that there should exist a path such that a temporal property holds. EF p means that there exists a path such that in a future state p holds, while EG p means that there exists a path such that p always holds.

It is easy to express weak, strong, and strong cyclic goals in CTL. If g is the proposition representing the set of final states, weak solutions have to satisfy the goal EF g, and strong solutions, AF g. In order to express strong cylic solutions, we have to introduce another CTL operator: weak until. A(p W q) is satisfied if p holds forever or it holds until p holds. Strong cyclic solutions are represented with the formula A(EF g W g): either we are in a loop from which there is the possibility to get to g (EF g) or, if we get out of it, we get to g.

The strong until operator can be very useful to express temporal goals: A(p U q) is like A(p W q), but it does not allow for infinite paths where p always holds and q never becomes true; similarly for E(p U q) and E(p W q). Finally, the basic next step operator AX p states that p holds in all the successor states, while EX p state that p holds in at least one successor state.

Example 17.11 Consider the domain depicted in Figure 17.8 (a) (the same as the domain shown in Figure 17.1). The goal at(r1, l1) $\rightarrow$ EF at(r1, l4) corresponds to the requirement for a weak solution with initial state s1 and goal state s4. Similarly, at(r1, l1) $\rightarrow$ AF at(r1, l4) corresponds to the requirement of a strong solution. The requirement for the robot to try to reach location l4 from l1 by avoiding location l3 is expressed by the goal at(r1, l1) $\rightarrow$ (EF at(r1, l4) $\wedge$ AG $\neg$at(r1, l3)). The requirement for the robot "keep moving back and forth between locations l2 and, if possible, l4" can be expressed by the goal AG (AF at(r1, l2) $\wedge$ EF at(r1, l4)). ■

A *planning problem* is the tuple (Σ, S_0, g), where Σ is a nondeterministic state-transition system, $S_0 \subseteq S$ is a set of initial states, and g is a goal for Σ.

Let π be a plan for Σ and Σ_π be the corresponding execution structure. A plan π satisfies goal g from initial state $s_0 \in S$, written $\pi, s_0 \models g$, if $\Sigma_\pi, (s_0, c_0) \models g$. A plan π satisfies goal g from the set of initial states S_0 if $\pi, s_0 \models g$ for each $s_0 \in S_0$. The formal definition of $K, s_0 \models \phi$, where K is a Kripke structure and ϕ a CTL formula, can be found in Appendix C.

Σ_π is a Kripke structure. In model checking, Σ_π is the model of a system and g is a property to be verified. The plan validation problem, i.e., the problem of determining whether a plan satisfies a goal, is thus formulated as the model checking problem of determining whether the CTL formula g is true in the Kripke structure Σ_π, which represents the behaviors of the system Σ "controlled" by the plan π.

17.3.2 Planning Algorithms

Planning for CTL goals could in principle be done with a forward search and a progression of the CTL formula in a way similar to planning with LTL control rules (see Chapter 10). Thus, for instance, if the goal is A(p U q), we progress the goal to $q \vee$ AX ($p \wedge$ A(p U q)), we check whether q holds in the initial state, and if not,

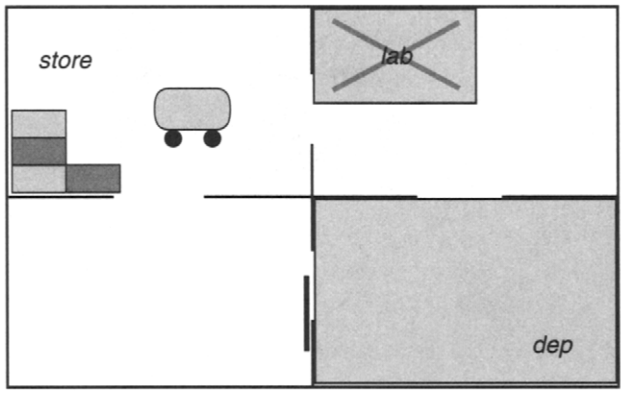

Figure 17.9 A domain example for planning for extended goals.

we select an action a applicable in the initial state. Then in all the states resulting from the application of a to the initial state, we check whether p holds and recursively call the procedure. The main difference from planning with LTL control rules is that particular attention must be paid to the universal and existential path quantifiers.

However, an explicit state forward search does not work in practice. Enumerating all the possible states is hopeless in the case of nondeterminism. An alternative approach is to use symbolic model checking techniques that work on sets of states rather than single states. In this section, we give some guidelines on how this can be done.

Consider a variation of Example 17.1 (see page 405) in which location $l2$ is a store, location $l5$ is a lab, and location $l4$ is a department ("dep" in Figure 17.9). Suppose the lab is a dangerous room, and if the robot gets into the lab, there is no way to come out. Suppose the goal is "keep moving back and forth between the store and, if possible, the department." Formally we have AG (AF store $\wedge$ EF dep).

The idea underlying the planning algorithm is to construct a control automaton that controls the search in the state space. The control automaton can be constructed by progressing the CTL goal. Each state of the automaton corresponds to a subgoal to be solved, which then becomes a context in the resulting plan. Arcs between states in the control automaton determine when the search switches from one context to another. For instance, the control automaton extracted by progressing the goal AG (AF store $\wedge$ EF dep) is shown in Figure 17.10. In the control automaton, we have two contexts: the one on the left, corresponding to the context where the next goal to satisfy is EF dep (and AF store afterward), and the one on the right, corresponding to the context where the next goal to satisfy is AF store (and EF dep afterward). The generated plan will have to keep satisfying both EF dep and AF store. When it is time to satisfy AF store (see the context on the right in the figure), the search has to find an action such that, if store holds, then we switch context (because the goal is

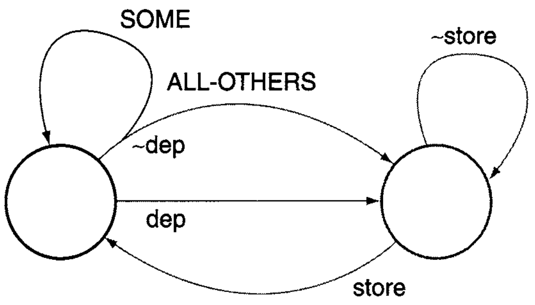

Figure 17.10 The control automaton for AG (AF store ∧ EF dep).

already satisfied) by searching for a plan that satisfies EF dep. If store does not hold, then we stay in the same context and search for a plan that satisfies AF store, i.e., that is guaranteed to reach the store. In order to satisfy EF dep (see the context on the left in the figure), the search has to find an action such that, if dep holds, then we switch context. If dep does not hold, then EF dep must be satisfied for *some* of the outcomes (the arc going back to the same context) and for *all the other* outcomes that do not satisfy EF dep. Indeed, if we generate a plan that satisfies EF dep, we have to take into account that the path that does not reach dep might invalidate the possibility to satisfy AF store at the next step. Consider, for instance, the case in which we generate a plan that moves from the store to the room north of the department, and the result is that the robot ends up in the lab. In this case, EF dep is satisfied, but this prevents achieving the goal of going back to the store because the robot is stuck in the lab forever. This is the reason why, in the control automaton, the arc that goes from the left context to the right context (in the case the robot is not in the department) is labeled ALL-OTHERS.

The control automaton is then used to guide the search for a plan. The idea is that the algorithm associates with each state of the control automaton (i.e., with each context that represents a subgoal) the plan that satisfies the subgoal. Initially, all plans are associated with each context. Then the association is iteratively refined: a context is chosen, and the search starts to get rid of the plans that do not satisfy the condition associated with the context. For instance, look at Figure 17.11. In step 1 we have chosen the context on the left, and we obtain the set of all the plans that satisfy EF dep. Notice that the plans specify when to change context. Notice also that, in this first step, the algorithm returns both the plan that goes through the door and the one that might end up in the lab. At this step, this is still a candidate plan because in the other context we have associated all the possible plans, and therefore for all the other plans that do not reach the department we can still reach the store.

The algorithm then considers the context on the right and generates the plans that satisfy AF store (step 2). The state in the lab is discarded because there is no plan that can reach the store from there. Control is back then to the context on the left (step 3). Because we have to guarantee that for all the cases where we do not

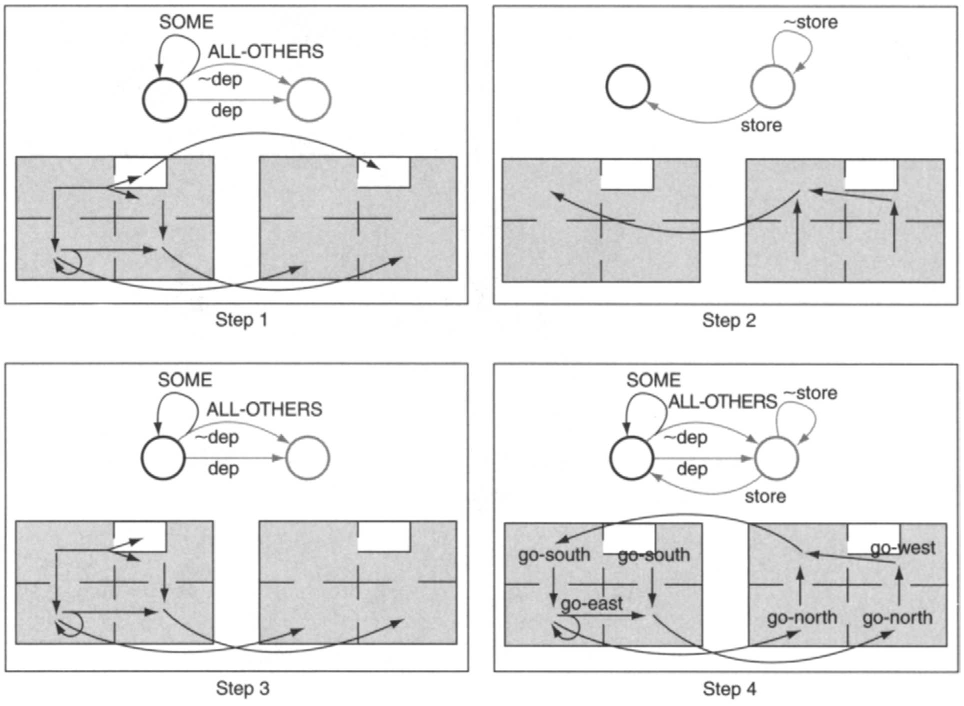

Figure 17.11 A simulation of the symbolic planning algorithm.

reach the department we thereafter reach the store, the plan that goes through the lab is discarded. We finally get to a fixed point and the final plan is returned (step 4).

Once the control automaton is constructed, the algorithm works on sets of states and sets of plans. It is then possible to implement it using BDDs effectively. It can be shown experimentally that, in the case of weak, strong, and strong cyclic reachability goals, beyond a small overhead for constructing the control automaton, the algorithm is comparable with the specialized algorithms for reachability defined in Section 17.2. It is easy to show that the algorithm outperforms by orders of magnitudes existing algorithms based on enumerative approaches.

17.3.3 Beyond Temporal Logic

In spite of the expressiveness of temporal logic, they cannot express some goals that seem to be important for, e.g., safety-critical applications. For instance, consider the model of a simplified controller of a railway switch in Figure 17.12. The switch can be in three main states: a reverse position (R) such that the train changes track, a direct position (D) such that the train keeps going on the same track,

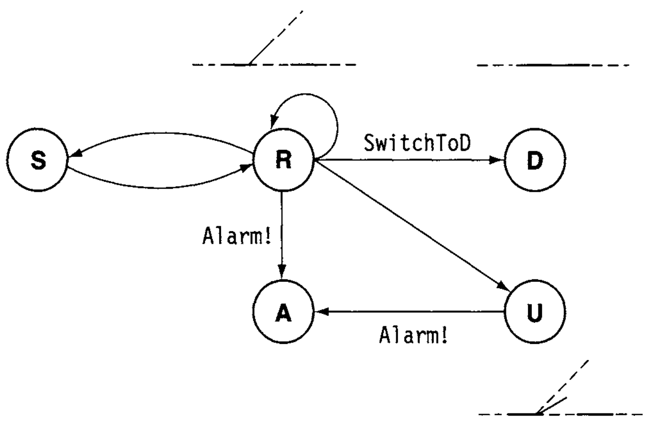

Figure 17.12 A railway switch example. The goal is *"try to reach the direct position; if you fail, send an alarm!"*

or an undefined position (U). The latter position is dangerous because a train might crash. There is a nondeterministic action that moves the switch from the reverse to the direct position (command SwitchToD). It may either succeed or fail. If it succeeds, then the switch gets to the direct position. There are two modes in which SwitchToD can fail. Either the switch does not move, or it ends up in the undefined position. We can send an alarm (command Alarm!) both from the reverse position and from the undefined position and get to the alarm state A. Consider the following goal:

Try to reach the direct position; if you fail, send an alarm!

There is no CTL formula (nor LTL formula) that can express this goal. There are indeed two problems here.

1. Consider the goal "try to reach the direct position." Its intended meaning is *do whatever is possible* to reach the direct position. Such a goal can be satisfied only by a plan that, in state R, always applies SwitchToD until it reaches either D or U. We cannot formalize this goal with the CTL formula EF D because then any plan that tries once, fails, and then keeps moving back and forth between R and S satisfies EF D without really trying its best to reach D. Notice also that a strong cyclic solution does not exist. Indeed, the *intentionality* of the goal is not captured in CTL.
2. In the goal, the intended meaning of "if you fail" is that you should first try to do whatever is possible to reach the direct position, and then, only in the case there is nothing else to do, you should send the alarm. In the example, this means we should send the alarm only in the case we get to U. Again, there is no way to express this in temporal logic. A formula that encodes the "if you fail" statement with a disjunction or with an implication cannot prevent the

planner from finding a plan that ignores the first part of the goal and satisfies the one that requires an alarm to be sent, e.g., by sending the alarm from R. In temporal logic, there is no way to express constructs for failure handling and, more generally for preferences. Indeed, the failure statement can be seen as a preference: the first preference is to try to move the switch to the right position; the second preference is to send an alarm.

We need a language with a different semantics that takes into account the points of failure. A first attempt toward this direction is the EaGLe language [136]. EaGLe has the following syntax.

- Reachability (basic) goals: DoReach p, TryReach p
- Maintenance (basic) goals: DoMaintain p, TryMaintain p
- Conjunction: g And g'
- Failure: g Fail g'
- Control operators: g Then g', Repeat g

Goal "DoReach p" requires a plan that guarantees to reach p despite nondeterminism. It fails if no such plan exists. Its semantics is similar to that of the CTL formula AFp, but DoReach keeps track of the points of failure and success. We explain this in the case of the goal "TryReach p." This goal requires a plan that does its best to reach p. It fails when there is no possibility to reach p. Figure 17.13 shows some examples that should help the reader understand its semantics. The trees in Figure 17.13 correspond to the unfolding of three different examples of domains. The semantics of TryReach p follows.

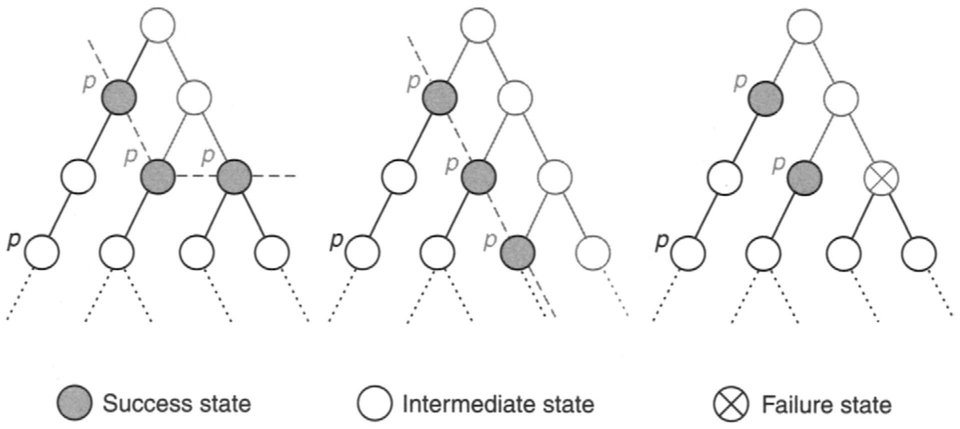

Figure 17.13 Semantics of TryReach p.

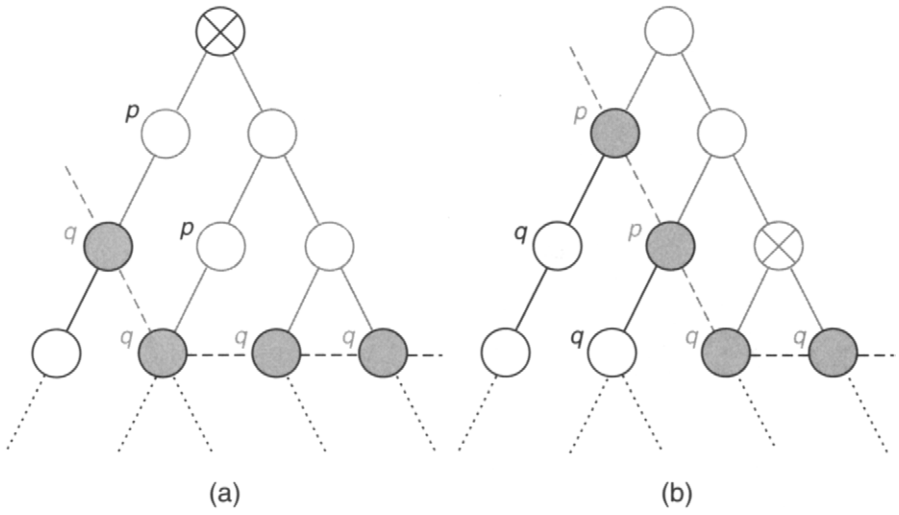

Figure 17.14 Semantics of g_1 Fail g_2: DoReach p Fail DoReach q (a); TryReach p Fail DoReach q (b).

- The states where p holds are *success states.*
- The states from which p can never be reached are *failure states.*
- The states where p does not hold but is reachable are marked as *intermediate states,* i.e., states that are neither success states nor failure states.

Goal "g_1 Fail g_2" deals with failure/recovery and with preferences among goals. The plan tries to satisfy goal g_1; whenever a failure occurs, goal g_2 is considered instead. Figure 17.14 (a), shows an example for goal DoReach p Fail DoReach q. Because DoReach p fails in the root state, then we plan for DoReach q. More interesting is the example in Figure 17.14 (b), i.e., the case of goal TryReach p Fail DoReach q. In the failure state of TryReach p, and just in that state, we plan for DoReach q.

The semantics of the other operators is given intuitively in the following way: goal "g_1 And g_2" requires satisfying g_1 and g_2 in parallel; goal "g_1 Then g_2" requires satisfying g_1 and then satisfying g_2; goal "Repeat g" requires satisfying g in a cyclic way.

Given this semantics, it is possible to generate a control automaton that guides the search for a plan and an algorithm that iteratively refines the set of plans corresponding to each state in the control automaton. This can be done in a way similar to the control automaton construction and the search algorithm for CTL goals. The search can thus be performed by means of symbolic BDD-based model checking techniques. See Dal Lago *et al.* [136] for a discussion of a possible algorithm.

17.4 Planning under Partial Observability

In this section we address the problem of planning under partial observability. We restrict the discussion to the case of reachability goals and in particular to strong solutions, i.e., solutions guaranteed to reach a given set of states (see Section 17.2), in spite of nondeterminism and in spite of partial observability.

17.4.1 Domains, Plans, and Planning Problems

We consider a nondeterministic state-transition system $\Sigma = (S, A, \gamma)$, where S is the set of states, A is the set of actions, and γ is the state-transition function (they are the same as in Section 17.2.1). Partial observability can be formalized with a set of *observations* that represent the part of the state-transition system that is visible and an *observation function* that defines what observations are associated with each state.

Definition 17.1 Let $\mathcal{S}$ be the set of states of a state-transition system, and let Ω be a finite set of observations. An *observation function* over $\mathcal{S}$ and Ω is a function $\mathcal{O} : \mathcal{S} \rightarrow 2^{\Omega}$, which associates with each state s the set of possible observations $\mathcal{O}(s) \subseteq \Omega$. We require that for each $s \in \mathcal{S}$, $\mathcal{O}(s) \neq \emptyset$. ■

The condition $\mathcal{O}(s) \neq \emptyset$ states the simple technical requirement that some observation is associated with each state. We allow for incomplete information in the case different states result in the same observation, i.e., $\mathcal{O}(s_1) = \mathcal{O}(s_2)$, with $s_1 \neq s_2$. Observation functions can model both full and null observability as special cases. *Null observability* is modeled with observation functions that map all the states to the same observation: $\Omega = \{o\}$ and $\mathcal{O}(s) = \{o\}$ for each $s \in \mathcal{S}$. In this case, observations carry no information because they are indistinguishable for all the states. *Full observability* is modeled with a one-to-one mapping between states and observations: $\Omega = \mathcal{S}$ and $\mathcal{O}(s) = \{s\}$. In this case, observations carry all the information contained in the state of the domain.

Definition 17.1 does not allow for a direct representation of *action-dependent* observations, i.e., observations that depend on the last executed action. However, these observations can be easily modeled by representing explicitly in the state of the domain the relevant information on the last executed action.

Example 17.12 Figure 17.15 shows a simple robot navigation domain. The robot can be in four positions corresponding to the states of the domain: $\mathcal{S} = \{\mathsf{NW}, \mathsf{NE}, \mathsf{SW}, \mathsf{SE}\}$. The robot can move in the four directions: $\mathcal{A} = \{\mathsf{GoNorth}, \mathsf{GoSouth}, \mathsf{GoEast}, \mathsf{GoWest}\}$. The actions are applicable if there is not a wall in the direction of motion. If the actions are deterministic, we have, e.g., $\gamma(\mathsf{NW}, \mathsf{GoEast}) = \{\mathsf{NE}\}$, while $\gamma(\mathsf{NW}, \mathsf{GoSouth}) = \emptyset$ because of the wall blocking movement there. We can

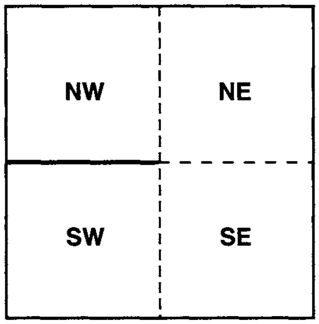

Figure 17.15 A simple partially observable domain.

define a set of propositions $\mathcal{P}$ = {north-west, north-east, south-west, south-east}, and $\mathcal{S}$ is defined to be the set of states where only one of the propositions in $\mathcal{P}$ holds.

If the robot can perform only local observations, it cannot distinguish between the two positions NW and SW. We model this partial observability with a set of observations Ω = {west, ne, se}, and the observation function $\mathcal{O}$ such that $\mathcal{O}$(NW) = $\mathcal{O}$(SW) = west, $\mathcal{O}$(NE) = ne, and $\mathcal{O}$(SE) = se. ■

An alternative way to represent partial observability is to define a set of *observation variables*, whose values can be observed at run-time, during the execution of a plan. We define the *evaluation of an observation variable* to be a relation that specifies what values can be assumed at run-time by the observation variables.

Definition 17.2 Let $\mathcal{S}$ be the set of states of a state-transition system, and let $\mathcal{V}$ be a finite set of observation variables. The *evaluation of an observation variable* $v \in \mathcal{V}$ is the relation $\mathcal{X}_v : \mathcal{S} \times \{\top, \bot\}$. ■

Without loss of generality, we assume that observation variables are Boolean. The symbols $\top$ and $\bot$ stand for true and false, respectively, and represent the evaluation of the Boolean variable v to true or false. In a state $s \in \mathcal{S}$, an observation variable $v \in \mathcal{V}$ may give no information: this is specified by stating that both $\mathcal{X}_v(s, \top)$ and $\mathcal{X}_v(s, \bot)$ hold, i.e., both the true and false values are possible. In this case, we say that the observation variable v is *undefined* in s. If $\mathcal{X}_v(s, \top)$ holds and $\mathcal{X}_v(s, \bot)$ does not hold, then the value of v in state s is true. The dual holds for the false value. In both cases, we say that v is *defined in* s. An observation variable is always associated with a value, i.e., for each $s \in \mathcal{S}$, at least one of $\mathcal{X}_v(s, \top)$ and $\mathcal{X}_v(s, \bot)$ holds. As the next example shows, the evaluation of observation variables can be easily related to observations by defining the set of observations as all the evaluations of observation variables.

Example 17.13 Consider the situation shown in Figure 17.15. Suppose that the sensors of the robot can detect walls in the current position. This can be formalized with the set of observation variables $\mathcal{V} = \{\mathsf{WallN}, \mathsf{WallS}, \mathsf{WallW}, \mathsf{WallE}\}$. The evaluation is such that $\mathcal{X}_{\mathsf{WallE}}(\mathsf{NW}, \bot)$ and $\mathcal{X}_{\mathsf{WallW}}(\mathsf{NW}, \top)$. In this case, every observation variable is defined in every state of the domain.

In a different formulation of the domain, a further observation variable could detect whether a wall is at distance 2 from the current position, e.g., 2WallS detects walls at south at distance 2. In this case, 2WallS is undefined in state NW, i.e., $\mathcal{X}_{\mathsf{2WallS}}(\mathsf{NW}, \bot)$ and $\mathcal{X}_{\mathsf{2WallS}}(\mathsf{NW}, \top)$.

There exists a relation between observations and observation variables. For instance, the observation west is equivalent to the following result of the evaluation: $\mathcal{X}_{\mathsf{WallN}}(\mathsf{NW}, \top)$, $\mathcal{X}_{\mathsf{WallW}}(\mathsf{NW}, \top)$, $\mathcal{X}_{\mathsf{WallS}}(\mathsf{NW}, \top)$, $\mathcal{X}_{\mathsf{WallE}}(\mathsf{NW}, \bot)$. ■

A partially observable planning domain is a state-transition system with partial observability. We represent partial observability with observation variables and their evaluations.

Definition 17.3 A *planning domain* $\mathcal{D}$ is a tuple $(\Sigma, \mathcal{V}, \mathcal{X})$, where:

- $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$ is a state-transition system.
- $\mathcal{V}$ is a finite set of observation variables.
- $\mathcal{X}_v : \mathcal{S} \times \{\top, \bot\}$ is an evaluation of the observation variables. ■

In partially observable domains, plans need to branch on conditions on the value of observation variables.

Definition 17.4 The set of *conditional plans* Π for a domain $\mathcal{D} = (\Sigma, \mathcal{V}, \mathcal{X})$ is the minimal set such that:

- $\lambda \in \Pi$.
- If $a \in \mathcal{A}$, then $a \in \Pi$.
- If $\pi_1, \pi_2 \in \Pi$, then $\pi_1; \pi_2 \in \Pi$.
- If $v \in \mathcal{V}$, and $\pi_1, \pi_2 \in \Pi$, then **if** v **then** π_1 **else** $\pi_2 \in \Pi$. ■

Intuitively, λ is the empty plan, i.e., the plan that does nothing. The plan $\pi_1; \pi_2$ is the sequential composition of the plans π_1 and π_2. The plan **if** v **then** π_1 **else** π_2 is a conditional plan that branches on the value of the observation variable v.

Example 17.14 For the situation shown in Figure 17.15, a plan that moves the robot from the uncertain initial condition NW or SW to state SW is

GoEast ; (**if** WallN **then** GoSouth **else** λ) ; GoWest

Intuitively, action GoEast is executed first. Notice that in the initial condition, all the observation variables would have the same value for both NW and SW: therefore the states are indistinguishable, and it is pointless to observe. Action GoEast moves the robot either to state NE or to SE, depending on the initial position of the robot. If observation variable WallN is true, then the robot is in NE, and action GoSouth moves the robot to SE. If WallN is false, the conditional plan does nothing, i.e., λ is executed. Finally, GoWest moves the robot from state SE to SW. ■

Let us formalize the execution of a plan. Given an observation variable $v \in \mathcal{V}$, we denote with $v_\top$ the set of states where v evaluates to true: $v_\top \doteq \{s \in \mathcal{S} : \mathcal{X}_v(s, \top)\}$. Similarly, $v_\bot \doteq \{s \in \mathcal{S} : \mathcal{X}_v(s, \bot)\}$ is the set of states where v is false. If v is undefined in a state s, then $s \in v_\top \cap v_\bot$. Under partial observability, plans have to work on sets of states whose elements cannot be distinguished, i.e., on belief states. We say that an *action a is applicable to a nonempty belief state Bs* iff a is applicable in all states of Bs. Plan execution is represented by the function $\Gamma(\pi, Bs)$, which, given a plan π and a belief state Bs, returns the belief state after execution.

Definition 17.5 Let Bs be a nonempty belief state, i.e., $\emptyset \neq Bs \subseteq \mathcal{S}$.

1. $\Gamma(\pi, \emptyset) \doteq \emptyset$.
2. $\Gamma(a, Bs) \doteq \{s' : s' \in \gamma(s, a),$ with $s \in Bs\}$, if $a \in \mathcal{A}$ is applicable in Bs.
3. $\Gamma(a, Bs) \doteq \emptyset$, if a is not applicable in Bs.
4. $\Gamma(\lambda, Bs) \doteq Bs$.
5. $\Gamma(\pi_1; \pi_2, Bs) \doteq \Gamma(\pi_2, \Gamma(\pi_1, Bs))$.
6. $\Gamma(\textbf{if } v \textbf{ then } \pi_1 \textbf{ else } \pi_2, Bs) \doteq \Gamma(\pi_1, Bs \cap v_\top) \cup \Gamma(\pi_2, Bs \cap v_\bot)$, if condition $app(v, \pi_1, \pi_2, Bs)$ holds, where:
$$\begin{aligned} app(v, \pi_1, \pi_2, Bs) = \quad & (Bs \cap v_\top \neq \emptyset \rightarrow \Gamma(\pi_1, Bs \cap v_\top) \neq \emptyset) \wedge \\ & (Bs \cap v_\bot \neq \emptyset \rightarrow \Gamma(\pi_2, Bs \cap v_\bot) \neq \emptyset). \end{aligned}$$
7. $\Gamma(\textbf{if } v \textbf{ then } \pi_1 \textbf{ else } \pi_2, Bs) \doteq \emptyset$, otherwise.

■

We say that a plan π is applicable in $Bs \neq \emptyset$ if and only if $\Gamma(\pi, Bs) \neq \emptyset$. For conditional plans, we collapse into a single set the execution of the two branches (item 6). Condition $app(v, \pi_1, \pi_2, Bs)$ guarantees that both branches are executable: the then-branch π_1 must be applicable in all states of Bs where v is true, and the else-branch π_2 must be applicable in all states of Bs where v is false. This condition correctly allows for states where v is undefined, i.e., where v is both true and false.

Example 17.15 Figure 17.16 depicts the execution of the plan in Example 17.14. $\Gamma(\text{GoEast}, \{\text{NW}, \text{SW}\}) = \{\text{NE}, \text{SE}\}$, i.e., after the robot moves east, it is guaranteed to be either in NE or SE. The plan then branches on the value of WallN. This allows the planner to distinguish between state NE and SE: if the robot is in NE, then it moves south; otherwise, it does nothing. At this point the robot is guaranteed to be in SE and can finally move west. This plan is guaranteed to reach SW from any of the initial states, either with three actions (if the initial state is NW), or with two actions (if the initial state is SW). ■

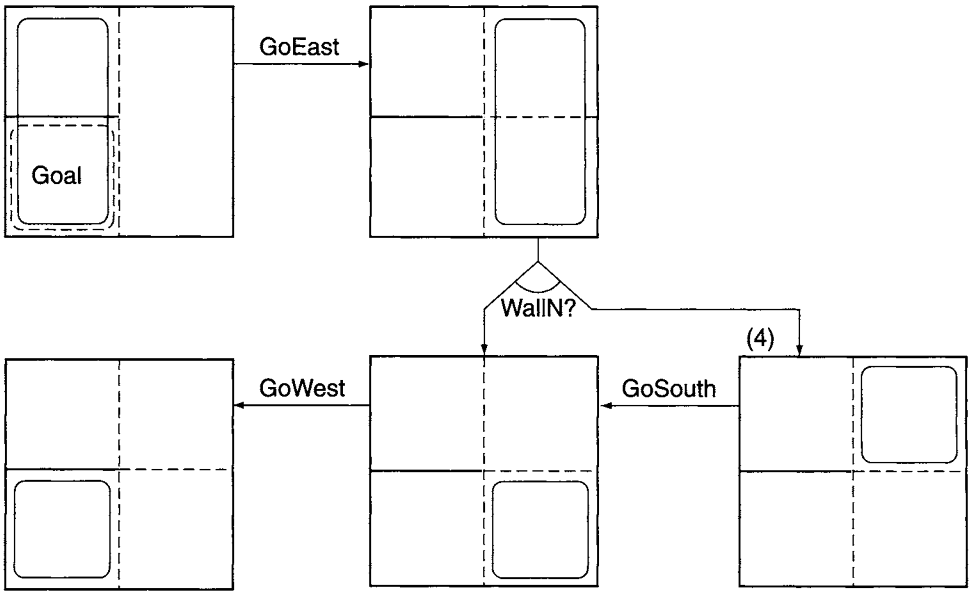

Figure 17.16 Execution of a conditional plan.

We formalize the notion of a planning problem under partial observability as follows.

Definition 17.6 A *planning problem* is a 3-tuple $(\mathcal{D}, S_0, S_g)$, where $\mathcal{D} = (\Sigma, \mathcal{V}, \mathcal{X})$ is a planning domain, $\emptyset \neq S_0 \subseteq \mathcal{S}$ is the initial belief state, and $\emptyset \neq S_g \subseteq \mathcal{S}$ is the goal belief state. The plan π is a strong solution to the problem $(\mathcal{D}, S_0, S_g)$ if and only if $\emptyset \neq \Gamma(\pi, S_0)) \subseteq S_g$. ∎

Example 17.16 Consider the problem given by the domain shown in Figure 17.15, the initial belief state $S_0 = \{\text{NW}, \text{SW}\}$, and the goal belief state $S_g = \{\text{SW}\}$. The plan in Example 17.14 is a strong solution to the planning problem. ∎

17.4.2 Planning Algorithms

We do not describe any detailed algorithm in this section; rather, we give some hints on how different planning algorithms can be defined. The common underlying idea of the planning algorithms is that the search space can be seen as an AND/OR graph over belief states. The AND/OR graph can be recursively constructed from the initial belief state, expanding each encountered belief state by every possible combination of applicable actions and observations.

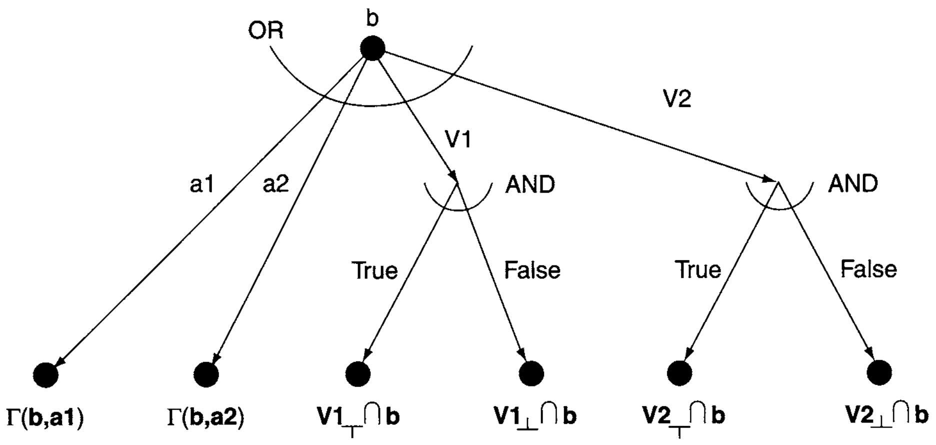

Figure 17.17 The search space as an AND/OR graph.

Figure 17.17 shows an example of an AND/OR graph on belief states. Each node in the graph is a belief state. A node b can be connected to OR-nodes or to AND-nodes. The OR-nodes connected to a node b are

- the belief states produced by the actions applicable to the belief state b (for each applicable action a, we have an OR-node $\Gamma(b, a)$ connected to node b), or
- the pairs of belief states produced by observations $v \in \mathcal{V}$ (each pair corresponds to two branches, one in which v is true and the other in which v is false).

The pair of belief states corresponding to the two branches over the observation variable v are AND-nodes in the AND/OR graph. For each $v \in \mathcal{V}$, a belief state b is connected to two AND-nodes: the subset of b in which the value of v is true (i.e., $v_\top \cap b$) and the subset of b in which the value of v is false (i.e., $v_\bot \cap b$).

A tentative solution plan can be incrementally constructed during the search. If π is the plan associated with node b, then the OR-node connected to b by an action a applicable in b will have an associated plan $\pi; a$. The OR-node connected to b by an observation variable $v \in \mathcal{V}$ will have an associated plan $\pi;$ **if** v **then** π_1 **else** π_2, where π_1 is the plan associated with the AND-node $v_\top \cap b$ and π_2 is the plan associated with the AND-node $v_\bot \cap b$.

The search terminates when it has constructed an acyclic subgraph of the AND/OR graph whose root is the initial belief state b_0 and all the leaf nodes are belief states that are subsets of the goal belief state b_g. Figure 17.18 shows a subgraph that corresponds to the plan a1; **if** v1 **then** π_1 **else** π_2, where $\pi_1 =$ a3; **if** v2 **then** π_3 **else** π_4, and $\pi_2 =$ a2; **if** v2 **then** π_5 **else** π_6, and so on.

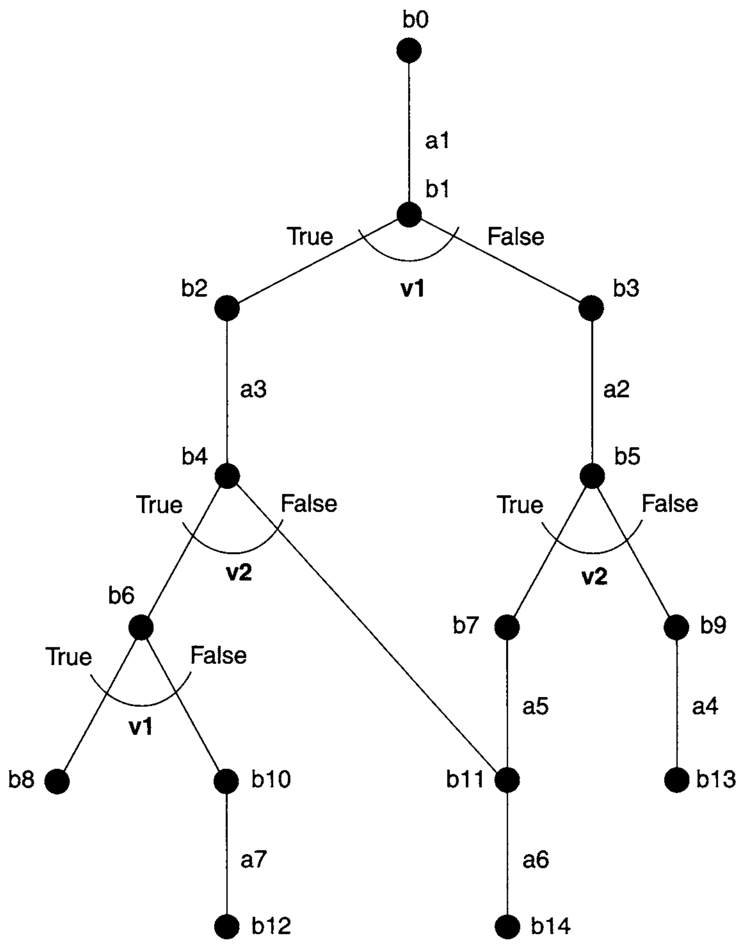

Figure 17.18 An AND/OR graph corresponding to a conditional plan.

A planning algorithm can search this AND/OR graph in different ways and with different strategies. For instance, a breadth-first forward search given a belief state *b* proceeds by exploring all the possible OR-nodes. The termination condition is reached when we have at least one AND/OR subgraph that meets the conditions described earlier. Breadth-first search can generate optimal plans, but this approach can be very expensive. A depth-first search selects one of the possible OR-nodes and needs to explore each branch of the selected AND-nodes. It can be much more convenient than a breadth-first search, but it does not guarantee to return optimal plans. Different heuristics can be applied in the depth-first search, e.g., select all the observations first, select the OR-nodes that produce smaller belief states, etc.

17.5 Planning as Model Checking versus MDPs

The MDP approach and the model checking approach take very different views of the planning problem. With MDPs, we have probabilities, a utility function, and an optimization problem. With planning as model checking, we have nondeterminism but no probabilities, a logical specification of the goal, and a satisfiability problem.

Probabilities are a very simple and natural way to express preferences, while the work on preferences in the model checking approach is still beginning [136]. Utility functions are a simple and effective way to express competing criteria. While costs and probabilities can provide useful information in some domains, there are domains where modeling transitions with costs and/or probabilities is rather difficult and unnatural in practice, e.g., because not enough statistical data are available. In MDP-based planning, planning is performed through the construction of optimal policies with respect to rewards and probability distributions. As a consequence, it is not possible to force a planner based on probabilistic models to generate a solution that satisfies some logical requirements. For instance, if both a strong and a strong cyclic solution do exist, an MDP-based planner cannot be forced to find a strong solution that is guaranteed to avoid possible loops at execution time. However, we have also discussed how the temporal logic currently used in the model checking approach is not fully satisfactory for expressing extended goals (see Section 17.3.3).

The basic version of MDP algorithms can rarely scale up to relevant applications. However, a significant amount of work on large state MDPs has tried to address this problem (see, e.g., the work on factorized MDPs, abstraction, and symbolic representations mentioned in Chapter 16). The model checking approach to planning can exploit the use of BDD-based symbolic techniques that have been shown to outperform in many cases explicit state enumerative techniques. Planners based on symbolic model checking, like the Model Based Planner (MBP) [65][5] have been shown to outperform planners based on MDPs (such as GPT [83]) in several experimental benchmarks [66, 128]. However, these results should be interpreted carefully because the two approaches and the two kinds of planners solve different problems.

17.6 Discussion and Historical Remarks

The idea of planning by using explicit state model checking techniques has been around since the work by Kabanza *et al.* [293] and the relative planner SimPlan. In SimPlan, temporal goals can be expressed in an extension of Linear Temporal Logic (LTL), and the associated plans are constructed under the hypothesis of full observability. SimPlan relies on the *explicit-state* model checking paradigm, where

5. MBP is available at http://sra.itc.it/tools/mbp.

individual states are manipulated (rather than sets of states as in the planning algorithms presented in this chapter). The enumerative nature of this approach can be a major drawback in large domains. For this reason, in SimPlan, LTL formulas are also used to describe user-defined control strategies, which can provide aggressive pruning of the search space. This is done in the same style of control strategies in TLPlan [33]. LTL allows for the specification of temporally extended goals. Because LTL and CTL have incomparable expressiveness, the classes of problems solved by SimPlan are incomparable with those discussed in this chapter. For instance, LTL can express strong goals but not strong cyclic goals, which require properties of branching computation structures. However, LTL can express fairness conditions, such as "always eventually p," that cannot be expressed in CTL.

The idea of using model checking techniques became popular with the proposal of the "planning via symbolic model checking" paradigm (the one described in this chapter), which has shown how the approach can deal in practice with planning under uncertainty. The idea of planning as model checking was first introduced in two works [127, 234]. Strong planning was first proposed by Cimatti *et al.* [130]. Strong cyclic planning was also first proposed by Cimatti *et al.* [129] and then revised by Daniele *et al.* [137]. A full formal account and an extensive experimental evaluation of weak, strong, and strong cyclic planning were presented by Cimatti *et al.* [128]. The framework has been extended to deal with partial observability [66] and with extended goals [136, 440, 441]. All the results described in the works cited here have been implemented in the MBP [65].

Along this line, there have been different proposals. The work by Jensen and Veloso [281] exploits the idea of planning via symbolic model checking as a starting point for the work on the UMOP.[6] Jensen *et al.* extended the framework to deal with contingent events in their proposal for adversarial weak, strong, and strong cyclic planning [283]. They also provided a novel algorithm for strong and strong cyclic planning that performs heuristic-based, guided, BDD-based search for nondeterministic domains [282].

BDDs have also been exploited in classical planners (see Traverso *et al.* [511] for a report about BDD-based planners for classical planning in deterministic domains). Among them, MIPS [169] showed remarkable results in the AIPS '00 planning competition for deterministic planning domains. BDD-based planners like MIPS are specialized to deal with deterministic domains and are therefore more efficient than planners like MBP on classical planning domains. One of the reasons for this is the use of advanced mechanisms to encode PDDL planning problems into BDDs (see, e.g., [168]). An interesting open research issue is whether these encoding techniques can be lifted to the case of nondeterministic domains.

Other approaches are related to model checking techniques. Bacchus and Kabanza [33] use explicit-state model checking to embed control strategies expressed in LTL in TLPlan (see Chapter 10). The work by Goldman *et al.* [238, 240, 241] presents a method where model checking with timed automata is used to verify that generated plans meet timing constraints.

6. UMOP is available at http://www-2.cs.cmu.edu/ runej/publications/umop.html.

Finally, the SPUDD planner [269] makes use of Algebraic Decision Diagrams, data structures similar to BDDs, to do MDP-based planning. In SPUDD, decision diagrams are used to represent much more detailed information (e.g., the probabilities associated with transitions) than in planning as model checking. This partly reduces the main practical advantages of decision diagrams as they are used in planning as model checking.

17.7 Exercises

17.1 Consider Example 17.1 (see page 405) with the goal to move the robot to state s4. Let the set of initial states be s1 and s2. Are π_1, π_2, and π_3 weak, strong, or strong cyclic solutions?

17.2 In Example 17.1, with the goal to move the robot to state s4, modify the action move(r1,l1,l4) such that its outcome can also be in state s3. Write a weak solution, a strong solution, and a strong cyclic solution (if they exist).

17.3 In Example 17.1, with the goal to move the robot to state s4, modify the action move(r1,l1,l4) such that its outcome can also be in a dead end state s6. Write a weak solution, a strong solution and a strong cyclic solution (if they exist).

17.4 Suppose we translate the MDP planning problem of Exercise 16.8 into a nondeterministic planning problem in which start-fill and start-wash may either succeed or fail, and end-fill and end-wash are guaranteed to succeed.

(a) Draw the state-transition system. How many weak policies are there? How many strong policies? How many strong cyclic policies?

(b) Suppose that Dan wants both to wash the dishes and wash his clothes (and that both start-fill(x) and start-wash(x) may either succeed or fail, regardless of whether x = dw or x = wm). Draw the state-transition system for this problem. (Hint: The diagram will be quite large. To make it easier to draw, do not give names to the states, and use abbreviated names for the actions.)

(c) In the state-transition system of part (b), are there any states that Strong-Plan will not visit? How about Weak-Plan?

17.5 Consider Example 17.11 (see page 418). Consider the plan π shown in Table 17.3. Let the initial state be s1 and the goal state s4. Is π a weak, strong, or strong cyclic solution? What about the plan shown in Table 17.4?

17.6 In the DWR domain, consider the possibility that containers are downloaded dynamically from boats at different locations. Write the CTL goal that makes the robot move all the containers as they appear to a given location.

17.7 In the DWR domain, the robot cannot know whether there is a container in a given location unless the robot is in the location. Consider the goal to have all containers in a given location. Write a conditional plan that is a strong solution.

Table 17.3 First Plan for Exercise 17.5.

State	*Context*	*Action*	*Next state*	*Next context*
s1	c0	move(r1,l1,l4)	s1	c1
s1	c0	move(r1,l1,l4)	s4	c4
s4	c0	wait	s4	c0
s1	c1	move(r1,l1,l2)	s2	c2
s2	c2	move(r1,l2,l1)	s1	c2
s1	c2	move(r1,l1,l2)	s2	c2

Table 17.4 Second Plan for Exercise 17.5.

State	*Context*	*Action*	*Next state*	*Next context*
s1	c0	move(r1,l1,l4)	s1	c1
s1	c0	move(r1,l1,l4)	s4	c4
s4	c0	wait	s4	c0
s1	c1	move(r1,l1,l2)	s2	c2
s2	c2	move(r1,l2,l1)	s1	c0

17.8 Run MBP on the scenarios described in Examples 17.1, 17.5, 17.7, and 17.11 (see pages 405, 409, 411, and 418, respectively).

17.9 Using the scenario from Example 17.1, find a policy that satisfies the EaGLe goal Try Reach at(r1,l4) Fail Try Reach at(r1,l5).