CHAPTER 2

Representations for Classical Planning

2.1 Introduction

A necessary input to any planning algorithm is a description of the problem to be solved. In practice, it usually would be impossible for this problem description to include an explicit enumeration of all the possible states and state transitions: Such a problem description would be exceedingly large, and generating it would usually require more work than solving the planning problem. Instead, a problem representation is needed that does not explicitly enumerate the states and state transitions but makes it easy to compute them on-the-fly.

In this chapter, we discuss three different ways to represent classical planning problems. Each of them is equivalent in expressive power, in the sense that a planning domain represented in one of these representations can also be represented using either of the other representations.

1. In a *set-theoretic representation* (Section 2.2), each state of the world is a set of propositions, and each action is a syntactic expression specifying which propositions belong to the state in order for the action to be applicable and which propositions the action will add or remove in order to make a new state of the world.
2. In a *classical representation* (Section 2.3), the states and actions are like the ones described for set-theoretic representations except that first-order literals and logical connectives are used instead of propositions. This is the most popular choice for restricted state-transition systems.
3. In a *state-variable representation* (Section 2.5), each state is represented by a tuple of values of n state variables $\{x_1, \ldots, x_n\}$, and each action is represented by a partial function that maps this tuple into some other tuple of values of the n state variables. This approach is especially useful for representing domains in which a state is a set of attributes that range over finite domains and whose values change over time.

There are also various ways to extend these approaches. Examples include the use of logical axioms to infer things about states of the world and the use of more general logical formulas to describe the preconditions and effects of an action. Section 2.4 gives an overview of such approaches for classical representations.

2.2 Set-Theoretic Representation

In this section we discuss set-theoretic representations of classical planning problems. For brevity, we will usually call such problems *set-theoretic planning problems*, and we will refer to the representation scheme as *set-theoretic planning*.

2.2.1 Planning Domains, Problems, and Solutions

A set-theoretic representation relies on a finite set of proposition symbols that are intended to represent various propositions about the world.

Definition 2.1 Let $L = \{p_1, \ldots, p_n\}$ be a finite set of *proposition symbols*. A *set-theoretic planning domain* on L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^L$, i.e., each state s is a subset of L. Intuitively, s tells us which propositions currently hold. If $p \in s$, then p holds in the state of the world represented by s, and if $p \notin s$, then p does not hold in the state of the world represented by s.
- Each action $a \in A$ is a triple of subsets of L, which we will write as $a = (\text{precond}(a), \text{effects}^-(a), \text{effects}^+(a))$. The set $\text{precond}(a)$ is called the *preconditions* of a, and the sets $\text{effects}^+(a)$ and $\text{effects}^-(a)$ are called the *effects* of a. We require these two sets of effects to be disjoint, i.e., $\text{effects}^+(a) \cap \text{effects}^-(a) = \emptyset$. The action a is *applicable* to a state s if $\text{precond}(a) \subseteq s$.
- S has the property that if $s \in S$, then, for every action a that is applicable to s, the set $(s - \text{effects}^-(a)) \cup \text{effects}^+(a) \in S$. In other words, whenever an action is applicable to a state, it produces another state. This is useful to us because once we know what A is, we can specify S by giving just a few of its states.
- The state-transition function is $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if $a \in A$ is applicable to $s \in S$, and $\gamma(s, a)$ is undefined otherwise.

■

Definition 2.2 A *set-theoretic planning problem*[1] is a triple $\mathcal{P} = (\Sigma, s_0, g)$, where:

- s_0, the *initial state*, is a member of S.

1. When the meaning is clear from context, we will drop the adjective *set-theoretic* and just say *planning domain* and *planning problem*.

- $g \subseteq L$ is a set of propositions called *goal propositions* that give the requirements that a state must satisfy in order to be a goal state. The set of goal states is $S_g = \{s \in S \mid g \subseteq s\}$.

■

Definition 2.2 specifies what a planning problem is semantically, but it is not the syntactic specification we would use to describe a planning problem to a computer program. The latter will be discussed in Section 2.2.3.

Example 2.1 Here is one possible set-theoretic representation of the domain described in Example 1.1.

$L = \{$onground, onrobot, holding, at1, at2$\}$, where:

onground means that the container is on the ground;
onrobot means that the container is on the robot;
holding means that the crane is holding the container;
at1 means that the robot is at location 1; and
at2 means that the robot is at location 2.

$S = \{s_0, \ldots, s_5\}$, where:

$s_0 = \{$onground, at2$\}$; $s_1 = \{$holding, at2$\}$;
$s_2 = \{$onground, at1$\}$; $s_3 = \{$holding, at1$\}$; and
$s_4 = \{$onrobot, at1$\}$; $s_5 = \{$onrobot, at2$\}$.

$A = \{$take, put, load, unload, move1, move2$\}$ where:

take	=	({onground},	{onground},	{holding});
put	=	({holding},	{holding},	{onground});
load	=	({holding, at1},	{holding},	{onrobot});
unload	=	({onrobot, at1},	{onrobot},	{holding});
move1	=	({at2},	{at2},	{at1}); and
move2	=	({at1},	{at1},	{at2}).

■

Definition 2.3 A *plan* is any sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$, where $k \geq 0$. The *length* of the plan is $|\pi| = k$, the number of actions. If $\pi_1 = \langle a_1, \ldots, a_k \rangle$ and $\pi_2 = \langle a'_1, \ldots, a'_j \rangle$ are plans, then their *concatenation* is the plan $\pi_1 \cdot \pi_2 = \langle a_1, \ldots, a_k, a'_1, \ldots, a'_j \rangle$.

■

The state produced by applying π to a state s is the state that is produced by applying the actions of π in the order given. We will denote this by extending the state-transition function γ as follows:

$$\gamma(s, \pi) = \begin{cases} s & \text{if } k = 0 \text{ (i.e., } \pi \text{ is empty)} \\ \gamma(\gamma(s, a_1), \langle a_2, \ldots, a_k \rangle) & \text{if } k > 0 \text{ and } a_1 \text{ is applicable to } s \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 2.4 Let $\mathcal{P} = (\Sigma, s_0, g)$ be a planning problem. A plan π is a *solution* for $\mathcal{P}$ if $g \subseteq \gamma(s_0, \pi)$. A solution π is *redundant* if there is a proper subsequence of π that is also a solution for $\mathcal{P}$; π is *minimal* if no other solution plan for $\mathcal{P}$ contains fewer actions than π. ■

Note that a minimal solution cannot be redundant.

Example 2.2 In the planning domain described in Example 2.1, suppose the initial state is s_0 and $g = \{\text{onrobot}, \text{at2}\}$. Let:

$\pi_1 = \langle\text{move2}, \text{move2}\rangle$
$\pi_2 = \langle\text{take}, \text{move1}\rangle$
$\pi_3 = \langle\text{take}, \text{move1}, \text{put}, \text{move2}, \text{take}, \text{move1}, \text{load}, \text{move2}\rangle$
$\pi_4 = \langle\text{take}, \text{move1}, \text{load}, \text{move2}\rangle$
$\pi_5 = \langle\text{move1}, \text{take}, \text{load}, \text{move2}\rangle$

Then π_1 is not a solution because it is not applicable to s_0; π_2 is not a solution because although it is applicable to s_0, the resulting state is not a goal state; π_3 is a redundant solution; and π_4 and π_5 are the only minimal solutions. ■

2.2.2 State Reachability

If s is a state, then the set of all *successors* of s is

$$\Gamma(s) = \{\gamma(s, a) \mid a \in A \text{ and } a \text{ is applicable to } s\}.$$

If we let $\Gamma^2(s) = \Gamma(\Gamma(s)) = \bigcup\{\Gamma(s') \mid s' \in \Gamma(s)\}$, and similarly for $\Gamma^3, \ldots, \Gamma^n$, then the set of states *reachable* from s is the *transitive closure*:

$$\hat{\Gamma}(s) = \Gamma(s) \cup \Gamma^2(s) \cup \ldots.$$

An action a is said to be *relevant* for a goal g iff $g \cap \text{effects}^+(a) \neq \emptyset$ and $g \cap \text{effects}^-(a) = \emptyset$. Intuitively, these conditions state that a can contribute toward producing a state in $S_g = \{s \in S \mid g \subseteq s\}$. Let us define the *regression set* of a goal g, for an action a that is relevant for g, to be the minimal set of propositions required in a state s in order to apply a to s and to get g:

$$\gamma^{-1}(g, a) = (g - \text{effects}^+(a)) \cup \text{precond}(a).$$

Thus for any state $s, \gamma(s, a) \in S_g$ iff $\gamma^{-1}(g, a) \subseteq s$, i.e., s is a superset of the regression set.

The set of the regression sets over all actions relevant for a goal g is:

$$\Gamma^{-1}(g) = \{\gamma^{-1}(g, a) \mid a \in A \text{ is relevant for } g\}.$$

A goal state is reachable in a single step from a state s iff s is a superset of an element of $\Gamma^{-1}(g)$, i.e., there is $s' \in \Gamma^{-1}(g)$: $s' \subseteq s$. Similarly, the set of all regression sets of g in two steps is $\Gamma^{-2}(g) = \Gamma^{-1}(\Gamma^{-1}(g))$, etc., and the set of all regression sets of g, whose supersets are states from which g is *reachable*, is the transitive closure:

$$\hat{\Gamma}^{-1}(g) = \Gamma^{-1}(g) \cup \Gamma^{-2}(g) \cup \ldots.$$

Proposition 2.1 *A planning problem $\mathcal{P} = (\Sigma, s_0, g)$ has a solution iff $S_g \cap \hat{\Gamma}(s_0) \neq \emptyset$.*

Proposition 2.2 *A planning problem $\mathcal{P} = (\Sigma, s_0, g)$ has a solution iff s_0 is a superset of some element in $\hat{\Gamma}^{-1}(g)$.*

2.2.3 Stating a Planning Problem

For set-theoretic planning, we have defined the planning domain $\Sigma = (S, A, \gamma)$, which is independent of any particular goal or initial state, and the planning problem $\mathcal{P} = (\Sigma, s_0, g)$, which includes a domain, an initial state, and a goal.

We began this chapter by mentioning the need for a way to specify $\mathcal{P}$ without giving all of the members of S and γ explicitly. To accomplish that, we will use the *statement* of $\mathcal{P}$, defined to be the triple $P = (A, s_0, g)$. $\mathcal{P}$ and P can be regarded as semantic and syntactic specifications, respectively (for more about this, see Section 2.3.4).

One difficulty is that the statement of a planning problem is ambiguous because it does not specify the set of states S (see Example 2.3).

Example 2.3 Let $P = (\{\mathtt{a1}\}, s_0, \mathtt{g})$, where:

$$\begin{aligned} \mathtt{a1} &= (\{\mathtt{p1}\}, \{\mathtt{p1}\}, \{\mathtt{p2}\}) \\ s_0 &= \{\mathtt{p1}\} \\ g &= \{\mathtt{p2}\} \end{aligned}$$

Then P is the statement of the planning problem $\mathcal{P}$, in which:

$$\begin{aligned} L &= \{\mathtt{p1}, \mathtt{p2}\} \\ S &= \{\{\mathtt{p1}\}, \{\mathtt{p2}\}\} \\ \gamma(\{\mathtt{p1}\}, \mathtt{a1}) &= \{\mathtt{p2}\} \end{aligned}$$

However, P is also the statement of the planning problem $\mathcal{P}'$, in which:

$$\begin{aligned} L &= \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}\} \\ S &= \{\{\mathsf{p1}\}, \{\mathsf{p2}\}, \{\mathsf{p1}, \mathsf{p3}\}, \{\mathsf{p2}, \mathsf{p3}\}\} \\ \gamma(\{\mathsf{p1}\}, \mathsf{a1}) &= \{\mathsf{p2}\} \\ \gamma(\{\mathsf{p1}, \mathsf{p3}\}, \mathsf{a1}) &= \{\mathsf{p2}, \mathsf{p3}\} \end{aligned}$$

Note, however, that p3 plays no role in problem $\mathcal{P}'$: the set $\hat{\Gamma}^{-1}(g)$ of regression sets and the set $\hat{\Gamma}(s_0)$ of states reachable from s_0 are the same in both $\mathcal{P}$ and $\mathcal{P}'$:

$$\hat{\Gamma}^{-1}(g) = \{\{\mathsf{p1}\}, \{\mathsf{p2}\}\}; \quad \hat{\Gamma}(s_0) = \{\{\mathsf{p1}\}, \{\mathsf{p2}\}\}$$

■

The following proposition generalizes the final equation:

Proposition 2.3 *Let $\mathcal{P}$ and $\mathcal{P}'$ be two planning problems that have the same statement. Then both $\mathcal{P}$ and $\mathcal{P}'$ have the same set of reachable states $\hat{\Gamma}(s_0)$ and the same set of solutions.*

This proposition means that the statement of a planning problem is unambiguous enough to be acceptable as a specification of a planning problem.

2.2.4 Properties of the Set-Theoretic Representation

We now discuss some advantages and disadvantages of the set-theoretic representation scheme.

Readability. One advantage of the set-theoretic representation is that it provides a more concise and readable representation of the state-transition system than we would get by enumerating all of the states and transitions explicitly.

Example 2.4 Consider the DWR domain shown in Figure 2.1. The robot can be at either loc1 or loc2. The robot can be empty or can be holding any one of the containers, and likewise for the crane. The rest of the containers can be distributed in any order between pile1 and pile2. The total number of possible states is 2,580,480 (see Exercise 2.17). If we used a state-transition representation in which the states were named $s_1, s_2, \ldots, s_{2,580,480}$, it would be difficult to keep track of what each state meant.

In a set-theoretic representation, we can make each state's meaning more obvious by using propositions to represent the statuses of the containers, the

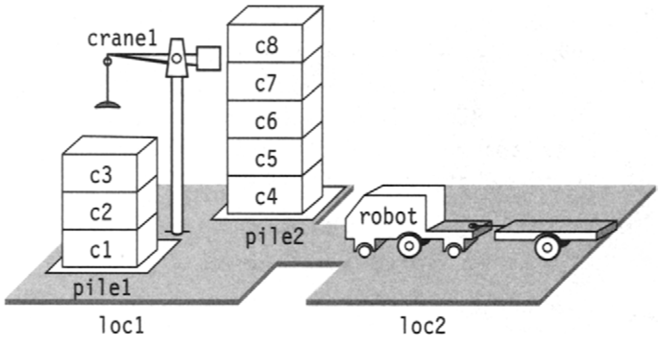

Figure 2.1 A DWR state for Example 2.4.

crane, and the robot. For example, the state shown in Figure 2.1 might be represented as:

{nothing-on-c3, c3-on-c2, c2-on-c1, c1-on-pile1,
nothing-on-c8, c8-on-c7, c7-on-c6, c6-on-c5,
c5-on-c4, c4-on-pile2, robot-at-loc2, crane-empty}

Furthermore, the set-theoretic representation lets us avoid specifying all 2,580,480 of the states (and their associated state transitions) explicitly because we can generate them as needed using techniques such as the ones described next. ■

Computation. A proposition in a state s is assumed to *persist* in $\gamma(s, a)$ unless explicitly mentioned in the effects of a. These effects are defined with two subsets: $\text{effects}^+(a)$, the propositions to *add* to s, and $\text{effects}^-(a)$, the propositions to *remove* from s in order to get the new state $\gamma(s, a)$. Hence, the transition function γ and the applicability conditions of actions rely on very easily computable *set operations*: if $\text{precond}(a) \subseteq s$, then $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$.

Expressivity. A significant problem is that not every state-transition system Σ has a set-theoretic representation. The work-around is to construct a state-transition system Σ' that is equivalent to Σ but does have a set-theoretic representation (see Example 2.5).

Example 2.5 Suppose a computer has an n-bit register r and a single operator incr that assigns $r \leftarrow r + 1 \bmod m$, where $m = 2^n$. Let $L = \{\text{val}_0, \ldots, \text{val}_{m-1}\}$, where each val_i is the proposition "the register r contains the value i." Then we can represent the computer as a state-transition system $\Sigma = (S, A, \gamma)$, where each state is a member of L, $A = \{\text{incr}\}$, and $\gamma(\text{val}_i, \text{incr}) = \text{val}_{i+1} \bmod m$ for each i. Now, suppose that r contains some value c in the initial state and that the goal is for r

to contain a prime number. Then the classical planning problem is $\mathcal{P} = (\Sigma, s_0, g)$, where $s_0 = \mathsf{val}_c$ and $S_g = \{\mathsf{val}_i \mid i \text{ is prime}\}$.

If we use the above definitions for Σ and $\mathcal{P}$, then there is no set-theoretic action that represents incr, nor is there any set of propositions $g \subseteq 2^L$ that represents the set of goal states S_g. We can circumvent this difficulty by defining a state-transition system Σ' and planning problem $\mathcal{P}'$ as follows:

$$
\begin{aligned}
L' &= L \cup \{\mathsf{prime}\} \\
S' &= 2^{L'} \\
A' &= \{\mathsf{incr}_0, \ldots, \mathsf{incr}_{m-1}\} \\
\gamma'(\mathsf{val}_i, \mathsf{incr}_i) &= \begin{cases} \{\mathsf{prime}, \mathsf{val}_{i+1} \bmod m\} & \text{if } i+1 \bmod m \text{ is prime} \\ \{\mathsf{val}_{i+1} \bmod m\} & \text{otherwise} \end{cases} \\
\Sigma' &= (S', A', \gamma') \\
S'_g &= \{s \subset 2^{L'} \mid \mathsf{prime} \in s\} \\
\mathcal{P}' &= (\Sigma', s_0, S_{g'})
\end{aligned}
$$

$\mathcal{P}'$ has the following set-theoretic representation:

$$
\begin{aligned}
g' &= \{\mathsf{prime}\}; \\
\mathsf{precond}(\mathsf{incr}_i) &= \{\mathsf{val}_i\}, \quad i = 0, \ldots, n \\
\mathsf{effects}^-(\mathsf{incr}_i) &= \begin{cases} \{\mathsf{val}_i, \neg\,\mathsf{prime}\} & \text{if } i \text{ is prime} \\ \{\mathsf{val}_i\} & \text{otherwise} \end{cases} \\
\mathsf{effects}^+(\mathsf{incr}_i) &= \begin{cases} \{\mathsf{val}_{i+1} \bmod m, \mathsf{prime}\} & \text{if } i+1 \bmod m \text{ is prime} \\ \{\mathsf{val}_{i+1} \bmod m\} & \text{otherwise} \end{cases}
\end{aligned}
$$

However, creating this set-theoretic representation incurs a large computational cost. There are 2^n different actions, and in order to write all of them, we must compute all of the prime numbers from 1 to 2^n.

■

Another expressivity issue with this representation scheme is that not every set of propositions in L corresponds to a meaningful state of a domain we are trying to represent.

Example 2.6 Consider the domain of Example 2.1 (see page 21). The state {holding} is ambiguous in our intended interpretation of the domain because it does not contain a proposition that gives the robot location. Furthermore, the state {holding, at2, onrobot} is not consistent with our intended interpretation because it says that the container is both on the crane and on the robot. ■

If the actions of a planning problem and s_0 are written correctly, then an ambiguous or inconsistent state s will never appear in any solution to the planning problem: either s will not be reachable from s_0 (i.e., $s \notin \hat{\Gamma}(s_0)$) or g will not be reachable from s (i.e., s is not a superset of an element of $\Gamma^{-1}(g)$). However, this puts all the burden of a good definition of a domain on the specification of the set of actions.

2.3 Classical Representation

The classical representation scheme generalizes the set-theoretic representation scheme using notation derived from first-order logic.[2] States are represented as sets of logical atoms that are true or false within some interpretation. Actions are represented by *planning operators* that change the truth values of these atoms.

2.3.1 States

To develop a language for classical planning, we will start with a first-order language $\mathcal{L}$ in which there are finitely many predicate symbols and constant symbols and *no* function symbols; thus every term of $\mathcal{L}$ is either a variable symbol or a constant symbol. We will augment $\mathcal{L}$ to include some additional symbols and expressions. For the predicate and constant symbols of $\mathcal{L}$, we will use alphanumeric strings that are at least two characters long (e.g., crane1 or r2) with a sans-serif font. For the variable symbols, we will use single characters, possibly with subscripts (e.g., x or y_{13}).

A *state* is a set of ground atoms of $\mathcal{L}$. Since $\mathcal{L}$ has no function symbols, the set S of all possible states is guaranteed to be finite. As in the set-theoretic representation scheme, an atom p holds in s iff $p \in s$. If g is a set of literals (i.e., atoms and negated atoms), we will say that s *satisfies* g (denoted $s \models g$) when there is a substitution σ such that every positive literal of $\sigma(g)$ is in s and no negated literal of $\sigma(g)$ is in s.

Example 2.7 Suppose we want to formulate a DWR planning domain in which there are two locations (loc1, loc2), one robot (r1), one crane (crane1), two piles (p1, p2), and three containers (c1, c2, c3). The set of constant symbols is {loc1, loc2, r1, crane1, p1, p2, c1, c2, c3, pallet}. Recall that pallet is a symbol that denotes the object that sits at the bottom of a pile; hence if a pile p3 is empty, then top(pallet,p3). One of the states is the state s_1 shown in Figure 2.2. ■

2. See Appendix B for an introduction and first-order logic.

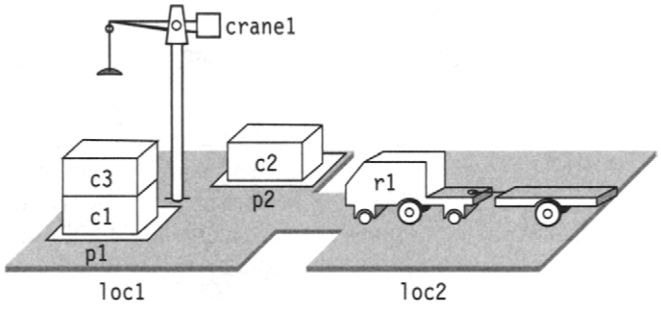

Figure 2.2 The DWR state s_1 = {attached(p1,loc1), attached(p2,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

Note that the truth value of an atom may vary from state to state. For example, at(r1,loc2) holds in the state depicted in Figure 2.2, but it does not hold in a state where robot r1 moves to location loc1. Hence, the predicate at can be considered as a function of the set of states; it will be called a *fluent* or *flexible relation*. However, not every predicate symbol in $\mathcal{L}$ is necessarily fluent. For example, the truth value of adjacent(loc1,loc2) is not intended to vary from state to state for a given DWR domain. A state-invariant predicate such as adjacent is called a *rigid relation*.

Note also that although $\mathcal{L}$ is a first-order language, a state is not a set of first-order formulas—it is just a set of ground atoms. Both here and in the set-theoretic representation scheme, we use the *closed-world assumption*: an atom that is not explicitly specified in a state does not hold in that state.

2.3.2 Operators and Actions

The transition function γ is specified generically through a set of planning operators that are instantiated into actions.

Definition 2.5 In classical planning, a *planning operator* is a triple o = (name(o), precond(o), effects(o)) whose elements are as follows:

- name(o), the *name* of the operator, is a syntactic expression of the form $n(x_1, \ldots, x_k)$, where n is a symbol called an *operator symbol*, $x_1, \ldots, x_k$ are all of the variable symbols that appear anywhere in o, and n is unique (i.e., no two operators in $\mathcal{L}$ have the same operator symbol).
- precond(o) and effects(o), the *preconditions* and *effects* of o, respectively, are generalizations of the preconditions and effects of a set-theoretic

action: instead of being sets of propositions, they are sets of literals (i.e., atoms and negations of atoms). ■

Rigid relations cannot appear in the effects of any operator o because they are invariant over all the states; they can be used only in precond(o). In other words, any predicate in effects(o) is a flexible relation.

Rather than writing $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$ to define a planning operator, we will usually write operator definitions as shown in Example 2.8.

Example 2.8 Here are the planning operators for the DWR domain.

```
move(r, l, m)
    ;; robot r moves from location l to an adjacent location m
    precond:  adjacent(l, m), at(r, l), ¬ occupied(m)
    effects:  at(r, m), occupied(m), ¬ occupied(l), ¬ at(r, l)

load(k, l, c, r)
    ;; crane k at location l loads container c onto robot r
    precond:  belong(k, l), holding(k, c), at(r, l), unloaded(r)
    effects:  empty(k), ¬ holding(k, c), loaded(r, c), ¬ unloaded(r)

unload(k, l, c, r)
    ;; crane k at location l takes container c from robot r
    precond:  belong(k, l), at(r, l), loaded(r, c), empty(k)
    effects:  ¬ empty(k), holding(k, c), unloaded(r), ¬ loaded(r, c)

put(k, l, c, d, p)
    ;; crane k at location l puts c onto d in pile p
    precond:  belong(k, l), attached(p, l), holding(k, c), top(d, p)
    effects:  ¬ holding(k, c), empty(k), in(c, p), top(c, p), on(c, d), ¬ top(d, p)

take(k, l, c, d, p)
    ;; crane k at location l takes c off of d in pile p
    precond:  belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
    effects:  holding(k, c), ¬ empty(k), ¬ in(c, p), ¬ top(c, p), ¬ on(c, d),
                top(d, p)
```

■

The purpose of an operator's name is to provide an unambiguous way to refer to the operator or to substitution instances of the operator without having to write their preconditions and effects explicitly. If o is an operator or an *operator instance* (i.e., a substitution instance of an operator), then name(o) refers unambiguously to o. Thus, when it is clear from the context, we will write name(o) to refer to the entire operator o.

Example 2.9 take(k, l, c, d, p) is the last of the operators in Example 2.8, and take(crane1,loc1,c1,c2,p1) is the following operator instance:

```
take(crane1,loc1,c1,c2,p1)
   ;; crane crane1 at location loc1 takes c1 off of c2 in pile p1
   precond:  belong(crane1,loc1), attached(p1,loc1),
             empty(crane1), top(c1,p1), on(c1,c2)
   effects:  holding(crane1,c1), ¬empty(crane1), ¬in(c1,p1),
             ¬top(c1,p1), ¬on(c1,c2), top(c2,p1)
```

■

Definition 2.6 For any set of literals L, L^+ is the set of all atoms in L, and L^- is the set of all atoms whose negations are in L. In particular, if o is an operator or operator instance, then precond$^+(o)$ and precond$^-(o)$ are o's positive and negative preconditions, respectively, and effects$^+(o)$ and effects$^-(o)$ are o's positive and negative effects, respectively. ■

Definition 2.7 An *action* is any ground instance of a planning operator. If a is an action and s is a state such that precond$^+(a) \subseteq s$ and precond$^-(a) \cap s = \emptyset$, then a is *applicable* to s, and the result of applying a to s is the state:

$$\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a).$$

■

Thus, like in set-theoretic planning, state transitions can easily be computed using set operations.

Example 2.10 take(crane1,loc1,c3,c1,p1) is applicable to the state s_1 of Figure 2.2. The result is the state $s_5 = \gamma(s_1, \text{take(crane1, loc1, c1, c2, p1)})$ shown in Figure 2.3. ■

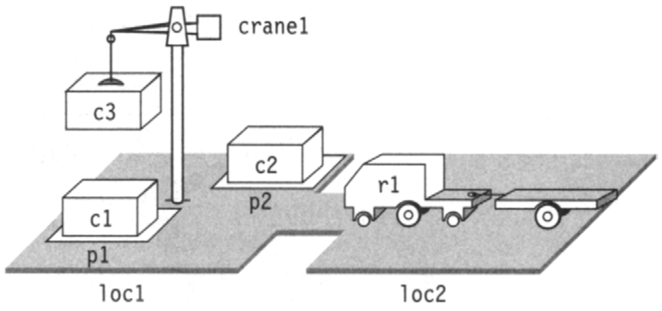

Figure 2.3 The DWR state s_5 = {attached(p1,loc1), in(c1,p1), top(c1,p1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), holding(crane1,c3), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

2.3.3 Plans, Problems, and Solutions

Here are the definitions of classical planning domains, problems, plans, and solutions.

Definition 2.8 Let $\mathcal{L}$ be a first-order language that has finitely many predicate symbols and constant symbols. A *classical planning domain*[3] in $\mathcal{L}$ is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^{\{\text{all ground atoms of } \mathcal{L}\}}$;
- $A = \{$all ground instances of operators in $O\}$, where O is a set of operators as defined earlier;
- $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if $a \in A$ is applicable to $s \in S$, and otherwise $\gamma(s, a)$ is undefined; and
- S is closed under γ, i.e., if $s \in S$, then for every action a that is applicable to s, $\gamma(s, a) \in S$.

■

Definition 2.9 A *classical planning problem* is a triple $\mathcal{P} = (\Sigma, s_0, g)$, where:

- s_0, the initial state, is any state in set S;
- g, the goal, is any set of ground literals; and
- $S_g = \{s \in S \mid s \text{ satisfies } g\}$.

The *statement* of a planning problem $\mathcal{P} = (\Sigma, s_0, g)$ is $P = (O, s_0, g)$.

■

The statement of a planning problem is the syntactic specification that we would use, e.g., to describe $\mathcal{P}$ to a computer program. Its properties are like those for set-theoretic planning (Section 2.2.3): More than one classical planning problem may have the same statement, but if two classical problems have the same statement, then they will have the same set of reachable states and the same set of solutions.

Since g contains negated atoms, the definition of γ^{-1} differs from the one in Definition 2.1 (see page 20). An action a is *relevant* for g, i.e., a can produce a state that satisfies g, if:

- $g \cap \text{effects}(a) \neq \emptyset$, i.e., a's effects contribute to g; and
- $g^+ \cap \text{effects}^-(a) = \emptyset$ and $g^- \cap \text{effects}^+(a) = \emptyset$, i.e., a's effects do not conflict with g.

3. Our definition of a planning domain is more specific than the usual informal use of that expression. A domain is relative to a language $\mathcal{L}$; e.g., in the DWR example we would know from $\mathcal{L}$ all the constants, locations, piles, robots, cranes, and containers there are in the domain. The informal use is to say that a domain is a set of operators, i.e., that a domain includes all planning problems whose statements can be expressed with O and some initial state and goal.

If a is relevant for g, then:

$$\gamma^{-1}(g, a) = (g - \text{effects}(a)) \cup \text{precond}(a)$$

The set of all regression sets over all actions relevant for a goal g is as before:

$$\Gamma^{-1}(g) = \{\gamma^{-1}(g, a) \mid a \text{ is an operator instance relevant for } g\}$$

$\hat{\Gamma}(s_0)$ and $\hat{\Gamma}^{-1}(g)$ have the same properties as earlier.

The definitions of a plan and the result of applying it to a state are the same as they were in set-theoretic planning. A plan π is a solution for $\mathcal{P}$ if $\gamma(s_0, \pi)$ satisfies g. As before, a solution is redundant if a proper subsequence of it is a solution, and its length is shortest or minimal if no solution contains fewer actions.

Example 2.11 Consider the following plan:

$$\begin{aligned}\pi_1 = \langle &\text{take(crane1, loc1, c3, c1, p1)},\\ &\text{move(r1, loc2, loc1)},\\ &\text{load(crane1, loc1, c3, r1)}\rangle\end{aligned}$$

This plan is applicable to the state s_1 shown in Figure 2.2, producing the state s_6 shown in Figure 2.4. π_1 is a minimal solution to the DWR problem $\mathcal{P}_1$ whose initial state is s_1 and whose goal formula is:

$$g_1 = \{\text{loaded(r1, c3), at(r1, l2)}\}$$

π_1 is also a solution to the DWR problem $\mathcal{P}_2$ whose initial state is s_1 and whose goal formula is $g_2 = \{\text{at(r1, loc1)}\}$, but in this case π_1 is redundant. ■

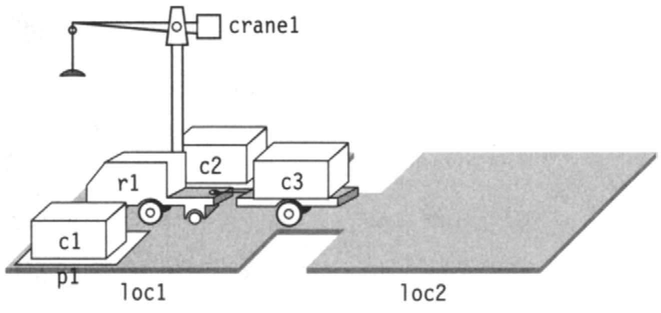

Figure 2.4 The state s_6 = {attached(p1,loc1), in(c1,p1), top(c1,p1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc1), occupied(loc1), loaded(r1,c3)}.

2.3.4 Semantics of Classical Representations

So far, we have deliberately blurred the distinction between the syntactic specification of a classical planning problem and what it means. The distinction is analogous to the distinction between a logical theory and its models (see Appendix B). Intuitively, the semantics is given by Σ and the syntax is specified by P.

Suppose $P = (O, s_0, g)$ is the statement of a classical planning problem $\mathcal{P}$. Let Σ be the restricted state-transition system for $\mathcal{P}$, and I is a function, called an *interpretation*, such that (see Figure 2.5):

- For every state t of P, $I(t)$ is a state of Σ.
- For every operator instance o of P, $I(o)$ is an action of Σ.

To specify the semantics of the representation, we define the pair (Σ, I) to be a *model* of P if for every state t of P and every operator instance o of P:

$$\gamma(I(s), I(o)) = I((s - \text{effects}^-(o)) \cup \text{effects}^+(o))$$

This distinction between P and Σ is needed in order to establish the soundness property[4] of a representation scheme.

In classical planning, we have used a first-order language $\mathcal{L}$ but have used none of the inferential capabilities of first-order logic. As a consequence, the question of whether Σ is a model of P is quite trivial. However, there are cases where we might like to extend the classical planning formalism to allow states, goals, and operators to contain nonatomic first-order formulas. With such an extension, the semantic considerations are more complicated (see Section 2.4.5).

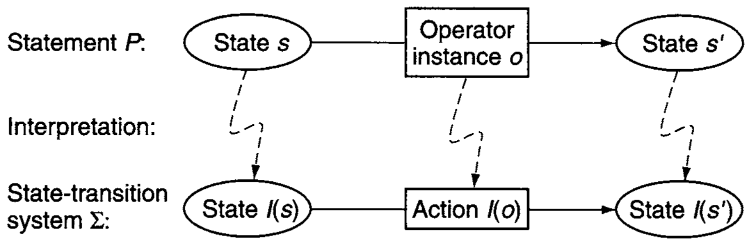

Figure 2.5 Interpretation of a planning domain as a state-transition system.

4. See Appendix B.

2.4 Extending the Classical Representation

Since the classical planning formalism is very restricted, extensions to it are needed in order to describe interesting domains. It is possible to extend the classical planning formalism, while staying within the same model of restricted state-transition systems, by allowing expressions that are more general than just collections of literals in the preconditions and effects of the operator, as well as in the descriptions of states. The following subsections discuss several examples.

2.4.1 Simple Syntactical Extensions

One of the simplest extensions is to allow the goal formula g to be any existentially closed conjunct of literals, such as $\exists x\ \exists y(\mathsf{on}(x, \mathsf{c1}) \wedge \mathsf{on}(y, \mathsf{c2}))$. This brings no increase in expressive power, for if g is an existentially closed conjunct, it can be replaced by a single ground atom p and a planning operator whose precondition is g with the quantifiers removed and whose effect is p. However, in some cases the extension may make the planning representation look more natural for human readers.

Another simple extension is to allow the use of *typed variables and relations.* The statement of a problem can involve not only the specification of O, s_0, and g but also the definition of sets of constants. For example, in the DWR domain, the set of constants is partitioned into classes such as locations, robots, cranes, piles, and containers.

A typed variable ranges over a particular set, e.g., l is in locations, r in robots. A type is a unary predicate whose truth value does not change from state to state. Figure 2.6 illustrates how to specify part of the DWR domain using typed variables; the notation is that of the PDDL planning language [387].

Like the previous extension, this one gives no increase in expressive power. If we want to say that a variable l must be of type location, we can do this within ordinary classical planning by introducing location as a new predicate symbol and including location(l) in the precondition of any operator that uses l. However, this extension can reduce the programming effort needed to specify a classical planning domain: if a planning system enforces the type restrictions, then the domain author has a way to ensure that each variable has the correct range of values. Furthermore, the extension may improve the efficiency of such a planning system by reducing the number of ground instances it needs to create.

2.4.2 Conditional Planning Operators

There are many situations in which we might want the effects of an action to depend on the input situation. One of the simplest examples would be to model the effects of a push button that turns a lamp off if the lamp is on and vice versa. One way to do this is to extend the classical operator formalism to include conditional operators as described below.

```
(define (domain dwr)
   (:requirements :strips :typing)
   (:types location pile robot crane container)
   (:predicates
      (adjacent ?l1 ?l2 - location) (attached ?p - pile ?l - location)
      (belong ?k - crane ?l - location)
      (at ?r - robot ?l - location) (occupied ?l - location)
      (loaded ?r - robot ?c - container) (unloaded ?r - robot)
      (holding ?k - crane ?c - container) (empty ?k - crane)
      (in ?c - container ?p - pile) (top ?c - container ?p - pile)
      (on ?k1 - container ?k2 - container))
   (:action move
      :parameters (?r - robot ?from ?to - location)
      :precondition (and (adjacent ?from ?to)
            (at ?r ?from) (not (occupied ?to)))
      :effect (and (at ?r ?to) (not (occupied ?from))
            (occupied ?to) (not (at ?r ?from))))
   (:action load
      :parameters (?k - crane ?c - container ?r - robot)
      :vars (?l - location)
      :precondition (and (at ?r ?l) (belong ?k ?l)
            (holding ?k ?c) (unloaded ?r))
      :effect (and (loaded ?r ?c) (not (unloaded ?r))
            (empty ?k) (not (holding ?k ?c))))
```

Figure 2.6 A partial PDDL specification of the DWR domain. adjacent is declared to be a binary predicate symbol whose arguments both are of type location, attached is declared to be a binary predicate symbol whose first arguments have type pile and location, and so forth.

A *conditional operator* o is an $(n+1)$-tuple,

$$n, t_1, t_2, \ldots, t_n,$$

where n is a syntactic expression $o(x_1, \ldots, x_k)$ as in the usual classical operator definition, and each t_i is a pair $(\text{precond}_i, \text{effects}_i)$ such that precond_i is a set of preconditions and each effects_i is a set of effects.

Let a be a ground instance of a conditional operator, s be a state, and $I = \{i : s \text{ satisfies } \text{precond}_i(a)\}$. If $I \neq \emptyset$, then we say that o is applicable to the state s and that the result is the state:

$$\gamma(s,a) = (s - \bigcup_{i \in I} \text{effects}_i^{+}(a)) \cup \bigcup_{i \in I} \text{effects}_i^{-}(a)$$

Instead of a set of pairs $(\text{precond}_i, \text{effects}_i)$, an operator with conditional effects is usually written in an equivalent but more compact expression, as illustrated in Example 2.12. The operator has the usual unconditional preconditions and effects followed by the conditional effects and their qualifying conditions. Syntactical facilities such as nested conditionals are also allowed by many planning languages.

Example 2.12 In Example 2.8 (see page 29), the location of a container loaded on a robot or held by a crane is not explicit. Suppose we need to keep this location explicit, e.g., with the at predicate. Since the location changes for a container loaded on a robot with a move operator, we can respecify a new move operator with a condition effect as follows:

```
move(r, l, m, c)
    ;; robot r with possibly a container c moves from location l to location m
    precond: adjacent(l, m), at(r, l), ¬ occupied(m)
    effects: at(r, m), occupied(m), ¬ occupied(l), ¬ at(r, l),
             (when loaded(r, c) then at(c, m), ¬ at(c, l))
```

In the last line, the construct *when* is followed by an additional precondition; when loaded(r, c) is true in state s, then the additional effects at(c, m) and $\neg$ at(c, l) take place in state $\gamma(s, a)$. This operator is equivalent to two pairs $(\text{precond}_i, \text{effects}_i)$ corresponding to the two cases of a loaded and an unloaded robot. ■

2.4.3 Quantified Expressions

Suppose we have robots that can hold several containers, and suppose we want to keep track of the location of all containers, as discussed in Example 2.12. One way to express this is to extend the operator representation so that effects(o) is not just a set of literals but a logical formula in which some of the literals represent additional preconditions and other literals represent effects that will occur if those preconditions are satisfied. To represent this, we might use a notation such as the following:

```
move(r, l, m, x)
    ;; robot r moves from location l to m with possibly several containers
    precond: at(r, l)
    effects: ¬ at(r, l), at(r, m),
             (forall x when loaded(r, x) then at(x, m), ¬ at(x, l))
```

In the final line of the operator, the literal loaded(r, x) is an additional precondition, and the literals at(x, m) and $\neg$ at(x, l) are effects that will occur for *every* container x such that the precondition is satisfied. For example, suppose we apply the action $a =$ move(r1, l1, l2) to this state:

$$s = \{\text{at(r1, l1), at(c1, l1), at(c2, l1), loaded(r1, c1), loaded(r1, c2)}\}$$

In this case, the positive and negative effects of a will be:

$$\text{effects}^+(a) = \{\mathsf{at(r1,l2), at(c1,l2), at(c2,l2)}\}$$
$$\text{effects}^-(a) = \{\neg\,\mathsf{at(r1,l1)}, \neg\,\mathsf{at(c1,l1)}, \neg\,\mathsf{at(c2,l1)}\}$$

These effects will produce this state:

$$\gamma(s,a) = \{\mathsf{at(r1,l2), at(c1,l2), at(c2,l2), loaded(r1,c1), loaded(r1,c2)}\}$$

Because of our assumption of finite domains, the *forall* quantifier can be expanded into a finite expression. In the case of the previous move operator, this expression is a finite conjunction of conditional effects for all the containers of the problem.

Quantified expressions can also be convenient for specifying concisely the initial state or the goal of a planning problem, e.g., for saying that all the robots are initially unloaded or that the goal is to have all containers stacked into a destination pile. Typed variables and rigid predicates for testing the types are very useful for handling efficiently quantified expressions.

2.4.4 Disjunctive Preconditions

Continuing the previous example, suppose that we can move the robot cart from location l to location m under either of two conditions: either there must be a road from l to m, or the cart must have all-wheel drive. One way to represent this is to extend the operator representation so that precond(o) may be any logical formula composed of conjuncts and disjuncts of literals. Then we could create an operator such as the following:

```
move(r,l,m)
    ;; move robot cart r from location l to location m
    precond: at(r, l)
             ∧(road(l, m) ∨ all-wheel-drive(r))
    effects: ¬ at(r, l), at(r, m),
             (forall x when loaded(r, x) then at(x, m), ¬ at(x, l))
```

Thus, the action move(r1,l1,l2) would be applicable to the state

$$\{\mathsf{road(l1,l2), at(r1,l1), at(c1,l1), at(c1,l1), loaded(c1,r1), loaded(c2,r1)}\}$$

and to the state

$$\{\mathsf{all\text{-}wheel\text{-}drive(r1), at(r1,l1), at(c1,l1), at(c1,l1), loaded(c1,r1), loaded(c2,r1)}\}$$

but would not be applicable to the state

$$\{\mathsf{at(r1,l1), at(c1,l1), at(c1,l1), loaded(c1,r1), loaded(c2,r1)}\}$$

2.4.5 Axiomatic Inference

Sometimes we may want to infer conditions that are not stated explicitly in a state of the world. For example, it could be useful in the DWR domain to have two formulas saying that the adjacent property between locations is symmetric and that a crane is empty iff it is not holding anything:

$$\forall l\, \forall l'\ (\mathsf{adjacent}(l, l') \leftrightarrow \mathsf{adjacent}(l', l)) \tag{2.1}$$

$$\forall k\ (\forall x\, \neg\, \mathsf{holding}(k, x)) \leftrightarrow \mathsf{empty}(k) \tag{2.2}$$

Furthermore, we might also like to say that in the initial state s_0 of a particular DWR problem, all containers are in location loc1, i.e.:

$$\forall c\ (\mathsf{container}(c) \rightarrow \mathsf{in}(c, \mathsf{loc1})) \tag{2.3}$$

Formula 2.1 can easily be introduced into the classical planning formalism because adjacent is a predicate whose truth value does not change from state to state: it is a rigid relation. Recall that rigid relations do not appear in the *effects* of any operator because their truth values do not change; they may appear only in preconditions. Hence, whatever is deduced with a formula like Formula 2.1 remains true everywhere. However, special care needs to be taken with Formula 2.2 because holding and empty both are flexible relations.

To handle Formula 2.2 properly, we need to remove empty from the effects of all operators and to rewrite Formula 2.2 as two implications that give the truth value of empty from that of holding, for all cranes and containers being held:

$$\forall k\ (\neg\exists x\ \mathsf{holding}(k, x) \rightarrow \mathsf{empty}(k))$$
$$\forall k\ (\exists x\ \mathsf{holding}(k, x) \rightarrow \neg\mathsf{empty}(k))$$

This technique establishes a further distinction between two classes of flexible relations.

1. *Primary relations*, such as holding, can appear in the effects and preconditions of operators.
2. *Secondary relations*, such as empty, cannot appear in effects but only in preconditions; secondary relations are deduced from primary and secondary relations.

In a way, secondary relations are a "syntactical sugar" that simplifies the specification of operators. States and state transitions are entirely defined by primary relations.

What about Formula 2.3, which is intended to be true in just one particular state? If we rely solely on the machinery that we introduced so far for representing and computing state transitions, i.e., difference and union of sets, then clearly Formula 2.3 cannot be handled in a general way. We have to restrict the language so that the only nonatomic formulas allowed must be true everywhere. However, because we are restricting the language to finite domains, formulas such as Formula 2.3 can be dealt with using an *expansion* of the formula into its corresponding ground atoms. This technique is also applied to quantified formulas in effects of operators (see Section 2.4.3).

Although some of the original approaches to AI planning were based on axiomatic inference [249], many classical planners do not use it. This is mainly in order to avoid complications such as the ones described above. As a general rule, it is much easier to use axiomatic inference in planners that keep track of the current state, which is something that most classical planners do not do. Axiomatic inference plays an important role in several successful nonclassical planners (see Chapters 10 and 11) that keep track of the current state.

2.4.6 Function Symbols

The planning language $\mathcal{L}$ can be extended to include function symbols. However, function symbols must be used with care because their presence can make planning problems undecidable (see Chapter 3). Also, some classical planning procedures work by creating all possible ground instances of every atom, which is not possible with function symbols because there may be infinitely many ground atoms.

2.4.7 Attached Procedures

Attached procedures can sometimes be used to evaluate predicates and function symbols. For example, suppose we want to formulate a DWR domain such that each container c has a numeric weight $\mathsf{weight}(c)$, and we can load c onto a robot r only if the container's weight is less than the robot's weight limit $\mathsf{maxweight}(r)$. Then for the load operator, we might have a constraint $\mathsf{less}(\mathsf{weight}(c),\mathsf{maxweight}(r))$, where less is a predicate evaluated by a procedure that returns true if $\mathsf{weight}(c) < \mathsf{maxweight}(r))$ and false otherwise.

The ability to perform computations with attached procedures can be crucial in real-world planning problems. However, attached procedures require other computational means than set difference and union for computing $\gamma(s, a)$. An attached procedure usually cannot be called unless all of its variables have been bound—a requirement that causes great difficulty for many classical planning procedures.

2.4.8 Extended Goals

Recall from Chapter 1 the notion of *extended goals*, which are requirements not only on the final state of the problem but also on the states traversed to get from the initial state to the final state. Part V of this book describes some formalisms that explicitly incorporate extended goals, but these are not classical planning formalisms.

Extended goals are not allowed in classical planning problems *per se*, but certain kinds of planning problems with extended goals can be mapped into equivalent classical planning problems.[5] Here are several examples.

- Consider a DWR domain in which there is some location bad-loc to which we never want our robots to move. We can express this by changing the move operator to include an additional precondition $\neg$ at(r, bad-loc).
- Consider a DWR problem in which we want r1 to go first to loc1 and later to loc2. We can express this by adding an additional effect visited(m) to the move operator and putting both visited(loc1) and at(loc2) into the problem's goal formula.
- Suppose we want to require every solution to contain five actions or fewer. Like the above problem, this one requires us to extend classical planning to include function symbols. Then we can put an atom count(f(f(f(f(f(0)))))) in the initial state, add a precondition count(f(y)) to every action, and add effects $\neg$count(f(y)) and count(y) to every action.
- Consider a DWR problem in which we want r1 to visit loc1 twice. This problem cannot be translated into a classical planning problem that has the same set of solutions as the original problem. We could keep track of whether we have visited loc1 twice by introducing two different move operators move$_1$ and move$_2$ that have slightly different preconditions and effects—but if we do this, then we no longer have the same set of solutions as in the original planning problem. We can ensure that r1 visits loc1 twice if we extend the classical formalism to include the predicate symbols, function symbols, and attached procedures needed to perform integer arithmetic. If we do this, then we can put into the initial state a set of atoms consisting of visited(l,0) for every location l. Next, we can modify the move(r, l, m) operator to have a precondition visited(m, i) and effects $\neg$visited(m, i) and visited($m, i + 1$). Finally, we can add an additional goal visited(loc1, 2).
- Suppose we want to require that the number of times r1 visits loc1 must be at least three times the number of times it visits loc2. This again requires us to extend classical planning by introducing integer arithmetic. In this case, we can modify the planning problem as follows. First, put atoms into the initial state of the form visited(r, l, 0) for every robot r and every location l. Next, modify the operator move(r, l, m) to have an additional precondition visited(r, m, i) and additional effects $\neg$visited(r, m, i) and visited($r, m, i + 1$). Finally, modify the goal to include the atoms visited(r1,loc1,i), visited(r1,loc2,j), and $i \geq 3j$.

In several of these examples, it was necessary to extend classical planning to accommodate function symbols and integer arithmetic. As we discussed in

5. Unlike the other subsections, this one does not discuss a way to extend the classical representation scheme. Instead, it discusses how to map certain kinds of nonclassical planning problems into classical planning problems.

Sections 2.4.6 and 2.4.7, not all classical planning algorithms can be extended to accommodate these things. Furthermore, even with these extensions, certain kinds of extended goals cannot be accommodated (e.g., see Example 10.6).

2.5 State-Variable Representation

The state-variable representation is a third representation for classical planning systems, equivalent in expressiveness to the two previous ones, set-theoretic representation and classical representation. The main motivation here is to rely on *functions* instead of flexible *relations.*

2.5.1 State Variables

For example, consider the atom at(r1, loc1) in the classical representation of the DWR domain: it represents an element of a relation between a robot r1 and a location loc1. The truth value of this relation varies from one state to another. However, because r1 cannot be at two places at once and it is necessarily somewhere, then, for any given state s, there is *one and only one* location l such that at(r1, l) holds in s. In our intended model of this domain, this relation is a *function* that maps the set of states into the set of locations.

It can be advantageous to represent this relation using an explicit functional notation, e.g., $\mathrm{rloc}_{\mathrm{r1}} : S \rightarrow \mathrm{locations}$, such that the value of $\mathrm{rloc}_{\mathrm{r1}}(s)$ gives the unique location of robot r1 in state s. Here, the symbol $\mathrm{rloc}_{\mathrm{r1}}$ is a *state-variable symbol* that denotes a *state-variable function* whose values are characteristic attributes of the current state.

Furthermore, if there are several robots in a domain, it is convenient to rely on a more general notation in order to refer to the location of any robot r. Hence, instead of defining one state-variable $\mathrm{rloc}_{\mathrm{r1}}$ for each robot in the domain, we will use a function from the set of robots and the set of states into the set of locations, e.g., $\mathrm{rloc}(\mathrm{r1}, s) = \mathrm{loc1}$.

Let D be the finite set of all constant symbols in a planning domain. For example, in the DWR domain, D might include containers, robots, locations, cranes, and so forth. It is convenient to partition D into various *classes* of constants, such as the classes of robots, locations, cranes, and containers. We will represent each constant by an *object symbol* such as r1, loc2, or crane1.

In order to write unground expressions (see upcoming discussions), we will use *object variables.* These are variable symbols that range over sets of constants.[6] Each object variable v will have a range D^v that is the union of one or more classes. A term is either a constant or an object variable.

6. These are the usual variables in a first-order language, qualified here as object variables to distinguish them from state variables.

A k-ary *state variable* is an expression of the form $x(v_1, \ldots, v_k)$, where x is a state-variable symbol, and each v_i is either an object symbol or an object variable. A state variable denotes an element of a state-variable function,

$$x : D_1^x \times \ldots \times D_k^x \times S \rightarrow D_{k+1}^x,$$

where each $D_i^x \subseteq D$ is the union of one or more classes.

Example 2.13 Consider a simplified DWR domain in which there are no piles and no cranes; robots can load and unload containers autonomously. As we said earlier, we can use rloc(r1, s) to designate the current location of the robot r1. In a state s we might have rloc(r1, s) = loc1; in another state s' we might have rloc(r1, s') = loc2; hence rloc: robots $\times S \rightarrow$ locations.

Similarly, cpos(c1, s) could designate the current position of the container c1, which is either a location or a robot if c1 is on a robot. Thus in a state s we might have cpos(c1, s) = loc1, and in another state we might have cpos(c1, s') = r1. Hence cpos: containers $\times S \rightarrow$ locations $\cup$ robots.

Finally, rload: robots $\times S \rightarrow$ containers $\cup$ {nil} can be used as a state variable to identify which container is loaded on a robot, if any. ■

Since all state variables depend on the current state, the $(k+1)$th argument of a k-ary state variable will be left implicit: $x(v_1, \ldots, v_k) = v_{k+1}$ refers implicitly to the value of this state variable in the current state s.

A state variable $x(c_1, \ldots, c_k)$ is *ground* if each c_i is a constant in D_i^x. A state variable $x(v_1, \ldots, v_k)$ is *unground* if one or more of $v_1, \ldots, v_k$ are object variables. For example, rloc(r) is an unground state variable, and rloc(r1) and cpos(c1) are ground state variables.

A state variable is intended to be a characteristic attribute of a state. Hence, in this representation a state s is specified by giving the values in s of all the ground state variables. More specifically, for every ground state variable $x(c_1, \ldots, c_k)$, a state s includes a syntactic expression of the form $x(c_1, \ldots, c_k) = c_{k+1}$ such that c_{k+1} is the value of $x(c_1, \ldots, c_k)$ in s, each c_i being a constant in the appropriate range.

Notice that the state variables are not necessarily independent, in the sense that they may not take any values within their respective domains independently of each other. In Example 2.13 we have rload (r) = c iff cpos (c) = r, e.g., we cannot have a meaningful state s with rload(r1) = c1 and cpos(c1) = loc2.

Example 2.14 Continuing the previous example, consider a planning problem that has one robot (r1), three containers (c1, c2, c3), and three locations (l1, l2, l3). Here are some examples of possible states for this problem:

$$s_0 = \{\text{rloc(r1)} = \text{l1}, \text{rload(r1)} = \text{nil}, \text{cpos(c1)} = \text{l1}, \text{cpos(c2)} = \text{l2}, \text{cpos(c3)} = \text{l2}\}$$

$$s_1 = \{\text{rloc(r1)} = \text{l1}, \text{rload(r1)} = \text{c1}, \text{cpos(c1)} = \text{r1}, \text{cpos(c2)} = \text{l2}, \text{cpos(c3)} = \text{l2}\}$$

■

Some properties, such as adjacent(loc1, loc2) in the DWR domain, do not vary from one state to another. We called these properties *rigid relations*. For a k-ary rigid relation $r(v_1, \ldots, v_k)$, there will be sets $D_1^r, \ldots, D_k^r$ such that each D_i^r is the union of one or more classes and $r \subseteq D_1^r \times \ldots \times D_k^r$. For example, adjacent $\subseteq$ locations $\times$ locations.

By definition, rigid relations are invariant for a given planning domain. Hence, they do not need to be stated in every state. They have to be specified in the problem statement, together with the specification of the various classes of the constants in the domain, e.g., locations(loc2), robots(r1), containers(c3), adjacent(loc1,loc2), adjacent(loc2,loc3), etc.

2.5.2 Operators and Actions

Let us define planning operators and actions for the state-variable representation.

Definition 2.10 A *planning operator* is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$ where:

- name(o) is a syntactic expression of the form $n(u_1, \ldots, u_k)$, where n is a symbol called an *operator symbol*, $u_1, \ldots, u_k$ are all of the object variable symbols that appear anywhere in o, and n is unique (i.e., no two operators can have the same operator symbol).
- precond(o) is a set of expressions on state variables and relations.
- effects(o) is a set of assignments of values to state variables of the form $x(t_1, \ldots, t_k) \leftarrow t_{k+1}$, where each t_i is a term in the appropriate range. ■

Example 2.15 Consider the simplified DWR domain in which there are no piles and no cranes; robots can load and unload containers autonomously. This simplified domain has only three operators.

```
move(r, l, m)
    ;; robot r at location l moves to an adjacent location m
    precond: $\text{rloc}(r) = l, \text{adjacent}(l, m)$
    effects: $\text{rloc}(r) \leftarrow m$

load(c, r, l)
    ;; robot r loads container c at location l
    precond: $\text{rloc}(r) = l, \text{cpos}(c) = l, \text{rload}(r) = \text{nil}$
    effects: $\text{rload}(r) \leftarrow c, \text{cpos}(c) \leftarrow r$

unload(c, r, l)
    ;; robot r unloads container c at location l
    precond: $\text{rloc}(r) = l, \text{rload}(r) = c$
    effects: $\text{rload}(r) \leftarrow \text{nil}, \text{cpos}(c) \leftarrow l$
```

Here, c, r, l, and m are object variables. There are three state variables in these operators, rloc(r), rload(r), and cpos(c):

rloc: robots $\times S \rightarrow$ locations
rload: robots $\times S \rightarrow$ containers $\cup$ {nil}
cpos: containers $\times S \rightarrow$ locations $\cup$ robots

There is one rigid relation: adjacent $\subseteq$ locations $\times$ locations. To keep the notation simple, the state argument in state variables is implicit: the conditions in precond(a) refer to the values of state variables in a state s, whereas the updates in effects(a) refer to the values in the state $\gamma(s, a)$. This is similar to the notation used in the set-theoretic and classical representations where the effects$^+$ and effects$^-$ of an action refer also to changes in $\gamma(s, a)$. Note that the types of object variables, here left implicit, can be specified in a straightforward way in precond(o) with unary rigid relations: locations(l), robots(r), containers(c), etc. ■

An action a is a ground operator o that meets the rigid relations in precond(o), i.e., every object variable in o is replaced by a constant of the appropriate class such that these constants meet the rigid relations in precond(o). For example, the ground operator move(r1,loc1,loc4) is not considered to be an action if adjacent(loc1,loc4) does not hold in the domain.[7]

An action a is applicable in a state s when the values of the state variables in s meet the conditions in precond(a). In that case, the state $\gamma(s, a)$ is produced by updating the values of the state variables according to the assignments in effects(a). For example, in the domain of Figure 2.2 adjacent(loc2,loc1) holds, hence move(r1,loc2,loc1) is an action; this action is applicable to the state s_1 because rloc(r1) = loc2 in state s_1.

As in the classical representation, the set of states S is defined as the states that are reachable with the specified planning operators from some initial state, given the rigid relations of the problem.

A goal is given by specifying the values of one or more ground state variables. For example, g = {cpos(c1) = loc2, cpos(c2) = loc3} specifies that the container c1 has to be brought to location loc2 and c2 to loc3. More generally, a goal can be specified with a set of expressions on state variables, as in the preconditions of operators.

2.5.3 Domains and Problems

Let us now summarize what has been introduced so far and define more formally planning domains and problems. A state-variable representation relies on the following ingredients.

7. This is a simple matter of economy because this ground operator cannot contribute to any state transition.

- *Constant symbols*, also called *object symbols*, are partitioned into disjoint classes corresponding to the objects of the domain, e.g., the finite sets of robots, locations, cranes, containers, and piles in the DWR domain.
- *Object variable symbols*: these are typed variables, each of which ranges over a class or the union of classes of constants, e.g., $r \in$ robots, $l \in$ locations, etc.
- *State variable symbols*: these are functions from the set of states and zero or more sets of constants into a set of constants, e.g., rloc, rload, and cpos.
- *Relation symbols*: these are rigid relations on the constants that do not vary from state to state for a given planning domain, e.g., adjacent(loc1,loc2), belong(pile1,loc1), attached(crane1,loc1).
- *Planning operators*: they have preconditions that are sets of expressions on state variables and relations plus effects that are sets of assignments of values to state variables, as illustrated in Example 2.15.

A planning language $\mathcal{L}$ in the state-variable representation is given by the definition of a finite set of state variables X, a finite set of rigid relations R, and the sets of constants that define their domains.

Let X be the set of all ground state variables of $\mathcal{L}$, i.e., if x is a k-ary state variable in X, then for every tuple $(c_1, \ldots, c_k) \in D_1^x \times \ldots \times D_k^x$ that meets the rigid relations in R, the ground state variables $x(c_1, \ldots, c_k) \in X$.

Definition 2.11 Let $\mathcal{L}$ be a planning language in the state-variable representation defined by X and R. A *planning domain* in $\mathcal{L}$ is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq \prod_{x \in X} D_x$, where D_x is the range of the ground state variable x; a state s is denoted $s = \{(x = c) \mid x \in X\}$, where $c \in D_x$.
- $A = \{$all ground instances of operators in O that meet the relations in $R\}$, where O is a set of operators as defined earlier; an action a is applicable to a state s iff every expression $(x = c)$ in precond(a) is also in s.
- $\gamma(s, a) = \{(x = c) \mid x \in X\}$, where c is specified by an assignment $x \leftarrow c$ in effects(a) if there is such an assignment, otherwise $(x = c) \in s$.
- S is closed under γ, i.e., if $s \in S$, then for every action a that is applicable to s, $\gamma(s, a) \in S$.

■

Definition 2.12 A *planning problem* is a triple $\mathcal{P} = (\Sigma, s_0, g)$, where s_0 is an initial state in S and the goal g is a set of expressions on the state variables in X.

■

The goal g may contain unground expressions; e.g., $g = \{$cpos$(c) =$ loc2$\}$ requires moving any container c to loc2. A state s satisfies a goal g if there is a substitution σ such that every expression of $\sigma(g)$ is in s. A goal g corresponds to a set of goal states S_g defined as follows:

$$S_g = \{s \in S \mid s \text{ satisfies } g\}$$

As in the classical representation, we can define the relevance of an action a for a goal g and $\gamma^{-1}(g, a)$. If we restrict for simplicity to ground goals, then an action a is relevant for g iff both of the following conditions hold:

- $g \cap \text{effects}(a) \neq \emptyset$.
- For every expression $(x = c)$ in g, effects(a) contains no assignment of the form $x \leftarrow d$ such that c and d are different constant symbols.

If a is relevant for g, then $\gamma^{-1}(g, a) = (g - \vartheta(a)) \cup \text{precond}(a)$, where $\vartheta(a) = \{(x = c) \mid (x \leftarrow c) \in \text{effects}(a)\}$.

Definition 2.13 The *statement* of a planning problem is a 4-tuple $P = (O, R, s_0, g)$, where O is the set of operators, R is the set of rigid relations, s_0 is the initial state, and g is the goal. ■

Note that the statement of a planning problem is here a 4-tuple, instead of a 3-tuple as in the previous representations. Because rigid relations are not explicit in the specification of the state, they cannot be deduced from s_0, hence we need the additional term R in P.

A state-variable representation is ground if no state variable has any arguments. As we will discuss in Section 2.6, ground state-variable representations are similar to set-theoretic representations, and unground state-variable representations are similar to classical representations. Note that a ground state-variable representation does not have rigid relations because it does not have object variables: these relations need to be specified implicitly through the actions in A.

As in the classical representation, a plan for a problem P is a sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$ such that a_1 is applicable to s_0, i.e., the conditions and relations in precond(a_1) are met in state s_0, a_2 is applicable in $\gamma(s_0, a_1), \ldots,$ and the conditions in the goal g are met in $\gamma(s_0, a_k)$.

2.5.4 Properties

We have not introduced logical connectors in preconditions or effects, neither in states or goals of this representation. Clearly, the lack of negation operators is not a restrictive assumption on the expressiveness of the representation.

- The negation in the effects of classical planning operators is already expressed here through the state-variable assignments because we are using functions instead of relations. For example, rloc(r) $\leftarrow m$ in the move operator of Example 2.15 corresponds to the two literals at(r, m) and $\neg$ at(r, l) in the effects of that operator in Example 2.8.

- The negation in preconditions can be handled through a simple syntactical change, as discussed in the following section (in the comparison between the set-theoretic and the classical representations).

Furthermore, complex expressions on state variables, such as rload(r) $\neq$ nil or cpos(c) $\in$ {loc1, loc2}, can easily be introduced in preconditions and goals, as will be illustrated in Section 8.3. Other extensions, similar to those discussed in Section 2.4, can also be added to the state-variable representation.

Recall that here a state s is a tuple $s \in \prod_{x \in X} D_x$: we said that s is specified by giving the value of every ground state variable in X. In order to avoid a fastidious enumeration of all the ground state variables,[8] it is possible to rely on a *default value assumption*, similar to the *closed-world assumption* that assumes the value false for every atom not explicitly stated as true. More interesting schemes are possible when the range of a state variable is an ordered set or a lattice.

Finally, let us go back to the advantages of the state-variable representation. In classical planning, these advantages are simply the conciseness of the functional notation. This conciseness goes beyond what has been illustrated up to here. For example, some of the axiomatic inference extensions of classical planning, discussed in Section 2.4.5, are already provided in the state-variable representation. An axiom such as Formula 2.2, which states that a crane is empty iff it is not holding anything, is simply expressed through a state variable. For example, holds: cranes $\times$ S $\rightarrow$ containers $\cup$ {nil} tells us that the crane k is empty iff holds(k) = nil. Similarly, a state variable that denotes what object, if any, is above a container, e.g., above: (containers $\cup$ {pallet}) $\times$ S $\rightarrow$ containers $\cup$ {nil}, tells us that a container c is at the top of a pile iff above(c) = nil. Hence, we do not need to manage explicitly state variables equivalent to the predicates empty and top. Other advantages of the state-variable representation will appear later, in particular in Part IV, when we will go beyond classical planning.

2.6 Comparisons

From a theoretical point of view, the classical and state-variable representation schemes have equivalent expressivity: a problem represented in one of them can easily be translated into the other with at most a linear increase in size. To translate a classical representation into a state-variable representation, replace every positive literal $p(t_1, \ldots, t_n)$ with a state-variable expression of the form $p(t_1, \ldots, t_n) = 1$ and every negative literal $\neg p(t_1, \ldots, t_n)$ with a state-variable expression of the form $p(t_1, \ldots, t_n) = 0$.

Here is how to translate a state-variable representation into a classical representation. First, in the initial state and the preconditions of all operators, replace every

8. For the same domain, there are in general fewer ground state variables than ground atoms in the classical representation: if $|D_x| = 2^m$ then x corresponds to m propositions.

state-variable expression $x(t_1, \ldots, t_n) = v$ with an atom of the form $x(t_1, \ldots, t_n, v)$. Next, for every state-variable assignment $x(t_1, \ldots, t_n) \leftarrow v$ that appears in the effects of an operator, replace it with a pair of literals $\neg x(t_1, \ldots, t_n, w)$ and $x(t_1, \ldots, t_n, v)$, and insert $x(t_1, \ldots, x_n, w)$ into the operator's preconditions.

Classical and state-variable representations are more expressive than set-theoretic representations. Although all three representation schemes can still represent the same set of planning domains, a set-theoretic representation of a problem may take exponentially more space than the equivalent classical and state-variable representations. For example, suppose we have a classical representation in which the number of constant symbols is k. Then each n-ary predicate corresponds to k^n propositions, and if a classical operator's preconditions and effects lists each contain two n-ary predicates, then the operator corresponds to $(4k)^n$ set-theoretic actions. For a more concrete example, see Example 2.5 (see page 25).

If we restrict all of the atoms and state variables to be ground, then from a theoretical point of view the set-theoretic, classical, and state-variable representation schemes are essentially equivalent: each can be translated into the other with at most a linear increase in size. More specifically:

- Ground classical and ground state-variable representations can be translated into each other as described earlier.
- To translate a set-theoretic representation into a ground classical representation, replace each action $a = (\text{precond}(a), \text{effects}^+(a), \text{effects}^-(a))$ with an operator whose preconditions and effects are $\text{precond}(a)$ and $\text{effects}^+(a) \cup \{\neg(e) \mid e \in \text{effects}^-(a)\}$.
- To translate a ground classical representation into a set-theoretic representation, do the following. First, for every negated atom $\neg e$ that appears anywhere as a precondition, create a new atom e', replace $\neg e$ with e' wherever the latter occurs as a precondition, add an additional effect $\neg e'$ to every operator whose effects include e, and add an additional effect e' to every operator whose effects include $\neg e$. Next, replace each ground atom $p(c_1, \ldots, c_n)$ (where each c_i is a constant symbol) with a proposition symbol $p_{c_1,\ldots,c_n}$. Next, replace each operator $(o, \text{precond}(o), \text{effects}(o))$ with an action $(\text{precond}(o), \text{effects}^+(o), \text{effects}^-(o))$.
- Ground state-variable representations can be translated into set-theoretic representations and vice versa by first translating them into classical representations and then translating them into the desired representation with one of the methods outlined earlier.

Beyond these theoretical considerations, there may be practical reasons why one or another of these representation schemes is preferable in a given domain. For example, if we want to express a concept that is essentially a single-valued function, e.g., the location of a robot r1, then it may be more convenient to use a state-variable representation (e.g., location(r1) = loc1) than a classical representation (e.g., location(r1,loc1)) because the former avoids the necessity of explicitly deleting location(r1,loc1) in order to assert location(r1,loc2).

2.7 Discussion and Historical Remarks

In the planning literature, there has often been confusion between the language for specifying a planning problem and the model of that problem (i.e., the state-transition system). This confusion is reflected in our somewhat awkward distinction between a planning problem and the statement of a planning problem (see Section 2.2.3).

Early work on automated planning was heavily influenced by work on automated theorem proving. One of the first formulations of automated planning [249] used an axiomatic description of the initial state, goal, and planning operators; used resolution theorem proving to produce a constructive proof that a plan exists; and then used answer extraction to find the actual plan (see Chapter 12). However, this approach was hampered by a problem that became known as the *frame problem* [383], which was the problem of specifying axiomatically not only what changes an operator would make to a state but also what elements of the state it would leave unchanged. One motivation for the development of the classical planning formulation was that it provided a simple solution to the frame problem: in the classical formulation, any atom not mentioned in an operator's effects remains unchanged.

The classical representation scheme is linked to STRIPS (see [189] and Chapter 4), an early automated planning system. In the original version of STRIPS, the states and planning operators were similar to those of classical planning, but the operators had more expressive power. Each operator had a precondition list, add list, and delete list, and these were allowed to contain arbitrary well-formed formulas in first-order logic. However, there were a number of problems with this formulation, such as the difficulty of providing a well-defined semantics for it [362].

In subsequent work, researchers restricted the representation so that the preconditions, add lists, and delete lists would contain only atoms. Nilsson used this formulation to describe STRIPS in his 1980 textbook [426]. Hence, the operators and the representation scheme became known as STRIPS-style operators and STRIPS-style planning, respectively.

In the UCPOP planner [436], Penberthy and Weld introduced a syntactic modification of STRIPS-style operators in which the operators did not have add lists and delete lists but instead had positive and negative effects. This representation is the basis of our classical planning operators. Classical planning and STRIPS-style planning have equivalent expressive power, in the sense that any classical planning problem can easily be translated into a STRIPS-style planning problem with the same set of solutions and vice versa (see Exercise 2.13).

The semantics of classical planning has been addressed by Lifschitz [362]. The community working on reasoning on action and change studies more ambitious representations in a logical framework, e.g., for handling the frame problem in general state-transition systems (without the restrictive assumptions A0 through A7 introduced in Chapter 1), for dealing with operators that may have implicit effects (the so-called ramification problem) and implicit preconditions (the qualification problem). A general framework for the study of the representations of actions is proposed by Sandewall [463].

The Action Description Language (ADL) representation, introduced by Pednault [434, 435], proposes a trade-off between the expressiveness of general logical formalism and the computation complexity of reasoning with that representation, i.e., computing the transition function γ. Starting from UCPOP [436], several planners [120, 142, 433, 438] were generalized to ADL or to representations close to ADL that handle most of the extensions introduced in Section 2.4. These extensions have been implemented in the PDDL planning language used in the AIPS (International Conference on AI Planning and Scheduling) planning competitions [196, 387].

Syntactic constructs similar to our state variables have been used in operations research for many years [422] and have been used indirectly in AI planning, e.g., to encode classical planning problems as integer programming problems [539]. The idea of using state variables rather than logical atoms directly in the planning operators was introduced in Bäckströms's planner SAS [38, 39, 40, 286] and in several other planners such as IxTeT [224]. We will come back to the latter in the temporal planning chapter (Chapter 14). The work on the former planner led to the extensive study of the complexity of several classes of state-variable planning, corresponding to a broad range of interesting syntactic and structural restrictions.

The set-theoretic, classical, and state-variable representation schemes all use a special case of the well-known *closed-world assumption*: the only atoms or state-variable expressions that are true in a state are the ones explicitly specified in the state.

The notion of two planning formalisms having "equivalent expressivity," which we discussed informally in Section 2.6, has been defined formally by Nebel [418], who defines two representation schemes to have equivalent expressivity if there are polynomial-time translations between them that preserve plan size exactly.

In this chapter, we have avoided discussing a plan's cost and whether it is optimal. This notion was not emphasized in the early years of research on classical planning because it was difficult enough to find a plan at all without worrying about whether the plan was optimal. However, some of the later formulations of classical planning (e.g., [560]) include cost explicitly. The usual way of doing this is to assume that each action has a cost and that the cost of a plan is the sum of the costs of its actions. However, this formulation is relatively limited in comparison with more sophisticated notions used in fields such as scheduling and operations research, such as monotonic cost functions, flexibility of a plan in order to allow for changes, penalties for achieving some but not all goals [355], and Pareto optimality [209].

2.8 Exercises

2.1 Here is a classical planning problem that involves moving blocks around on a table.

$s_0 = \{\texttt{on(c1, table)}, \texttt{on(c3, c1)}, \texttt{clear(c3)}, \texttt{on(c2, table)}, \texttt{clear(c2)}\}$
$g = \{\texttt{on(c2, c2)}, \texttt{on(c3, c3)}\}$

```
pickup(x)
   precond: on(x,table), clear(x)
   effects:  ¬on(x,table), ¬clear(x), holding(x)

putdown(x)
   effects:  holding(x)
   precond: ¬holding(x), on(x,table), clear(x)

unstack(x, y)
   precond: on(x, y), clear(x)
   effects:  ¬on(x, y), ¬clear(x), holding(x), clear(y)

stack(x, y)
   effects:  holding(x), clear(y)
   precond: ¬holding(x), ¬clear(y), on(x,table), clear(x)
```

(a) Rewrite the problem as a set-theoretic planning problem.

(b) Why are there separate operators for pickup and unstack, rather than a single operator for both?

(c) In the DWR domain, why do we not need two operators analogous to pickup and unstack for taking containers from a pile with a crane?

(d) What else can you say about the similarities and differences between this formulation of the Sussman anomaly and the one in Example 4.3 (see page 23)?

(e) Rewrite the problem as a state-variable planning problem.

(f) Rewrite the problem as a state-variable planning problem in which no state variable has any arguments.

2.2 Let $P_1 = (O, s_0, g_1)$ and $P_2 = (O, s_0, g_2)$ be two classical planning problems that have the same operators and initial state. Let $\pi_1 = \langle a_1, \ldots, a_n \rangle$ be any solution for P_1, and $\pi_2 = \langle b_1, \ldots, b_n \rangle$ be any solution for P_2.

(a) If $\pi_1 \cdot \pi_2$ is executable in P_2, then is it a solution for P_2?

(b) If no operator has negative effects, then is $\langle a_1, b_1, a_2, b_2, \ldots, a_n, b_n \rangle$ a solution for $(O, s_0, g_1 \cup g_2)$?

(c) Suppose no operator has any negative effects, and let π_2' be any permutation of π_2. Is π_2' a solution for P_2?

2.3 Let $P = (O, s_0, g)$ be the statement of a classical planning problem. In each of the following, explain the reason for your answer.

(a) Let $\Pi = \{\pi \mid \pi \text{ is applicable to } s_0\}$. Is Π finite?

(b) Let $S = \{\gamma(s_0, \pi) \mid \pi \in \Pi\}$. Is S finite?

(c) Let $\Pi' = \{\pi \mid \pi$ is nonredundant and is applicable to $s_0\}$. Is Π' finite?

(d) Suppose no operator has any negative effects, and let π' be any permutation of π. Is π' a solution for P?

(e) If π_1 and π_2 are plans, then π_1 is a *subplan* of π_2 if all actions in π_1 also appear in π_2 in the same order (although possibly with other actions in between). If π is a solution for P, then will π have a subplan that is a shortest solution for P?

(f) Let π be a solution for P, and let $P' = (O, s_0, g')$, where $g \subseteq g'$. Is π a subplan of a solution for P'?

2.4 Let Σ be the planning domain described in Example 2.7 (see page 27). Suppose we map the classical representation of Σ into a set-theoretic representation as described in Section 2.6. How many states will the set-theoretic representation contain? How many actions will it contain?

2.5 Specify the operators of the DWR domain of Example 2.8 (see page 29) entirely in a state-variable representation. Apply this set of operators to the domain of Example 2.7 extended with a third pile p3 and a second crane, both in loc2. The initial state is as depicted in Figure 2.2 with p3 initially empty. The goal is to have all containers in pile p3 in the following order: pallet, c3, c2, c1.

2.6 In the planning domain described in Example 2.7 (see page 27), let s be a state that is missing one or more atoms of the form in(c, p), while the position of the containers is correctly specified by the on(c, c$'$) predicates. Write a preprocessing procedure that will restore the missing atoms.

2.7 In the planning domain described in Example 2.7 (see page 27), let s be a state that is missing one or more atoms of the form top(c, p). Write a preprocessing procedure that will restore the missing atoms.

2.8 In the planning domain described in Example 2.7 (see page 27), let s be a state that is missing one or more atoms of the form on$(c,$ pallet$)$. Write a preprocessing procedure that will restore the missing atoms.

2.9 Which of the three previous preprocessing procedures would also work correctly on goal formulas? Why?

2.10 How will it change the DWR domain if we redefine the take operator as follows?

```
take(k, l, c, d, p)
    ;; crane k at location l takes c off d in pile p
    precond: belong(k, l), attached(p, l), empty(k),
             in(c, p), top(c, p), on(c, d), in(d, p)
    effects: holding(k, c), ¬ in(c, p), ¬ top(c, p),
             ¬ on(c, d), top(d, p), ¬ empty(k)
```

2.11 In the definition of a classical operator, why did we require that each variable of precond(o) and effects(o) must be a parameter of o?

2.12 Give the statement of a classical planning problem and a solution π for the problem such that π is not redundant and not shortest.

2.13 Rewrite the DWR domain in the STRIPS-style planning formalism of Section 2.7. (Hint: Replace each negated atom with a new nonnegated atom.)

2.14 Let P be the statement of a classical planning problem in which all of the predicates are ground (i.e., there are no variable symbols). Write an algorithm for converting P into the statement of a set-theoretic planning problem.

2.15 Suppose a state of the world is an arbitrary pair of integer coordinates (x, y), and thus S is infinite. Suppose we have an action a such that $\gamma((x, y), a)$ is the state $(x + 1, y + 1)$. How many classical planning operators are needed to express a?

2.16 If we extended the classical representation scheme to allow function symbols, then how would your answer to Exercise 2.15 change?

2.17 Show that the total number of states for the domain corresponding to Figure 2.1 is $8n(n!)$ if there are $n > 0$ containers.

2.18 Prove Proposition 2.1 (see page 23).

2.19 Prove Proposition 2.2 (see page 23).

2.20 Prove Proposition 2.3 (see page 24).

2.21 How many classical planning operators would you need to represent the incr operator of Example 2.5 (see page 25)? How many conditional operators would you need?

2.22 One way function symbols can be used is to encode lists. For example, the list $\langle$a, b, c$\rangle$ can be represented as cons(a, cons(b, cons(c, nil))), where cons is a binary function symbol and nil is a constant symbol. Write planning operators to put an element onto the front of a list and to remove the first element of a nonempty list.