Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

© 2008 Elsevier B.V. All rights reserved
DOI: 10.1016/S1574-6526(07)03022-2

Chapter 22

Automated Planning

Alessandro Cimatti, Marco Pistore, Paolo Traverso

22.1 Introduction

We intuitively refer to the term *Planning* as the deliberation process that chooses and organizes actions by anticipating their expected effects [24]. This deliberation aims at satisfying some pre-defined requirements and achieving some prestated objectives.

The intuition is that actions are executed in a given domain. They make the domain evolve and change its state. For instance, in a robot navigation domain, an action moving the robot changes its position; in the case of a microprocessor, an instruction can be viewed as an action that changes the value of the registers; a web service for booking flights can receive a message with a flight reservation confirmation, and this is an action that changes its state.

The deliberation process can organize actions in different ways. For instance, moving a robot to a given room and then to the corridor is an example of a sequential organization of actions; executing an instruction depending on the result of the execution of a previous one is an example of a conditional combination of actions; requesting for a flight reservation until a seat is available is an example of an iterative combination.

Actions are organized and combined with the aim to satisfy some requirements on the evolution of the domain. An example of a requirement for a mobile robot is that of "reaching a given room", while a requirement for a flight service can be that of "never exceeding a given number of overbooking".

Automated Planning is the area of Artificial Intelligence that studies this deliberation process computationally. Its aim is to support the planning activity by reasoning on conceptual models, i.e., abstract and formal representations of the domain, of the effects and the combinations of actions, and of the requirements to be satisfied and the objectives to be achieved. The conceptual model of the domain in which actions are executed is called the *planning domain*, combinations of actions are called *plans*, and the requirements to be satisfied are called *goals*. Intuitively, given a planning domain

and a goal, a *planning problem* consists in determining a plan that satisfies the goal in a given domain.

In this Chapter, we provide a general formal framework for Automated Planning. The framework is defined along the three main components of the planning problem: domains, plans, and goals.

– **Domains.** We allow for *nondeterministic domains*, i.e., domains in which actions may have different effects, and it is impossible to know at planning time which of the different possible outcomes will actually take place. We also allow for *partial observability*. It models the fact that in some situations the state of the domain cannot be completely observed, and thus cannot be uniquely determined. A model with partial observability includes the special cases of *full observability*, where the state can be completely observed and thus uniquely determined, and that of *null observability*, where no observation is ever possible at run time.

– **Plans.** We define plans where the action to be executed in a given state can depend on available information about the history of previous execution steps. The definition is general enough to include *sequential plans*, i.e., plans that are simply sequences of actions, *conditional plans*, i.e., plans that can choose a different action depending on the current situation at execution time, *iterative plans* that can execute actions until a situation occurs. We can have plans that depend on a finite number of execution steps (*finite-memory plans*), as well as plans that do not depend on the previous execution steps (*memory-less plans*). In general, plan executions result in trees (called execution trees) whose nodes correspond to states of the domain.

– **Goals.** We define goals as sets of acceptable trees that corresponds to desired evolutions of a planning domain. They can represent classical *reachability goals* that express conditions on the leaves of execution trees, which determine the final states to be reached after a plan is executed. More in general, they can represent more complex forms of "extended goals", like *temporally extended goals*, that express conditions on the whole execution tree.

Our framework is general enough to represent a relevant and significant set of planning problems. *Classical planning* (see, e.g., [22, 40]) can be modeled with deterministic domains, plans that are sequences of actions, and reachability goals. In addition, our framework is well suited for modeling certain forms of planning under uncertainty and incomplete information, which are being recently addressed in the research literature and are relevant to several real-world applications. Indeed, nondeterministic domains model uncertainty in action effects, while partial observability models uncertainty in observations. For instance, the so-called *conformant planning* (see, e.g., [14, 9]) can be modeled with nondeterministic domains, null observability, sequential plans, and reachability goals. *Contingent planning* (see, e.g., [13, 32, 5]) can be modeled with nondeterministic domains, conditional plans, and reachability goals. *Planning for temporally extended goals* (see, e.g., [44, 1, 35]) can be modeled with nondeterministic domains, history dependent plans, and goals that represent desired evolutions of the domain.

For practical reasons, the framework cannot be so general to include all the different planning problems that have been addressed in the literature so far. For instance, a difference with respect to planning based on Markov Decision Processes (MDP) [8] is that we do not represent probabilities of action outcomes in action domains, and goals represented as utility functions.

A final remark is in order. We define the planning framework model theoretically, independently of the language that can be used to describe the three components of a planning problem. For instance, different languages can be used to describe planning domains and plans, see, for instance [39, 26, 23, 38, 27]. This is the case also for goals. For instance, propositional logic can be used to represent reachability goals, while different temporal logics, such as LTL or CTL [21], or specialized goal languages (see, e.g., [17]) can express temporally extended goals.

In this Chapter, we start by defining a general framework that can model domains, plans and goals. In the next sections, we instantiate the framework to some specific cases along the different dimensions of the planning components: domains, plans, and goals. We conclude by reporting on state-of-the-art techniques in the field, and discussing some future research challenges.

22.2 The General Framework

In this section we define a general, formal framework for Automated Planning, which is able to capture a wide variety of planning problems addressed by the literature. In the next sections, we will show how the framework can by applied to capture the different specific problems.

22.2.1 Domains

A planning domain is defined in terms of its *states*, of the *actions* it accepts, and of the possible *observations* that the domain can exhibit. Some of the states are marked as *initial states* for the domain. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. Finally, an *observation function* defines what observations are associated to each state of the domain.

Definition 22.2.1 (Planning domain). *A* nondeterministic planning domain with partial observability *is a tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$, *where*:

- $\mathcal{S}$ *is the set of* states.
- $\mathcal{A}$ *is the set of* actions.
- $\mathcal{O}$ *is the set of* observations.
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states; *we require* $\mathcal{I} \neq \emptyset$.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ *is the* transition function; *it associates to each current state* $s \in \mathcal{S}$ *and to each action* $a \in \mathcal{A}$ *the set* $\mathcal{R}(s, a) \subseteq \mathcal{S}$ *of next states.*
- $\mathcal{X} : \mathcal{S} \to 2^{\mathcal{O}}$ *is the* observation function; *it associates to each state* s *the set of possible observations* $\mathcal{X}(s) \subseteq \mathcal{O}$.

We say that action a is executable *in state s if $\mathcal{R}(s, a) \neq \emptyset$. We require that in each state $s \in \mathcal{S}$ there is some executable action, that is some $a \in \mathcal{A}$ such that $\mathcal{R}(s, a) \neq \emptyset$. We also require that some observation is associated to each state $s \in \mathcal{S}$, that is, $\mathcal{X}(s) \neq \emptyset$.*

We say that $\mathcal{D}$ is finite state *if sets $\mathcal{S}$, $\mathcal{A}$, $\mathcal{O}$ are finite.*

Technically, a domain is described as a nondeterministic Moore machine, whose outputs (i.e., the observations) depend only on the current state of the machine, not on the input action. Uncertainty is allowed in the initial state and in the outcome of action execution. Also, the observation associated to a given state is not unique. This allows modeling noisy sensing and lack of information.

22.2.2 Plans and Plan Executions

A *plan* is a definition of the next action to be performed on a planning domain in a specific situation. A situation can be defined as the past history of the interactions of the (executor of the) plan with the planning domain. In the initial situation, the only information available to the executor is the initial (nondeterministic) observation o_0, and the executor reacts triggering action a_1. This leads to a new (nondeterministic) observation o_1, to which the executor reacts with an action a_2, which leads to a new (nondeterministic) observation o_2. This alternation of observations and actions can go on infinitely, or can stop when the executor stops triggering new actions.

Formally, we will define a plan as a partial function $\pi : \mathcal{O}^+ \rightharpoonup \mathcal{A}$ that associates an action $\pi(w)$ to a sequence of observations $w = o_0 o_1 \ldots o_n$. This way, the alternation of outputs and actions just described is $o_0 a_1 o_1 a_2 \ldots o_n$, where $a_{i+1} = \pi(o_0 o_1 \ldots o_i)$.

Definition 22.2.2 (Plan). *A* plan *for planning domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ is a partial function $\pi : \mathcal{O}^+ \rightharpoonup \mathcal{A}$ such that*:

- *if $o_0 o_1 \ldots o_n \in \mathrm{dom}(\pi)$ with $n > 0$, then $o_0 o_1 \ldots o_{n-1} \in \mathrm{dom}(\pi)$.*

If $\pi(w)$ is defined for some $w = o_0 o_1 \ldots o_n$, then we denote with $\pi^*(w)$ the sequence of outputs and actions $o_0 a_1 o_1 a_2 \ldots o_n$ such that $a_{i+1} = \pi(o_0 o_1 \ldots o_i)$ for $i = 1..n$.

Notice that the previous definition ensures that, if a plan defines an action to be executed for a sequence of observations, then an action is defined also for all the nonempty prefixes of the sequence.

Since we consider nondeterministic planning domains, the execution of an action may lead to different outcomes, and observations associated to these outcomes are also nondeterministic. Therefore, the execution of a plan on a planning domain can be described as a tree, where the branching corresponds to the different states reached by executing the planned action, and by the observations obtained from these states.

Formally, we define a *tree* τ with nodes labeled on set Σ (or Σ-labeled tree) as a subset of Σ^+ such that, if $\omega \cdot \sigma \in \tau$, with $\omega \in \Sigma^+$ and $\sigma \in \Sigma$, then also $\omega \in \tau$. Notice that tree τ can have finite branches—corresponding to strings ω that cannot be further extended in τ—as well as infinite branches—whenever there are sequences of strings $\omega_1, \omega_2, \ldots, \omega_n, \ldots$ such that ω_i is a strict prefix of ω_{i+1}.

We can now define an execution tree as a $(\mathcal{S}\times\mathcal{O})$-labeled tree, where component Σ of the label of the tree corresponds to a state in the planning domain, while component $\mathcal{O}$ describes the observation obtained from that state.

Definition 22.2.3 (Execution tree). *The* execution tree *for domain* $\mathcal{D} = \langle\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$ *and plan* π *is the* $(\mathcal{S}\times\mathcal{O})$*-labeled tree* τ *defined as follows*:

- $(s_0, o_0) \in \tau$, *where* $s_0 \in \mathcal{I}$ *and* $o_0 = \mathcal{X}(s_o)$;
- *if* $(s_0, o_0)(s_1, o_1)\ldots(s_n, o_n) \in \tau$, $\pi(o_0 o_1 \ldots o_n) = a_n$, $s_{n+1} \in \mathcal{R}(s_n, a_n)$ *and* $o_{n+1} \in \mathcal{X}(s_{n+1})$, *then* $(s_0, o_0)(s_1, o_1)\ldots(s_n, o_n)(s_{n+1}, o_{n+1}) \in \tau$.

Not all plans can be executed on a given domain. Indeed, it might be possible that the actions prescribed cannot be executed in all the states. We now define *executable* plans as those for which the triggered action is always executable on the domain.

Definition 22.2.4 (Executable plan). *Let* $\mathcal{D} = \langle\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$ *be a planning domain and* π *be a plan for* $\mathcal{D}$. *We say that* π *is* executable *if the following condition holds on the execution tree* τ *for* $\mathcal{D}$ *and* π:

- *if* $(s_0, o_0)(s_1, o_1)\ldots(s_n, o_n) \in \tau$ *and* $\pi(o_0 o_1 \ldots o_n) = a_n$ *then* $\mathcal{R}(s_n, a_n) \neq \emptyset$.

22.2.3 Goals and Problems

A *planning problem* consists of a planning domain and of a goal g that defines the set of desired behaviors. In the following, we assume that goal g defines a set of execution trees, namely the execution trees that exhibit the behaviors described by the goal (we say that these execution trees satisfy the goal).

Definition 22.2.5 (Planning problem). *A* planning problem *is a pair* $(\mathcal{D}, g)$, *where* $\mathcal{D} = \langle\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$ *is a planning domain and* g *is a set of* $(\mathcal{S}\times\mathcal{O})$*-labeled trees. A* solution *to planning problem* (D, g) *is a plan* π *such that the execution tree for* π *satisfies goal* g.

22.3 Strong Planning under Full Observability

The first problem we address is the problem of strong planning under full observability. This problem can be defined restricting the framework with two assumptions, one on the planning domain, and one on the goal.

The first assumption is that the domain is fully observable. This means that we can assume that execution will have no run-time uncertainty whatsoever on the reached state: before attempting an action, the executor will know precisely the state of the domain. Intuitively, this can be modeled by letting the set of observations to coincide with the set of states, and by assuming that the observation relation is actually an identity function. Formally,

Definition 22.3.1 (Fully observable domain). *A planning domain* $\mathcal{D} = \langle\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$ *is fully observable iff* $\mathcal{O} = \mathcal{S}$ *and* $\mathcal{X}(s) = s$.

For simplicity, in the following we will assume that fully observable planning domains are defined as tuples $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{R} \rangle$.

The second assumption is that we are interested in *strong* solutions, that guarantee that a set of target states will be reached in a finite number of steps, regardless of initial uncertainty in the initial states, and of nondeterministic action effects.

Definition 22.3.2 (Goal for strong planning). *Let $\mathcal{G}$ be a set of states. An execution tree π is a solutions to the strong planning problem $\mathcal{G}$ iff every branch of π is finite and ends in a state in $\mathcal{G}$.*

In this setting, we can restrict our solutions to a very specific form of plans, i.e., memoryless policies. Memoryless policies are plans where the selection of actions depends on the last observation only.

Definition 22.3.3 (Memoryless plans). *Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ be a finite state domain. Plan π for domain $\mathcal{D}$ is* memoryless *if, for all ω, ω', and o, $\pi(\omega o) = \pi(\omega' o)$.*

Intuitively, memoryless plans are enough to solve the problem due to full observability, and to the simplicity of the goal.

Memoryless plans can be described in a compact way as a partial function, called state-action table, mapping states to the actions to be executed in such states. More precisely, a state-action table SA is a subset of $\mathcal{S} \times \mathcal{A}$, and a deterministic state-action table is a state-action table SA such that $\langle s, a \rangle \in \mathrm{SA}$ and $\langle s, a' \rangle \in \mathrm{SA}$ imply $a = a'$. The definition of a plan corresponding to a deterministic state-action table is trivial.

We now describe an algorithm for strong planning. The algorithm operates on the planning problem: the sets of the initial states $\mathcal{I}$ and of the goal states $\mathcal{G}$ are explicitly given as input parameters, while the domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{I}, \mathcal{R} \rangle$ is assumed to be globally available to the invoked subroutines. The algorithm either returns a solution state-action table, or a distinguished value for state-action tables, called $\perp$, used to represent search failure. In particular, we assume that $\perp$ is different from the empty state-action table, that we will denote with $\emptyset$.

The algorithm, presented in Fig. 22.1, is based on a breadth-first search proceeding backwards from the goal, towards the initial states. At each iteration step, the set of states for which a solution has been already found is used as a target for the expansion preimage routine at line 5, that returns a new “slice” to be added to the state-action table under construction. Functions StrongPreImage is defined as follows:

$$\textsc{StrongPreImage}(S) \doteq \{\langle s, a \rangle \colon \emptyset \neq \mathcal{R}(s, a) \subseteq \mathcal{S}\}.$$

StrongPreImage(S) returns the set of state-action pairs $\langle s, a \rangle$ such that the execution of a in s is guaranteed to lead to states inside S, regardless of nondeterminism. We contrast the definition of StrongPreImage with the WeakPreImage function (that will be used in the following sections):

$$\textsc{WeakPreImage}(S) \doteq \{\langle s, a \rangle \colon \mathcal{R}(s, a) \cap S \neq \emptyset\}.$$

Intuitively, WeakPreImage(S) returns the set of state-action pairs $\langle s, a \rangle$ such that the execution of a in s *may* lead inside S, but it is not guaranteed to do so.

```
1   function StrongPlan(I, G);
2     OldSA := ⊥;
3     SA := ∅;
4     while (OldSA ≠ SA ∧ I ⊈ (G ∪ StatesOf(SA))) do
5       PreImage := StrongPreImage(G ∪ StatesOf(SA));
6       NewSA := PruneStates(PreImage, G ∪ StatesOf(SA));
7       OldSA := SA;
8       SA := SA ∪ NewSA;
9     done;
10    if (I ⊆ (G ∪ StatesOf(SA))) then
11      return SA;
12    else
13      return ⊥;
14    fi;
15  end;
```

Figure 22.1: The algorithm for strong planning.

In the strong planning algorithm, function StrongPreImage is called using as target the goal states G and the states that are already in the state-action table SA: these are the states for which a solution is already known. The returned preimage PreImage is then passed to function PruneStates, defined as follows:

$$\text{PruneStates}(\pi, S) \doteq \{\langle s, a\rangle \in \pi\colon\ s \notin S\}.$$

This function removes from the preimage table all the pairs $\langle s, a\rangle$ such that a solution is already known for s. This pruning is important to guarantee that only the shortest solution from any state appears in the state-action table. The termination test requires that the initial states are included in the set of accumulated states (i.e., $G \cup \text{StatesOf}(\text{SA})$), or that a fix-point has been reached and no more states can be added to state-action table SA. In the first case, the returned state-action table is a solution to the planning problem. In the second case, no solution exists.

Notice that the state-action table SA computed by the algorithm is not necessarily deterministic. However, a deterministic state-action table can be obtained from SA associating to state s an arbitrary action from set $\{a\colon\ \langle s, a\rangle \in \text{SA}\}$, whenever this set is notempty.

22.4 Strong Cyclic Planning under Full Observability

Strong cyclic planning can be defined in the same setting as strong planning: domains are fully observable, and plans are memoryless policies. The variation is in the set of acceptable executions: here, in addition to executions that terminate in the goal, we also accept infinite executions (e.g., that can loop for ever), with the proviso that the chance of reaching the goal is retained.

Definition 22.4.1 (Goal for strong cyclic planning). *Let $\mathcal{G}$ be a set of states. Then an execution tree π is a solution to the strong cyclic planning problem $\mathcal{G}$ iff every path in π either ends in a state in $\mathcal{G}$, or each of its finite prefixes has a suffix that ends in $\mathcal{G}$.*

We now present an algorithm for strong cyclic planning. The main difference with the algorithm presented in previous section is that here the resulting plans allow for infinite behaviors: loops must no longer be eliminated, but rather controlled, i.e., only certain, "good" loops must be kept. Infinite executions are accepted only if they correspond to "unlucky" patterns of nondeterministic outcomes, and if a goal state can be reached from each state of the execution under different patterns of nondeterministic outcomes.

The strong cyclic planning algorithm is presented in Fig. 22.2. The algorithm starts to analyze the universal state-action table with respect to the problem being solved, and eliminates all those state-action pairs which are discovered to be source of potential "bad" loops, or to lead to states which have been discovered not to allow for a solution. With respect to the algorithms presented in previous section, here the set of states associated with the state-action table being constructed is reduced rather than being extended: this approach amounts to computing a greatest fix-point.

The starting state-action table in function STRONGCYCLICPLAN is the universal state-action table UnivSA. It contains all state-action pairs that satisfy the applicability conditions:

$$\text{UnivSA} \doteq \{\langle s,\ a\rangle \colon \mathcal{R}(s,a) \neq \emptyset\}.$$

The "elimination" phase, where unsafe state-action pairs are discarded, corresponds to the while loop of function STRONGCYCLICPLAN. It is based on the repeated application of the functions PRUNEOUTGOING and PRUNE UNCONNECTED. The role of PRUNEOUTGOING is to remove all those state-action pairs which may lead out of $G \cup$ STATESOF(SA), which is the current set of potential solutions. Because of the elimination of these actions, 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 is quit when convergence is reached. The resulting state-action table is guaranteed to generate executions which either terminate in the goal or loop forever on states from which it is possible to reach the goal. Function STRONGCYCLICPLAN then checks whether the computed state-action table SA defines a plan for all the initial states, i.e., $\mathcal{I} \subseteq \mathcal{G} \cup$ STATESOF(SA). If this is not the case a failure is returned.

The state-action table obtained after the elimination loop is not necessarily a valid solution for the planning problem. Indeed, it may contain state-action pairs that, while preserving the reachability of the goal, still do not perform any progress toward it. In the strong cyclic planning algorithm, function REMOVENONPROGRESS on line 9 takes care of removing all those actions from a state whose outcomes do not lead to any progress toward the goal. This function is similar to the strong planning algorithm: it iteratively extends the state-action table by considering states at an increasing distance from the goal. In this case, however, a weak preimage is computed at any iteration step, since it is sufficient to guarantee progress towards the goal for *some* outcome of action execution. Moreover, the computed weak preimage is restricted to the state-action pairs that appear in the input state-action table, and hence that are "safe" according to the elimination phase.

```
1  function StrongCyclicPlan(I, G);
2    OldSA := ∅;
3    SA := UnivSA;
4    while (OldSA ≠ SA) do
5      OldSA := SA;
6      SA := PruneUnconnected(PruneOutgoing(SA, G), G);
7    done;
8    if (I ⊆ (G ∪ StatesOf(SA))) then
9      return RemoveNonProgress(SA, G);
10   else
11     return ⊥;
12   fi;
13 end;
```

```
1  function PruneUnconnected(SA, G);
2    NewSA := ∅;
3    repeat
4      OldSA := NewSA;
5      NewSA := SA ∩ WeakPreImage(G ∪ StatesOf(NewSA));
6    until (OldSA = NewSA);
7    return NewSA;
8  end;
```

```
1  function PruneOutgoing(SA, G);
2    NewSA := SA \ ComputeOutgoing(SA, G ∪ StatesOf(SA));
3    return NewSA; ;
4  end;
```

```
1  function RemoveNonProgress(SA, G);
2    NewSA := ∅;
3    repeat
4      PreImage := SA ∩ WeakPreImage(G ∪ StatesOf(NewSA));
5      OldSA := NewSA;
6      NewSA := NewSA ∪ PruneStates(PreImage, G ∪ StatesOf(NewSA));
7    until (OldSA = NewSA);
8  return NewSA;
9  end;
```

Figure 22.2: The algorithm for strong cyclic planning.

Functions PruneOutgoing, PruneUnconnected, and RemoveNonProgress, also presented in Fig. 22.2, exploit primitives WeakPreImage and Prune States, already defined in Section 22.3, and the primitive ComputeOutgoing, that takes as input a state-action table SA and a set of states S, and returns those state-action pairs which are not guaranteed to result in states in S:

$$\text{ComputeOutgoing}(\text{SA}, S) \doteq \left\{\langle s,\ a\rangle \in \text{SA}:\ \mathcal{R}(s, a) \not\subseteq S\right\}.$$

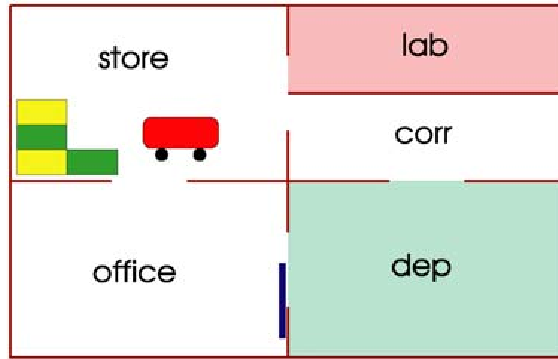

Figure 22.3: A simple nondeterministic planning domain.

22.5 Planning for Temporally Extended Goals under Full Observability

We now extend the problem of planning in fully observable domains by considering temporal goals. Under the hypothesis of full observability, the planning domain is still the same as the one formalized in Section 22.3. Plans cannot instead be limited to memoryless policies. In order to satisfy temporal goals, the plan function needs to select actions depending on the previous execution steps. Intuitively, this is due to the fact that plans need to keep track of which part of the temporal goal has been satisfied, and which one is still open. Consider for instance the following example.

Example 22.5.1. A simple domain is shown in Fig. 22.3. It consists of a building of five rooms, namely a store, a department dep, a laboratory lab, an office, and a corridor corr. A robot can move between the rooms. The laboratory is a dangerous room it is not possible to exit from. For the sake of simplicity, we do not model explicitly the objects, but only the movements of the robot. Between rooms office and dep, there is a door that the robot cannot control. Therefore, an *east* action from room office successfully leads to room dep only if the door is open. Another nondeterministic outcome occurs when the robot tries to move *east* from the store: in this case, the robot may end nondeterministically either in room corr or in room lab. The transition graph for the domain is represented in Fig. 22.4.

Consider now the goal of going from the corridor corr to room dep and then back to room store. The action to execute in room corr depends on whether the robot has already reached room dep and is going back to the store.

Plans are therefore *regular plans* that take into account previous execution steps and that are instantiated to the case of fully observable domains.

Definition 22.5.2 (Regular plan). *Plan π for finite state domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ is* regular *if there is a finite set of contexts C and a function $f : \mathcal{O}^+ \to C$ such that*:

- *if* $f(\omega) = f(\omega')$ *then* $\pi(\omega) = \pi(\omega')$,
- *if* $f(\omega) = f(\omega')$, *then* $f(\omega o) = f(\omega' o)$.

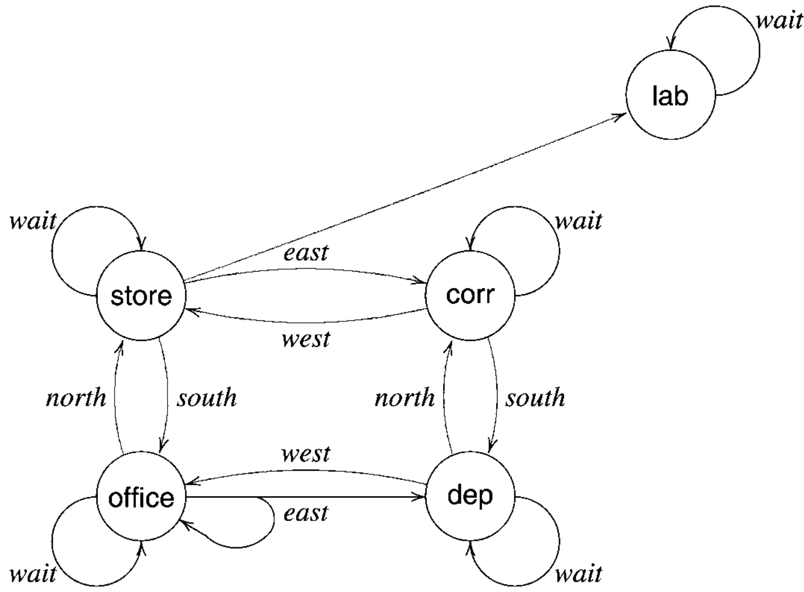

Figure 22.4: The transition graph of the navigation domain.

It is easy to see that regular plans can be defined specifying: (1) the finite set of contexts C, (2) an initialization function $init : \mathcal{O} \rightharpoonup C$ defining the initial context given the initial observation, and (3), an evolution function $evolve : C \times \mathcal{O} \rightharpoonup C$, defining the next context, given the current context and the observation.

In the following, we prefer a different alternative characterization of regular plans for fully observable domains, which is more adequate for the planning algorithm that we are going to define. More precisely, a regular plan can be defined in terms of an *action function* that, given a state and an *execution context*, specifies the action to be executed, and in terms of a *context function* that, depending on the action outcome, specifies the next execution context.

Definition 22.5.3 (Regular plans (for temporally extended goals)). *A* plan *for a fully observable domain D is a tuple* $\langle C, c_0, act, ctxt\rangle$, *where*:

- C *is a finite set of* (*execution*) *contexts*,
- $c_0 \in C$ *is the initial context*,
- $act : \mathcal{S} \times C \rightharpoonup \mathcal{A}$ *is the action function*,
- $ctxt : \mathcal{S} \times C \times \mathcal{S} \rightharpoonup C$ *is the context function*.

We require that a plan satisfies the following conditions:

1. $act(s_0, c_0)$ *is defined for each* $s_0 \in \mathcal{I}$;
2. *whenever* $act(s, c) = a$ *and* $ctxt(s, c, s') = c'$, *then* $\mathcal{R}(s, a) \neq \emptyset$ *and* $s' \in \mathcal{R}(s, a)$;

$act(\mathsf{store}, c_0) = south$	$ctxt(\mathsf{store}, c_0, \mathsf{office}) = c_0$
$act(\mathsf{office}, c_0) = east$	$ctxt(\mathsf{office}, c_0, \mathsf{dep}) = c_1$
	$ctxt(\mathsf{office}, c_0, \mathsf{office}) = c_0$
$act(\mathsf{dep}, c_1) = west$	$ctxt(\mathsf{dep}, c_1, \mathsf{office}) = c_1$
$act(\mathsf{office}, c_1) = north$	$ctxt(\mathsf{office}, c_1, \mathsf{store}) = c_1$
$act(\mathsf{store}, c_1) = wait$	$ctxt(\mathsf{store}, c_1, \mathsf{store}) = c_1$

Figure 22.5: An example of plan.

3. *whenever* $act(s, c) = a$ *and* $s' \in \mathcal{R}(s, a)$, *then there is some context* c' *such that* $ctxt(s, c, s') = c'$ *and* $act(s', c')$ *is defined.*

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 to each reached state s' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan. We require plans to be defined in all the initial states (Condition 1 in Definition 22.2.2), to be *executable*, i.e., the actions should be applicable and contexts should be defined over states that are the results of applying the actions (Condition 2), and to be *complete*, i.e., a plan should always specify how to proceed for all the possible outcomes of any action in the plan (Condition 3).

Example 22.5.4. An example of a plan is shown in Fig. 22.5. The plan leads the robot from room store to room dep going through the office, and then back to the store, again going through the office. Two contexts are used, namely c_0 when the robot is going to the dep and c_1 when the robot is going back to the store. This allows the plan to execute different actions in state office and in state store.

As discussed in Section 22.2, the execution of a plan can be described as a labeled tree. In the case of a fully observable domain, observations are not important, and the execution of a plan can be simply described as a $\mathcal{S}$-labeled tree.

Definition 22.5.5 (Execution tree (in a fully observable domain)). *The* execution tree *for a fully observable domain* $\mathcal{D}$ *and regular plan* π *is the* $\mathcal{S}$*-labeled tree* τ *defined as follows*:

- $s_0 \in \tau$, *where* $s_0 \in \mathcal{I}$;
- *if* $s_0 s_1 \dots s_n \in \tau$, $\pi(s_0 s_1 \dots s_n) = a_n$, $s_{n+1} \in \mathcal{R}(s_n, a_n)$, *then* $s_0 s_1 \dots s_n s_{n+1} \in \tau$.

Notice that, due to Condition 3 in Definition 22.2.2, execution trees obtained from regular plans do not contain finite paths.

We describe temporally extended goals by means of formulae in a temporal logic. In this setting, we use Computation Tree Logic (CTL) [21] that enables us to characterize the corresponding set of trees.

Definition 22.5.6 (CTL goal). *A CTL goal is defined by the following grammar, where s is a state of the domain $\mathcal{D}$*[1]:

$$
\begin{aligned}
g &::= p \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \\
&\quad\ \mathrm{A}(g\,\mathrm{U}\,g) \mid \mathrm{E}(g\,\mathrm{U}\,g) \mid \mathrm{A}(g\,\mathrm{W}\,g) \mid \mathrm{E}(g\,\mathrm{W}\,g) \\
p &::= s \mid \neg p \mid p \wedge p
\end{aligned}
$$

CTL combines temporal operators and path quantifiers. “X”, “U”, and “W” are the “next time”, “(strong) until”, and “weak until” temporal operators, respectively. “A” and “E” are the universal and existential path quantifiers, where a path is an infinite sequence of states. Formulas $\mathrm{AF}\, g$ and $\mathrm{EF}\, g$ (where the temporal operator “F” stands for “future” or “eventually”) are abbreviations of $\mathrm{A}(\top\,\mathrm{U}\,g)$ and $\mathrm{E}(\top\,\mathrm{U}\,g)$, respectively. $\mathrm{AG}\, g$ and $\mathrm{EG}\, g$ (where “G” stands for “globally” or “always”) are abbreviations of $\mathrm{A}(g\,\mathrm{W}\,\bot)$ and $\mathrm{E}(g\,\mathrm{W}\,\bot)$, respectively. A remark is in order. Even if negation $\neg$ is allowed only in front of basic propositions, it is easy to define $\neg g$ for a generic CTL formula g, by “pushing down” the negations: for instance, $\neg\mathrm{AX}\, g \equiv \mathrm{EX}\,\neg g$ and $\neg\mathrm{A}(g_1\,\mathrm{W}\,g_2) \equiv \mathrm{E}(\neg g_2\,\mathrm{U}\,(\neg g_1 \wedge \neg g_2))$.

We now define valid plans, i.e., plans that satisfy CTL goals, i.e., we define $\tau \models g$, where τ is the execution tree of a plan π for domain $\mathcal{D}$, and g is a CTL goal. The definition of predicate $\models$ is based on the standard semantics of CTL [21].

Definition 22.5.7 (Valid plan for a CTL goal). *Let π be a plan for domain $\mathcal{D}$. Let τ be the execution tree of π in domain $\mathcal{D}$. Let n be a node of τ.*

We define $\tau, n \models g$ as follows.

- $\tau, n \models s$ *iff* $n = s$.
- $\tau, n \models \neg s$ *if* $n \neq s$.
- $\tau, n \models g \wedge g'$ *if* $\tau, n \models g$ *and* $\tau, s \models g'$.
- $\tau, n \models g \vee g'$ *if* $\tau, n \models g$ *or* $\tau, n \models g'$.
- $\tau, n \models \mathrm{AX}\, g$ *if for all* n' *that are successors nodes of* n *in* τ, *then* $\tau, n' \models g$.
- $\tau, n \models \mathrm{EX}\, g$ *if there is some successor node* n' *of* n *in* τ *such that* $\tau, n' \models g$.
- $\tau, n \models \mathrm{A}(g\,\mathrm{U}\,g')$ *if for all paths* $n_0 n_1 n_2 \ldots$ *in* τ *with* $n = n_0$ *there is some* $i \geqslant 0$ *such that* $\tau, n_i \models g'$ *and* $\tau, n_j \models g$ *for all* $0 \leqslant j < i$.
- $\tau, n \models \mathrm{E}(g\,\mathrm{U}\,g')$ *if there is some path* $n_0 n_1 n_2 \ldots$ *in* τ *with* $n = n_0$ *and some* $i \geqslant 0$ *such that* $\tau, n_i \models g'$ *and* $\tau, n_j \models g$ *for all* $0 \leqslant j < i$.
- $\tau, n \models \mathrm{A}(g\,\mathrm{W}\,g')$ *if for all paths* $n_0 n_1 n_2 \ldots$ *of* τ *with* $n = n_0$, *either* $\tau, n_j \models g$ *for all* $j \geqslant 0$, *or there is some* $i \geqslant 0$ *such that* $\tau, n_i \models g'$ *and* $\tau, n_j \models g$ *for all* $0 \leqslant j < i$.

---

[1] Here we chose to identify each state of the domain with a basic Boolean proposition of CTL formulas. Actually, we would need only $\lceil \log_2 |S| \rceil$ basic propositions, using a boolean encoding of the states.

- $\tau, n \models \mathrm{E}(g \,\mathrm{W}\, g')$ *if there is some path* $n_0 n_1 n_2 \ldots$ *in* τ *with* $n = n_0$ *such that either* $\tau, n_j \models g$ *for all* $j \geqslant 0$, *or there is some* $i \geqslant 0$ *such that* $\tau, n_i \models g'$ *and* $\tau, n_j \models g$ *for all* $0 \leqslant j < i$.

We define $\tau \models g$ *if* $\tau, n_0 \models g$ *for all the initial states* $n_0 = s_0 \in \mathcal{I}$ *of* $\mathcal{D}$.

A planning algorithm can search the state space by progressing CTL goals. A CTL goal g defines conditions on the current state and on the next states to be reached. Intuitively, if g must hold in s, then some conditions must be projected to the next states. The algorithm extracts the information on the conditions on the next states by "progressing" the goal g. For instance, if g is $\mathrm{EF}\, g'$, then either g' holds in s or $\mathrm{EF}\, g'$ must still hold in some next state, i.e., $\mathrm{EX}\,\mathrm{EF}\, g'$ must hold in q. One of the basic building blocks of the algorithm is the function *progr* that rewrites a goal by progressing it to next states. *progr* is defined by induction on the structure of goals.

- $progr(s, s') = \top$ if $s = s'$, $\bot$, otherwise;
- $progr(s, \neg s') = \neg progr(s, s')$;
- $progr(s, g_1 \wedge g_2) = progr(s, g_1) \wedge progr(s, g_2)$;
- $progr(s, g_1 \vee g_2) = progr(s, g_1) \vee progr(s, g_2)$;
- $progr(s, \mathrm{AX}\, g) = \mathrm{AX}\, g$ and $progr(s, \mathrm{EX}\, g) = \mathrm{EX}\, g$;
- $progr(s, \mathrm{A}(g \,\mathrm{U}\, g')) = (progr(s, g) \wedge \mathrm{AX}\,\mathrm{A}(g \,\mathrm{U}\, g')) \vee progr(s, g')$;
- $progr(s, \mathrm{E}(g \,\mathrm{U}\, g')) = (progr(s, g) \wedge \mathrm{EX}\,\mathrm{E}(g \,\mathrm{U}\, g')) \vee progr(s, g')$;
- $progr(s, \mathrm{A}(g \,\mathrm{W}\, g')) = (progr(s, g) \wedge \mathrm{AX}\,\mathrm{A}(g \,\mathrm{W}\, g')) \vee progr(s, g')$;
- $progr(s, \mathrm{E}(g \,\mathrm{W}\, g')) = (progr(s, g) \wedge \mathrm{EX}\,\mathrm{E}(g \,\mathrm{W}\, g')) \vee progr(s, g')$.

The formula $progr(s, g)$ can be written in a normal form. We write it as a disjunction of two kinds of conjuncts, those of the form $\mathrm{AX}\, f$ and those of the form $\mathrm{EX}\, h$, since we need to distinguish between formulas that must hold in all the next states and those that must hold in some of the next states:

$$progr(s, g) = \bigvee_{i \in I} \left(\bigwedge_{f \in A_i} \mathrm{AX}\, f \wedge \bigwedge_{h \in E_i} \mathrm{EX}\, h \right),$$

where $f \in A_i$ ($h \in E_i$) if $\mathrm{AX}\, f$ ($\mathrm{EX}\, h$) belongs to the ith disjunct of $progr(s, g)$. We have $|I|$ different disjuncts that correspond to alternative evolutions of the domain, i.e., to alternative plans we can search for. In the following, we represent $progr(s, g)$ as a set of pairs, each pair containing the A_i and the E_i parts of a disjunct:

$$progr(s, g) = \{(A_i, E_i) \mid i \in I\}$$

with $progr(s, \top) = \{(\emptyset, \emptyset)\}$ and $progr(s, \bot) = \emptyset$.

Given a disjunct (A, E) of $progr(s, g)$, we can define a function that assigns goals to be satisfied to the next states. We denote with assign-progr$((A, E), S)$ the set of all

the possible assignments $i : S \to 2^{A \cup E}$ such that each universally quantified goal is assigned to all the next states (i.e., if $f \in A$ then $f \in i(s)$ for all $s \in S$) and each existentially quantified goal is assigned to one of the next states (i.e., if $h \in E$ and $h \notin A$ then $f \in i(s)$ for one particular $s \in S$).

Given the two basic building blocks *progr* and assign-progr, we can now describe the planning algorithm build-plan that, given a goal g_0 and an initial state s_0, returns either a plan or a failure.[2] The algorithm is reported in Fig. 22.6. It performs a depth-first forward search: starting from the initial state, it picks up an action, progresses the goal to successor states, and iterates until either the goal is satisfied or the search path leads to a failure. The algorithm uses as the "contexts" of the plan the list of the active goals that are considered at the different stages of the exploration. More precisely, a context is a list $c = [g_1, \dots, g_n]$, where the g_i are the active goals, as computed by functions *progr* and assign-progr, and the order of the list represents the *age* of these goals: the goals that are active since more steps come first in the list.

The main function of the algorithm is function build-plan-aux(s, c, pl, open), that builds the plan for context c from state s. If a plan is found, then it is returned by the function. Otherwise, $\perp$ is returned. Argument pl is the plan built so far by the algorithm. Initially, the argument passed to build-plan-aux is pl $= \langle C, c_0, act, ctxt \rangle = \langle \emptyset, g_0, \emptyset, \emptyset \rangle$. Argument open is the list of the pairs state-context of the currently open problems: if $(s, c) \in$ open then we are currently trying to build a plan for context c in state s. Whenever function build-plan-aux is called with a pair state-context already in open, then we have a loop of states in which the same sub-goal has to be enforced. In this case, function is-good-loop((s, c), open) is called that checks whether the loop is valid or not. If the loop is good, plan pl is returned, otherwise function build-plan-aux fails.

Function is-good-loop computes the set loop-goals of the goals that are active during the whole loop: iteratively, it considers all the pairs (s', c') that appear in open up to the next occurrence of the current pair (s, c), and it intersects loop-goals with the set setof(c') of the goals in list c'. Then, function is-good-loop checks whether there is some strong until goal among the loop-goals. If this is a case, then the loop is bad: the semantics of CTL requires that all the strong until goals are eventually fulfilled, so these goals should not stay active during a whole loop. In fact, this is the difference between strong and weak until goals: executions where some weak until goal is continuously active and never fulfilled are acceptable, while the strong until should be eventually fulfilled if they become active.

If the pair (s, c) is not in open but it is in the plan pl (i.e., (s, c) is in the range of function *act* and hence condition "defined pl.$act[s, c]$" is true), then a plan for the pair has already been found in another branch of the search, and we return immediately with a success. If the pair state-context is neither in open nor in the plan, then the algorithm considers in turn all the executable actions a from state s, all the different possible progresses (A, E) returned by function *progr*, and all the possible assignments i of (A, E) to $\mathcal{R}(s, a)$. Function build-plan-aux is called recursively for each destination state in $s' \in \mathcal{R}(s, a)$. The new context is computed by function order-goals($i[s']$, c): this function returns a list of the goals in $i[s']$ that are ordered by

[2] It is easy to extend the algorithm to the case of more than one initial state.

```
1   function build-plan(s_0, g_0): Plan
2      return build-plan-aux(s_0, [g_0], ⟨∅, g_0, ∅, ∅⟩, ∅)
3
4   function build-plan-aux(s, c, pl, open): Plan
5      if (s, c) ∈ open then
6         if is-good-loop((s, c), open) then return pl
7            else return ⊥
8         if defined pl.act[s, c] then return pl
9         foreach a ∈ 𝒜(p) do
10           foreach(A, E) ∈ progr(s, c) do
11              foreach i ∈ assign-progr((A, E), ℛ(s, a)) do
12                 pl′ := pl
13                 pl′.C := pl′.C ∪ {c}
14                 pl′.act[s, c] := a
15                 open′ := conc((s, c′), open)
16                 foreach s′ ∈ ℛ(s, a) do
17                    c′ := order-goals(i[s′], c)
18                    pl′.ctxt[s, c, s′] := c′
19                    pl′ := build-plan-aux(s′, c′, pl′, open′)
20                    if pl′ = ⊥ then next i
21                 return pl′
22        return ⊥
23
24     function is-good-loop((s, c), open): boolean
25           loop-goals := setof(c)
26           while(s, c) ≠ head(open) do
27              (s′, c′) := head(open)
28              loop-goals := loop-goals ∩ setof(c′)
29              open := tail(open)
31           if ∃g ∈ loop-goals: g = A(_ U _) or g = E(_ U _) then
32              return false
33           else
34              return true
```

Figure 22.6: A planning algorithm for CTL goals.

their "age": namely those goals that are old (they appear in $i[s']$ and also in c) appear first, in the same order as in c, and those that are new (they appear in $i[s']$ but not in c) appear at the end of the list, in any order. Also, in the recursive call, argument pl is updated to take into account the fact that action a has been selected from state s in context g. Moreover, the new list of open problems is updated to conc$((s, c)$, open), namely the pair (s, c) is added in front of argument open.

Any recursive call of build-plan-aux updates the current plan pl$'$. If all these recursive calls are successful, then the final value of plan pl$'$ is returned. If any of the recursive calls returns $\bot$, then the next combination of assign decomposition, progress component and action is tried. If all these combinations fail, then no plan is found and $\bot$ is returned.

22.6 Conformant Planning

The problem of conformant planning is the result of the assumption that no observation is available at run time. In such a setting, the execution will have to proceed blindly, without the possibility to acquire any information. Intuitively, we model the absence of information by associating each state to the same observation.

Definition 22.6.1 (Unobservable domain). *A planning domain* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ *is unobservable iff* $\mathcal{O} = \{\bullet\}$ *and* $\mathcal{X}(s) = \bullet$.

Since only one observation is available, it conveys no information at all. Therefore, plans can only depend on the length of the history, since $\mathcal{O}^*$ is a sequence of bullets. In this setting, meaningful plans can be presented as sequences of actions.

Definition 22.6.2 (Sequential plan). *Let* $a_1, \ldots, a_n$ *be a sequence of actions. Then, the corresponding plan is defined for any history of length* $i \leqslant n$, *and returns* a_i.

The problem of conformant planning requires to find a *strong* solution, that guarantees goal achievement for all initial states, and for nondeterministic action effects.

Definition 22.6.3 (Goal for conformant planning). *Let* $\mathcal{G}$ *be a set of states. An execution tree* π *is a solution to a conformant planning problem* $\mathcal{G}$ *iff all the branches are finite and of the same length, and they all end in* $\mathcal{G}$.

At this point, it should be clear that the problem we are tackling is much harder than the classical planning problem. Suppose we are given a possible conformant plan, having a run from one initial state to the goal; we still have to check that it is a valid conformant plan, i.e., it is applicable in each state in $\mathcal{I}$, and that the final state of each run is in $\mathcal{G}$. In fact, conformant planning reduces to classical planning if the set of initial states is a singleton and the domain is deterministic.

We notice that the branching in the execution tree of a sequential plan is only due to the nondeterminism in the action effects, since the same action is executed regardless of the activity of the system. Therefore, the ith level in the tree represents all the possible system states which can be reached by the domain after the execution of the first n actions in the plan. We also notice that such states are in fact "indistinguishable". Based on this observation, conformant planning can be tackled as search in the space of *belief states*. A belief state is a nonempty set of states, intuitively expressing a condition of uncertainty, by collecting together all the states which are indistinguishable. Intuitively, a belief state can be used to capture the ith level of the execution tree associated with a sequence of actions.

Belief states are a convenient representation mechanism: instead of analyzing all the traces associated with a candidate plan, the associated set of states can be collected into a belief state. In this setting, conformant planning reduces to deterministic search in the space of belief states, called the *belief space*. The belief space for a given domain is basically the power-set of the set of states of the domain. For technical reasons, we explicitly restrict our reasoning to nonempty belief states, and define the belief space as $\mathit{Pow}^+(\mathcal{S}) \doteq \mathit{Pow}(\mathcal{S}) \setminus \emptyset$.

```
1  function HEURCONFORMANTFWD(𝓘, 𝓖)
2     Open := {⟨𝓘, ε⟩};
3     Closed := ∅;
4     Solved := False;
5     while (Open ≠ ∅ ∧ ¬Solved) do
6        ⟨Bs, π⟩ := EXTRACTBEST(Open);
7        INSERT(⟨Bs, π⟩, Closed);
8        if Bs ⊆ 𝓖 then
9           Solved := True; Solution := π;
10       else
11          BsExp := FWDEXPANDBS(Bs);
12          BsPList := PRUNEBSEXPANSION(BsExp, Closed);
13          for ⟨Bs_i, a_i⟩ in BsPList do
14 INSERT(⟨Bs_i, π; a_i⟩, Open)
15          endfor
16       fi
17    done
18    if Solved then
19       return Solution;
20    else
21       return ⊥;
22    fi
23 end
```

Figure 22.7: The forward conformant planning algorithm.

The execution of actions is lifted from states to belief states by the following definition.

Definition 22.6.4 (Action applicability, execution). *An action a is applicable in a belief state Bs iff a is applicable in every state in Bs. If a is applicable in a belief state Bs, its execution in Bs, written $Exec(a, Bs)$, is defined as follows*:

$$Exec(a, Bs) \doteq \left\{s' \colon\ s \in Bs \text{ and } s' \in \mathcal{R}(s, a)\right\}.$$

Definition 22.6.5 (Plan applicability, execution). *The execution of plan π in a belief state Bs, written $Exec(\pi, Bs)$, is defined as follows*:

$$\begin{aligned} &Exec(\varepsilon, Bs) &&\doteq Bs, \\ &Exec(\pi, \bot) &&\doteq \bot, \\ &Exec(a; \pi, Bs) &&\doteq \bot, \ \text{ if } a \text{ is not applicable in } Bs, \\ &Exec(a; \pi, Bs) &&\doteq Exec(\pi, Exec(a, Bs)), \ \text{ otherwise}. \end{aligned}$$

$\bot$ is a distinguished symbol representing violation of action applicability. Plan π is applicable in a belief state Bs iff $Exec(\pi, Bs) \neq \bot$.

Fig. 22.7 depicts an algorithm for conformant planning. The algorithm searches the belief space, proceeding forwards from the set of initial states $\mathcal{I}$ towards the goal $\mathcal{G}$, and can be seen as a standard best-first algorithm, where search nodes are (uniquely

indexed by) belief states. *Open* contains a list of open nodes to be expanded, and *Closed* contains a list of closed nodes that have already been expanded. After the initialization phase, *Open* contains (the node indexed by) $\mathcal{I}$, while *Closed* is empty. The algorithm then enters a loop, where it extracts a node from the open list, stores it into the closed list, and checks if it is a success node (line 8) (i.e., it a subset of $\mathcal{G}$); if so, a solution has been found and the iteration is exited. Otherwise, the successor nodes are generated, and the ones that have already been expanded are pruned. The remaining nodes are stored in *Open*, and the iteration restarts. Each belief state *Bs* is associated with a plan π, that is applicable in $\mathcal{I}$, and that results exactly in *Bs*, i.e., $Exec(\pi, \mathcal{I}) = Bs$.

The algorithm loops (lines 5–17) until either a solution has been found ($Solved = True$) or all the search space has been exhausted ($Open = \emptyset$). A belief state *Bs* is extracted from the open pool (line 6), and it is inserted in closed pool (line 7). The belief states *Bs* is expanded (line 11) by means of the FWDEXPANDBS primitive. PRUNEBSEXPANSION (line 12) removes from the result of the expansion of *Bs* all the belief state that are in the *Closed*, and returns the pruned list of belief states. If *Open* becomes empty and no solution has been found, the algorithm returns with $\perp$ to indicate that the planning problem admits no conformant solution. The expansion primitive FWDEXPANDBS takes as input a belief state *Bs*, and builds a set of pairs $\langle Bs_i, a_i \rangle$ such that a_i is executable in *Bs* and the execution of a_i in *Bs* is contained in Bs_i. Notice that a_i is a conformant solution for the planning problem of reaching Bs_i from any nonempty subset of *Bs*.

$$\text{FWDEXPANDBS}(Bs) \doteq \big\{\langle Bs_i, a_i \rangle \colon Bs_i = Exec(a_i, Bs) \neq \perp\big\}.$$

Function PRUNEBSEXPANSION takes as input a result of an expansion of a belief state and *Closed*, and returns the subset of the expansion containing the pairs where each belief state has not been expanded. The PRUNEBSEXPANSION function can be defined as:

$$\text{PRUNEBSEXPANSION}(BsP, Closed) \doteq$$
$$\big\{\langle Bs_i, a_i \rangle \colon \langle Bs_i, a_i \rangle \in BsP, \text{ and } \langle Bs_i, \pi \rangle \in Closed \text{ for no plan } \pi\big\}.$$

When an annotated belief state $\langle Bs, \pi \rangle$ is inserted in *Open*, INSERT checks if another annotated belief state $\langle Bs, \pi' \rangle$ exists; the length of π and π' are compared, and only the pair with the shortest plan is retained.

Obviously, the algorithm described above can implement several search strategies, e.g., depth-first or breadth-first, depending on the implementation of the functions EXTRACTBEST (line 6) and INSERT (line 14). Variations based on backward search have been explored, but are not reported here for lack of space.

22.7 Strong Planning under Partial Observability

We consider now the problem of strong planning under partial observability. The problem is characterized by a generic domain, without constraints on the observations.

As in the case of strong planning under full observability, the acceptable execution trees can be presented by a set of goal states, that must be reached regardless of the initial condition and of nondeterministic action effects.

Definition 22.7.1 (Goal for strong planning under partial observability). *Let $\mathcal{G}$ be a set of states. An execution tree π is a solution to a problem of strong planning under partial observability $\mathcal{G}$ iff all the branches are finite, and they all end in $\mathcal{G}$.*

The availability of observations enables us to use richer plans than sequences: it is possible to delay at execution time the choice of the next action, depending on the observation, even in presence of uncertainty due to lack of full observability. Tree-shaped plans are needed, that define sequential courses of actions, which however depend on the observation that will arise at run time. Such tree-shaped plans correspond to the generic model of plans defined in Section 22.2, where observation histories identify specific courses of actions, with the only constraint that plans should not contain infinite branches.

Similarly to conformant planning, strong planning under partial observability can be solved by means of a search in the space of beliefs. In fact, conformant planning can be seen as a special case of planning with partial observability, where the observations are disregarded. The new element (with respect to conformant planning) is that the information conveyed by observations can be used to limit the uncertainty: the belief state modeling the current set of uncertainty can be reduced by ruling out the states that are incompatible with the observation. However, since the value of the observations that will occur during execution is not available at planning time, all possible options have to be taken into account: therefore, an observation "splits" a belief state in two belief states. These two belief states must both be solved in order to find a strong solution: for this reason, an AND/OR search in the space of beliefs is required (rather than a deterministic search).

Strong planning under full observability can also be seen as a special case of the problem addressed in this section. In fact, a memoryless policy can be mapped directly into a tree-shaped plan; however, the tree-shaped representation of the plan is potentially much more expensive than the memoryless policy representation (which is in essence a compact representation of a DAG). As far as the search algorithms are concerned, it would be possible to solve strong planning under full observability with an AND/OR search in the space of beliefs; however, full observability enables us to rule out uncertainty at execution time, so that all the belief states degenerate into singletons. In addition, the regressive search algorithm used with full observability is more amenable to deal with the branching factor due to nondeterminism than the progressive AND/OR search used with partial observability.

Finally, we notice that it would be in principle possible to reduce a problem of strong planning under partial observability to a problem of strong planning under full observability so that a regressive algorithm can be applied. However, this approach would results in an exponential blow up, due to the fact that a state would be required for every belief state in the original problem.

22.8 A Technological Overview

In this section, we overview the technologies underlying the main approaches to planning. Most of the work has been developed within the setting of classical planning. The first remark is that most of the planners work at the level of the language describing the domain, rather than explicitly manipulating an explicit representation of the

domain, which is in principle exponentially larger. Historically, the first classical planners were based on techniques such as regression and partial order planning, trying to exploit the causal connection between (sub)goals and action effects. The first computation breakthrough is due to the introduction of Planning Graphs [7], that enable for a “less intelligent” but efficient and compact overapproximation of the state space. Planning based on satisfiability decision procedures [36] is based on the generation of a propositional satisfiability problem, that is satisfiable only if the planning problem admits a solution (of given bound); the problem is then solved by means of efficient propositional SAT solvers, that are typically able to solve structured problem with large number of variables. Each of the problems is limited to bounded-length, i.e., it looks for a strong solution of specified length l. When this does not exist, the bound is iteratively increased l until a solution is found or a specified limit is reached. More recently, classical planning has been tackled by means of the integration of planning graphs with heuristic search techniques [31].

Some of the techniques developed in the setting of classical planning have also been used to tackle the problems described in this paper. The work in [54, 41, 45] pioneered the problem of generating conditional plans by extending the seminal approaches to classical planning (e.g., regression, partial order planning). These works address the problem in the case of partial observability by exploiting the idea of “sensing actions”, i.e., actions that when executed acquire information about the state of the domain. The proposed solutions never demonstrated experimentally the ability to scale up to nontrivial cases. Conformant and Sensorial Graphplan (CGP and SGP, resp.) [52, 55] were the first planners to extend planning graph techniques [7] to planning for reachability goals in the case of null observability and partial observability, respectively. These planners allowed for significant improvements in performance compared with previous extensions of classical planners. However, a practical weakness of this approach lies in the fact that algorithms are enumerative, i.e., a planning graph is built for each state that can be distinguished by observation. For this reason, both CGP and SGP are not competitive with more recent planners that address the same kind of problems. More recently, planning graphs used in cooperation with propositional satisfiability techniques in the CFF system, and efficient extension of the FF to deal with Conformant and Conditional planning [9, 32]. In CFF, planning graphs are used to compute heuristic measures and an AO*-like search is performed based on satisfiability techniques.

Among the planners based on reduction to a satisfiability problem, QBFPLAN [47] can deal with partial observability and reachability goals. The (bounded) planning problem is reduced to a QBF satisfiability problem, which is given in input to an efficient solver [48]. The approach exploits its symbolic approach to avoid exponential blow up caused by the explicit enumeration of states, but seems unable to scale up to large problems. Extensions to satisfiability techniques that can deal with conformant planning are reported in [11, 25].

A different approach to the problem of planning under partial observability is the idea of “Planning at the Knowledge Level”, implemented in the PKS planner [42]. This approach is based on a representation of incomplete knowledge and sensing at a higher level of abstraction. The extension presented in [43] provides a limited solution to the problem of deriving complete conclusions from observations.

Situation Calculus [46] provides a rather expressive formalism that has been used to do automated planning by reasoning in first order logic. In situation calculus it is possible to reason about actions with nondeterministic effects, which can be represented with disjunctive formulas. Partial observability has also been represented through knowledge or sensing actions [51, 53]. The problem of making situation calculus competitive in terms of performance with other more automated planning techniques has been addressed by providing the ability to specify a plans as programs (see, e.g., the work on Golog [37]).

DLVK [20] reduces conformant planning to answer set programming, by exploiting the Disjunctive Datalog-based system DVL. The produced answer set is to be interpreted as a (parallel) plan. The domain description language of DLVK is $\mathcal{K}$, where it is possible to express incomplete information, action nondeterminism, and initial uncertainty; in particular, in $\mathcal{K}$ it is possible to express transitions between knowledge states, where predicates can be three-valued (known true, known false, unknown). DLVK can produce conformant plans by requiring the underlying DLVK engine to perform "secure" reasoning, which amounts to iteratively producing weak plans, i.e., plans that are not guaranteed to reach the goal, and checking their security. DLVK tackles bounded conformant planning problems, i.e., the length of plans must be provided to the system.

Several approaches are based on the extension of techniques developed in model checking [16]. Among these, SIMPLAN [35] adopts an explicit-state representations, which limits its applicability to large state spaces. It was however the first planners to deal with nondeterministic domains and goals expressed in LTL, in the case of full observability. [18] presents an automata based approach to formalize planning in deterministic domains. The work in [28, 30, 29] presents a method where model checking with timed automata is used to verify that generated plans meet timing constraints.

A more recent approach is the one based on symbolic model checking. The work on the MBP planner has addressed the problem of planning for reachability goals under full observability [13], conformant planning [14], planning for reachability goals under partial observability [5], and planning for temporally extended goals [44, 17]. The underlying idea of symbolic model checking that is exploited in MBP is the following: sets of states are represented as propositional formulas, and search through the state space is performed as a set of logical transformations over propositional formulas. Such logical transformations are implemented in planning algorithms by exploiting Binary Decision Diagrams (BDDs) [10], that allow for a compact representation and effective manipulation of propositional formulae. MBP accepts as input languages for the description of the domain the $\mathcal{AR}$ action language [26]. A description of how $\mathcal{AR}$ is used as an input language for the MBP planner is given in [12, 15]. Several experimental comparisons show that symbolic model checking techniques are very competitive for planning under uncertainty.

Other BDD-based approaches to the problem of strong planning under partial observability have been proposed in the YKA [49] and JUSSIPOP planners [50]. These planners perform a backward search in the space of beliefs. As such, observations are used to recombine beliefs, according to a fixed cardinality-based heuristics. Some planners that are based on symbolic model checking techniques restrict to the case of full observability, see, e.g., UMOP [33, 34], or to classical planning, see, e.g., MIPS [19].

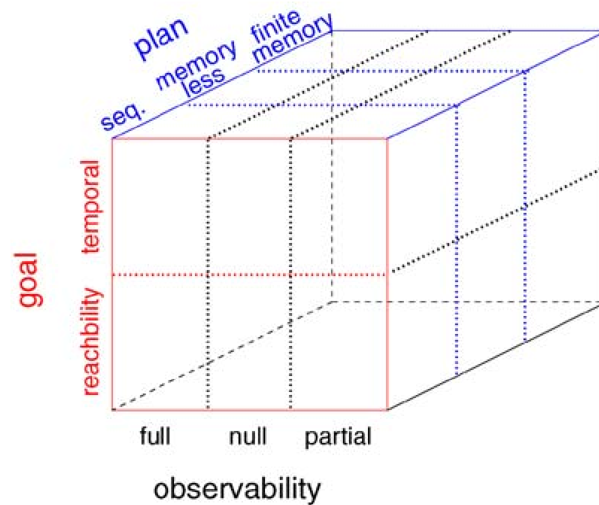

Figure 22.8: The different dimensions of a planning problem.

22.9 Conclusions

In this Chapter, we have proposed a general framework for planning, and instantiated it to some interesting planning problems. This is by no means exhaustive. Given the three planning components, domains, plans, and goals, one can think of different possible combinations (see Fig. 22.8).

We have started considering the case of *full observability*, and analyzing reachability and temporally extended goals. We have shown that memory-less plans are enough in the case of reachability goals, while finite-memory (or regular) plans are instead needed in the case of temporal goals. Of course, it would be possible to study the case in which we restrict acceptable solutions to memory-less plans, or plans with bounded memory. In fact, for temporally extended goals, some planning problems that can be solved with plans with finite but unbounded memory may have no solutions that are memory-less or bounded memory plans.

In addition, we have shown how *temporally extended goals* can be expressed in CTL. Different temporal logics can be used to express temporally extended goals, like Linear Time Logic (LTL), which has incomparable expressive power with respect to CTL (see [21] for a comparison), or more expressive temporal logics like CTL* or μ calculus, or specific languages for extended goals (see, e.g., [17, 2]).

In the case of *null observability*, we have just limited the analysis to reachability goals and sequential plans. We have not explored the case of null observability with temporally extended goals.

In the case of *partial observability*, the analysis is restricted to the case of reachability goals. Providing effective planning algorithm for the general case of partial observability and extended goals is a research challenge for the future. Some preliminary results in this directions are presented in [4, 6, 3].

Bibliography

[1] F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.

[2] C. Baral and J. Zhao. Goal specification in presence of non-deterministic actions. In R.L. de Mántaras and L. Saitta, editors. *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 273–277. IOS Press, 2004.

[3] P. Bertoli, A. Cimatti, and M. Pistore. Strong cyclic planning under partial observability. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI-06)*, pages 580–584, Riva del Garda, Italy, August 2006. IOS Press.

[4] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A framework for planning with extended goals and partial observability. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-03)*, June 2003.

[5] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Strong planning under partial observability. *Artificial Intelligence*, 170(4–5):337–384, 2006.

[6] P. Bertoli and M. Pistore. Planning with extended goals and partial observability. In *Proceedings the International Conference on Automated Planning and Scheduling (ICAPS-04)*, June 2004.

[7] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298, 1997.

[8] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: structural assumptions and computational leverage. *Journal of Artificial Intelligence Research (JAIR)*, 11:1–94, 1999.

[9] R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: a new approach. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, June 2004.

[10] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[11] C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147:85–118, 2003.

[12] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via model checking: a decision procedure for $\mathcal{AR}$. In *Proceeding of the Fourth European Conference on Planning (ECP-97)*, September 1997. Springer-Verlag.

[13] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1–2):35–84, July 2003.

[14] A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1–2):127–206, November 2004.

[15] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, 1998. AAAI-Press.

[16] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[17] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 447–454, Edmonton, Alberta, Canada, July 2002. AAAI-Press/The MIT Press.

[18] G. De Giacomo and M.Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *Proceeding of the Fifth European Conference on Planning (ECP-99)*, September 1999. Springer-Verlag.

[19] S. Edelkamp and M. Helmert. On the implementation of MIPS. In *AIPS-Workshop on Model-Theoretic Approaches to Planning*, pages 18–25, 2000.

[20] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence*, 144(1–2):157–211, 2003.

[21] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, 1990 (Chapter 16).

[22] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[23] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2–4):301–321, 1993.

[24] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, Inc., 2004.

[25] E. Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR-00)*, 2000.

[26] E. Giunchiglia, G.N. Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95(2):409–438, 1997.

[27] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, Menlo Park, July 26–30 1998. AAAI Press.

[28] R.P. Goldman, D.J. Musliner, K.D. Krebsbach, and M.S. Boddy. Dynamic abstraction planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97) and Ninth Innovative Applications of Artificial Intelligence Conference (IAAI-97)*, pages 680–686. AAAI Press, 1997.

[29] R.P. Goldman, D.J. Musliner, and M.J. Pelican. Using model checking to plan hard real-time controllers. In *Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, Breckeridge, CO, April 2000.

[30] R.P. Goldman, M. Pelican, and D.J. Musliner. Hard real-time mode logic synthesis for hybrid control: A CIRCA-based approach, March 1999. Working notes of the 1999 AAAI Spring Symposium on Hybrid Control.

[31] J. Hoffmann. FF: The fast-forward planning system. *AI Magazine*, 22(3):57–62, 2001.

[32] J. Hoffmann and R. Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-05)*, 2005.

[33] R.M. Jensen and M.M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research (JAIR)*, 13:189–226, 2000.

[34] R.M. Jensen, M.M. Veloso, and M.H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the Sixth European Conference on Planning (ECP-01)*, 2001.

[35] F. Kabanza, M. Barbeau, and R. St-Denis. Planning control rules for reactive agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[36] H.A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) and Eighth Innovative Applications of Artificial Intelligence Conference (IAAI-96)*, 1996.

[37] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: a logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–83, 1997.

[38] D. McDermott. PDDL—the planning domain definition language. Web page: http://www.cs.yale.edu/homes/dvm, 1998.

[39] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, 1989.

[40] J.S. Penberthy and D.S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 103–114, Cambridge, MA, October 1992. Morgan Kaufmann.

[41] M. Peot and D. Smith. Conditional nonlinear planning. In J. Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems (ICAPS-92)*, pages 189–197, College Park, MD, June 15–17 1992. Morgan Kaufmann.

[42] R. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, 2002.

[43] R. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, 2004.

[44] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*. AAAI Press, 2001.

[45] L. Pryor and G. Collins. Planning for contingency: A decision based approach. *J. of Artificial Intelligence Research*, 4:81–120, 1996.

[46] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[47] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[48] J. Rintanen. Improvements to the evaluation of quantified Boolean formulae. In T. Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1192–1197. Morgan Kaufmann Publishers, August 1999.

[49] J. Rintanen. Backward plan construction for planning as search in belief space. In *Proceedings of the International Conference on AI Planning and Scheduling (AIPS-02)*, 2002.

[50] J. Rintanen. Research on conditional planning with partial observability: The Jussi-POP/BBSP planning system. Web page: http://www.informatik.uni-freiburg.de/rintanen/planning.html, 2004.

[51] R.B. Scherl and H.J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1–2):1–39, 2003.

[52] D.E. Smith and D.S. Weld. Conformant Graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 889–896, Menlo Park, July 26–30, 1998. AAAI Press.

[53] T.C. Son and C. Baral. Formalizing sensing actions—a transition function based approach. *Artificial Intelligence*, 125(1–2):19–91, 2001.

[54] D. Warren. Generating conditional plans and programs. In *Proceedings of the Summer Conference on Artificial Intelligence and Simulation of Behaviour (AISB-76)*, 1976.

[55] D.S. Weld, C.R. Anderson, and D.E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 897–904, Menlo Park, July 26–30, 1998. AAAI Press.

This page intentionally left blank