Knowledge Representation and Reasoning

Ronald J. Brachman
AT&T Labs – Research
Florham Park, New Jersey
USA 07932
rjb@research.att.com

Hector J. Levesque
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 3H5
hector@cs.toronto.edu

Chapter 14

Actions

The language of FOL is sometimes criticized as being an overly "static" representation formalism. Sentences of FOL are either true or false in an interpretation and stay that way. Unlike procedural representations or production systems, there is seemingly nothing in FOL corresponding to any sort of *change*.

In fact, there are two sorts of changes that we might want to consider. First, there is the idea of changing what is believed about the world. Suppose α is a sentence saying that birds are the descendants of dinosaurs. At some point, you might come to believe that α is true, perhaps by being told directly. If you had no beliefs about α before, this is a straightforward process that involves adding α to your current KB. If you had previously thought that α was false, however, perhaps having concluded this from a number of other beliefs, dealing with the new information is a much more complicated process. The study of which of your old beliefs to discard is an important area of research known as *belief revision*, but one that is beyond the scope of this book.

The second notion of change to consider is when the beliefs themselves are about a changing world. Instead of merely believing that John is a student, for example, you might believe that John was not a student initially, but that he became a student by enrolling at a university, and that he later graduated, and ceased to be a student. In this case, while the world you are imagining is certainly changing, the beliefs you have about John's history as a whole need not change at all.[1]

In this chapter, we will study how beliefs about a changing world of this sort can in fact be represented in a dialect of FOL called the *situation calculus*. This is not the only way to represent a changing world, of course, but it is a simple and powerful way to do so. It also naturally lends itself to various sorts of reasoning, including planning, discussed separately in the next chapter.

[1] Of course, we might also have changing beliefs about a changing world, but we will not pursue this here.

14.1 The situation calculus

One way of thinking about change is to imagine being in a certain situation, with actions moving you from one situation to the next. The situation calculus is a dialect of FOL in which such situations and actions are taken to be objects in the domain. In particular, there are two distinguished sorts of first-order terms:

- *actions*: such as jump (the act of jumping), kick(x) (kicking object x), and put(r, x, y) (robot r putting object x on top of object y). The constant and function symbols for actions are completely application-dependent.
- *situations*, which denote possible world histories. A distinguished constant S_0 and function symbol *do* are used. S_0 denotes the initial situation, before any action has been performed; *do*(a, s) denotes the situation that results from performing action a in situation s.

For example, the situation term *do*(pickup(b_2), *do*(pickup(b_1), S_0)) denotes the situation that results from first picking up object b_1 in S_0 and then picking up object b_2. Note that this situation is not the same as *do*(pickup(b_1), *do*(pickup(b_2), S_0)), since they have different histories, even though the resulting states may be indistinguishable.

14.1.1 Fluents

Predicates and functions whose values may vary from situation to situation are called *fluents*, and are used to describe what holds in a situation. By convention, the last argument of a fluent is a situation. For example, the fluent Holding(r, x, s) might stand for the relation of robot r holding object x in situation s. Thus, we can have formulas like

$$\neg\text{Holding}(r, x, s) \wedge \text{Holding}(r, x, do(\text{pickup}(r, x), s))$$

which says that robot r is not holding x in some situation s, but is holding x in the situation that results from picking it up. Note that in the situation calculus there is no distinguished "current" situation. A single formula like this can talk about many different situations, past, present, or future.

Finally, a distinguished predicate *Poss*(a, s) is used to state that action a can be performed in situation s. For example,

$$\mathit{Poss}(\mathsf{pickup}(r, x), S_0)$$

says that the robot r is able to pick up object x in the initial situation.

This completes the specification of the dialect.

14.1.2 Precondition and effect axioms

To reason about a changing world, it is necessary to have beliefs not only about what is true initially, but also about how the world changes as the result of actions.

Actions typically have *preconditions*, that is, conditions that need to be true for the action to occur. For example, in a robotics setting, we might have the following:

- a robot can pick up an object if and only if it is not holding anything, the object is not too heavy, and the robot is next to the object:[2]

$$\mathit{Poss}(\mathsf{pickup}(r, x), s) \equiv \forall z. \neg\mathsf{Holding}(r, z, s) \wedge \neg\mathsf{Heavy}(x) \wedge \mathsf{NextTo}(r, x, s);$$

- it is possible for a robot to repair an object if and only if the object is broken and there is glue available:

$$\mathit{Poss}(\mathsf{repair}(r, x), s) \equiv \mathsf{Broken}(x, s) \wedge \mathsf{HasGlue}(r, s).$$

Actions typically also have *effects*, that is, fluents that are changed as a result of performing the action. For example,

- dropping a fragile object causes it to break:

$$\mathsf{Fragile}(x) \supset \mathsf{Broken}(x, \mathit{do}(\mathsf{drop}(r, x), s));$$

- repairing an object causes it to be unbroken:

$$\neg\mathsf{Broken}(x, \mathit{do}(\mathsf{repair}(r, x), s)).$$

Formulas like those above are often called *precondition axioms* and *effect axioms* respectively.[3] Effect axioms are called *positive* if they describe when a fluent becomes true, and *negative* otherwise.

[2] In this chapter, free variables should be assumed to be universally quantified from the outside.

[3] These are called "axioms" for historical reasons: a KB can be thought of as the axioms of a logical theory (like number theory or set theory), with the entailed beliefs considered as theorems.

14.1.3 Frame axioms

To fully capture the dynamics of a situation, we need to go beyond the preconditions and effects of actions. So far, if a fluent is not mentioned in an effect axiom for an action a, we would not know anything at all about it in the situation *do*(a, s). To really know how the world can change, it is also necessary to know what fluents are *unaffected* by performing an action. For example,

- dropping an object does not change its colour:

$$\mathsf{Colour}(x, c, s) \supset \mathsf{Colour}(x, c, \mathit{do}(\mathsf{drop}(r, x), s));$$

- dropping an object y does not break an object x when $x \neq y$ or x is not fragile:

$$\neg\mathsf{Broken}(x, s) \wedge [x \neq y \vee \neg\mathsf{Fragile}(x)] \supset \neg\mathsf{Broken}(x, \mathit{do}(\mathsf{drop}(r, y), s)).$$

Formulas like these are often called *frame axioms*. Observe that we would not normally expect them to be entailed by the precondition or effect axioms for the actions involved.

Frame axioms do present a serious problem, however, sometimes called the *frame problem*. Simply put, the problem is that it will be necessary to know and reason effectively with an extremely large number of frame axioms. Indeed, for any given fluent, we would expect that only a very small number of actions affect the value of that fluent; the rest leave it invariant. For instance, an object's colour is unaffected by picking things up, opening a door, using the phone, making linguini, walking the dog, electing a new Prime Minister of Canada *etc. etc.* All of these will require frame axioms. It seems very counterintuitive that we should need to even think about these $\approx 2 \times \mathcal{A} \times \mathcal{F}$ facts (where $\mathcal{A}$ is the number of actions, and $\mathcal{F}$, the number of fluents) about what does not change when we perform an action.

What counts as a solution to this problem? Suppose the person responsible for building a KB has written down *all* the relevant effect axioms. That is, for each fluent $F(\vec{x}, s)$ and action a that can cause the fluent to change, we have an effect axiom of the form

$$\phi(\vec{x}, s) \supset (\neg)F(\vec{x}, \mathit{do}(a, s)),$$

where $\phi(\vec{x}, s)$ is some condition on situation s. What we would like is a systematic procedure for generating all the frame axioms from these effect axioms. Moreover, if possible, we also want a parsimonious representation for them, since in their simplest form, there are too many.

And why do we want such a solution? There are at least three reasons:

- Frame axioms are necessary beliefs about a dynamic world that are not entailed by other beliefs we may have.
- For the convenience of the KB builder: generating the frame axioms automatically gives us modularity, since only the effect axioms need to be given by hand. This ensures there is no inadvertent omission or error.
- Such a solution is useful for theorizing about actions: we can see what assumptions need to be made to draw conclusions about what does not change.

We will examine a simple solution to the frame problem in Section 14.2.

14.1.4 Using the situation calculus

Given a KB containing facts expressed in the situation calculus as above, there are various sorts of reasoning tasks we can consider. We will see in the next chapter that we can do planning. In Section 14.3, we will see that we can figure out how to execute a high-level action specification. Here we consider two basic reasoning tasks: projection and legality testing.

The *projection task* is the following: given a sequence of actions and some initial situation, determine what would be true if those actions were performed starting in that initial situation. This can be formalized as follows:

> Suppose that $\phi(s)$ is a formula with a single free variable s of the situation sort, and that $\vec{a}$ is a sequence of actions $\langle a_1, \ldots, a_n \rangle$. To find out if $\phi(s)$ would be true after performing $\vec{a}$ starting in the initial situation S_0, we determine whether or not KB $\models \phi(do(\vec{a}, S_0))$, where $do(\vec{a}, S_0)$ is an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, S_0)) \ldots))$.

For example, using the above effect and frame axioms, it follows that the fluent $\neg\mathsf{Broken}(b_2, s)$ would hold after the sequence of actions

$$\langle \mathsf{pickup}(b_1), \mathsf{pickup}(b_2), \mathsf{drop}(b_2), \mathsf{repair}(b_2), \mathsf{drop}(b_1) \rangle .$$

In other words, the fluent holds in the situation

$$s = do(\mathsf{drop}(b_1), do(\mathsf{repair}(b_2), do(\mathsf{drop}(b_2), do(\mathsf{pickup}(b_2), do(\mathsf{pickup}(b_1), S_0))))).$$

It is a separate matter to determine whether or not the given sequence of actions could in fact be performed starting in the initial situation. This is called the *legality testing task*. For example, a robot might not be able to pick up more than one object at a time. We call a situation term *legal* if it is either the initial situation,

or the result of performing an action whose preconditions are satisfied starting in a legal situation. For example, although the term

$$do(\mathsf{pickup}(b_2), do(\mathsf{pickup}(b_1), S_0))$$

is well formed, it is not a legal situation, since the precondition for picking up b_2 (*e.g* not holding anything) will not be satisfied in a situation where b_1 has already been picked up. So the legality task is determining whether a sequence of actions leads to a legal situation. This can be formalized as follows:

> Suppose that $\vec{a}$ is a sequence of actions $\langle a_1, \ldots, a_n \rangle$. To find out if $\vec{a}$ can be legally performed starting in the initial situation S_0, we determine whether or not KB $\models Poss(a_i, do(\langle a_1, \ldots, a_{i-1} \rangle, S_0))$ for every i such that $1 \leq i \leq n$.

Before concluding this section on the situation calculus, it is perhaps worth noting some of the representational limitations of this language:

- *single agent*: there are no unknown or unobserved exogenous actions performed by other agents, and no unnamed events;
- *no time*: we have not talked about how long an action takes, or when it occurs;
- *no concurrency*: if a situation is the result of performing two actions, one of them is performed first and the other afterwards;
- *discrete actions*: there are no continuous actions like pushing an object from one point to another, or a bathtub filling with water;
- *only hypotheticals*: we cannot say that an action *has* occurred in reality, or *will* occur;
- *only primitive actions*: there are no actions that are constructed from other actions as parts, such as iterations or conditionals.

Many of these limitations can be dealt with by refinements and extensions to the dialect of the situation calculus considered here. We will deal with the last of these in Section 14.3 below.

But first we turn to a solution to the frame problem.

14.2 A simple solution to the frame problem

The solution to the frame problem we will consider depends on first putting all effect axioms into a normal form.

Suppose, for example, that there are two positive effect axioms for the fluent Broken:

$$\mathsf{Fragile}(x) \supset \mathsf{Broken}(x, do(\mathsf{drop}(r, x), s))$$
$$\mathsf{NextTo}(b, x, s) \supset \mathsf{Broken}(x, do(\mathsf{explode}(b), s)).$$

So an object is broken if it is fragile and it was dropped, or something next to it exploded. Using a universally quantified action variable a, these can be rewritten as a single formula

$$\exists r\{a = \mathsf{drop}(r, x) \wedge \mathsf{Fragile}(x)\} \vee$$
$$\exists b\{a = \mathsf{explode}(b) \wedge \mathsf{NextTo}(b, x, s)\} \supset$$
$$\mathsf{Broken}(x, do(a, s))$$

Similarly, a negative effect axiom like

$$\neg\mathsf{Broken}(x, do(\mathsf{repair}(r, x), s)),$$

saying that an object is not broken after it is repaired, can be rewritten as

$$\exists r\{a = \mathsf{repair}(r, x)\} \supset \neg\mathsf{Broken}(x, do(a, s)).$$

In general, for any fluent $F(\vec{x}, s)$, we can rewrite all of the positive effect axioms as a single formula of the form

$$\Pi_F(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)), \tag{1}$$

and all the negative effect axioms as a single formula of the form

$$\mathrm{N}_F(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)), \tag{2}$$

where $\Pi_F(\vec{x}, a, s)$ and $\mathrm{N}_F(\vec{x}, a, s)$ are formulas whose free variables are among the x_i, a, and s.

14.2.1 Explanation closure

Now imagine that we make a completeness assumption about the effect axioms we have for a fluent: assume that formulas (1) and (2) above characterize *all* the conditions under which an action a changes the value of fluent F. We can in fact formalize this assumption using what are called *explanation closure axioms* as follows:

$$\neg F(\vec{x}, s) \wedge F(\vec{x}, do(a, s)) \supset \Pi_F(\vec{x}, a, s) \tag{3}$$
if F were false, and made true by doing action a, then condition Π_F must have been true;

$$F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a, s)) \supset \mathrm{N}_F(\vec{x}, a, s) \tag{4}$$
if F were true, and made false by doing action a, then condition N_F must have been true.

Informally, these axioms add an "only if" component to the normal form effect axioms: (1) says that F is made true if Π_F holds, while (3) says that F is made true only if Π_F holds.[4] In fact, by rewriting them slightly, these explanation closure axioms can be seen to be disguised versions of frame axioms:

$$\neg F(\vec{x}, s) \wedge \neg\Pi_F(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$
$$F(\vec{x}, s) \wedge \neg\mathrm{N}_F(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)).$$

In other words, F remains false after doing a when Π_F is false, and F remains true after doing a when N_F is false.

14.2.2 Successor state axioms

If we are willing to make two assumptions about our KB, the formulas (1), (2), (3), and (4) can be combined in a particularly simple and elegant way. Specifically, we assume that our KB entails the following:

- integrity of the effect axioms for every fluent F:

  $$\neg\exists \vec{x}, a, s.\, \Pi_F(\vec{x}, a, s) \wedge \mathrm{N}_F(\vec{x}, a, s)$$

- unique names for actions:

  $$A(\vec{x}) = A(\vec{y}) \supset (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$
  $$A(\vec{x}) \neq B(\vec{y}), \text{ where } A \text{ and } B \text{ are distinct action names}$$

The first assumption is merely that no action a satisfies the condition to make the fluent F both true and false. The second assumption is that the only action terms that can be equal are two identical actions with identical arguments.

With these two assumptions, it can be shown that for any fluent F, KB entails that (1), (2), (3), and (4) together are logically equivalent to the following formula:

$$F(\vec{x}, do(a, s)) \equiv \Pi_F(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\mathrm{N}_F(\vec{x}, a, s)).$$

[4] Note that in (3) we need to ensure that F was originally false and was made true to be able to conclude that Π_F held, and similarly for (4).

A formula of this form is called a *successor state axiom* for the fluent F because it completely characterizes the value of fluent F in the successor state resulting from performing action a in situation s. Specifically, F is true after doing a if and only if before doing a, Π_F (the positive effect condition for F) was true or both F and $\neg \mathrm{N}_F$ (the negative effect condition for F) were true. For example, for the fluent Broken, we have the following successor state axiom:

$$\begin{aligned}&\mathsf{Broken}(x, do(a, s)) \equiv \\ &\quad \exists r\{a = \mathsf{drop}(r, x) \wedge \mathsf{Fragile}(x)\} \vee \\ &\quad \exists b\{a = \mathsf{explode}(b) \wedge \mathsf{NextTo}(b, x, s)\} \vee \\ &\quad \mathsf{Broken}(x, s) \wedge \forall r\{a \neq \mathsf{repair}(r, x)\}\end{aligned}$$

This says that an object x is broken after doing action a if and only if a is a dropping action and x is fragile, or a is a bomb exploding action when x is near to the bomb, or x was already broken and a is not the action of repairing it.

Note that it follows from this axiom that dropping a fragile object will break it. Moreover, it also follows logically that talking on the phone does not affect whether or not an object is broken (assuming unique names, *i.e.* talking on the phone is distinct from any dropping, exploding, or repairing action). Thus a KB containing this single axiom would entail all the necessary effect and frame axioms for the fluent in question.

14.2.3 Summary

We have, therefore, a simple solution to the frame problem in terms of the following axioms:

- successor state axioms, one per fluent,
- precondition axioms, one per action,
- unique name axioms for actions.

Observe that we do not get a small number of axioms at the expense of prohibitively long ones. The length of a successor state axiom is roughly proportional to the number of actions that affect the value of the fluent, and, as we noted earlier, we do not expect in general that very many of the actions would change the value of any given fluent.

The conciseness and perspicuity of this solution to the frame problem clearly depends on three factors:

1. the ability to quantify over actions, so that only actions changing the fluent need to be mentioned by name;
2. the assumption that relatively few actions affect each fluent, which keeps the successor state axioms short;
3. the completeness assumption for the effects of actions, which allows us to conclude that actions that are not mentioned explicitly in effect axioms leave the fluent invariant.

The solution also depends on being able to put effect axioms in the normal form used above. This would not be possible, for example, if we had actions whose effects were *nondeterministic*. For example, imagine an action flipcoin whose effect is to make either the fluent Heads or the fluent Tails true. An effect axiom like

$$\mathsf{Heads}(do(\mathsf{flipcoin}, s)) \vee \mathsf{Tails}(do(\mathsf{flipcoin}, s))$$

cannot be put into the required normal form. In general, we need to assume that every action a is deterministic in the sense that all the given effect axioms are of the form

$$\phi(\vec{x}, s) \supset (\neg) F(\vec{x}, do(a, s)).$$

How to deal in some way with nondeterministic choice and other complex actions is the topic of the next section.

14.3 Complex actions

So far, in our treatment of the situation calculus, we have assumed that there are only primitive actions, with effects and preconditions independent of each other. We have no way of handling *complex actions*, that is to say, actions that have other actions as components. Examples of these are actions like the following:

- *conditionals*: if the car is in the driveway then drive and otherwise walk;
- *iterations*: while there are blocks on the table, remove one;
- *nondeterministic choice*: pick a red block up off the table and put it on the floor;

and others, as described below. What we would like to do is to *define* such actions in terms of their primitive components in such a way that we can inherit their solution to the frame problem. To do this, we need a compositional treatment of the frame problem for complex actions. This is precisely what we will provide, and we will see that it results in a novel kind of programming language.

14.3.1 The Do formula

To handle complex actions in general, it is sufficient to show that for each complex action A we care about, there is a formula of the situation calculus, which we call $\mathbf{Do}(A, s, s')$, that says that action A when started in situation s can terminate legally in situation s'. Because complex actions can be nondeterministic, there may be more than one such s'. Consider, for example, the complex action

$$[\mathsf{pickup}(b_1)\ ;\ \textit{if}\ \mathsf{InRoom}(\mathsf{kitchen})\ \textit{then}\ \mathsf{putaway}(b_1)\ \textit{else}\ \mathsf{goto}(\mathsf{kitchen})].$$

For this action to start in situation s and terminate legally in s', the following sentence must be true:

$$\begin{array}{l}
\mathit{Poss}(\mathsf{pickup}(b_1), s)\ \wedge \\
\quad [\ (\mathsf{InRoom}(\mathsf{kitchen}, \mathit{do}(\mathsf{pickup}(b_1), s)) \\
\qquad \wedge\ \mathit{Poss}(\mathsf{putaway}(b_1), \mathit{do}(\mathsf{pickup}(b_1), s)) \\
\qquad \wedge\ s' = \mathit{do}(\mathsf{putaway}(b_1), \mathit{do}(\mathsf{pickup}(b_1), s))) \\
\qquad\quad \vee \\
\quad (\neg\mathsf{InRoom}(\mathsf{kitchen}, \mathit{do}(\mathsf{pickup}(b_1), s)) \\
\qquad \wedge\ \mathit{Poss}(\mathsf{goto}(\mathsf{kitchen}), \mathit{do}(\mathsf{pickup}(b_1), s)) \\
\qquad \wedge\ s' = \mathit{do}(\mathsf{goto}(\mathsf{kitchen}), \mathit{do}(\mathsf{pickup}(b_1), s)))\ ]
\end{array}$$

In general, we define the formula $\mathbf{Do}$ recursively on the structure of the complex action as follows:

1. For any primitive action A, we have

$$\mathbf{Do}(A, s, s') \stackrel{def}{=} \mathit{Poss}(A, s) \wedge s' = \mathit{do}(A, s).$$

2. For the sequential composition of complex actions A and B, $[A\ ;\ B]$, we have

$$\mathbf{Do}([A\ ;\ B], s, s') \stackrel{def}{=} \exists s''.\ \mathbf{Do}(A, s, s'') \wedge \mathbf{Do}(B, s'', s').$$

3. For a conditional involving a test ϕ[5] of the form [*if* ϕ *then* A *else* B], we have

$$\begin{array}{l}
\mathbf{Do}([\textit{if}\ \phi\ \textit{then}\ A\ \textit{else}\ B], s, s') \stackrel{def}{=} \\
\qquad [\phi(s)\ \wedge\ \mathbf{Do}(A, s, s')]\ \vee\ [\neg\phi(s) \wedge \mathbf{Do}(B, s, s')].
\end{array}$$

[5] If $\phi(s)$ is a formula of the situation calculus with a free variable s, then ϕ is that formula with the situation argument suppressed. For example, in a complex action we would use the test $\mathsf{Broken}(x)$ instead of $\mathsf{Broken}(x, s)$.

4. For a test action, $[\phi?]$, determining if a condition ϕ currently holds, we have

$$\mathbf{Do}([\phi?], s, s') \stackrel{def}{=} \phi(s) \wedge s' = s.$$

5. For a nondeterministic branch to action A or action B, $[A \mid B]$, we have

$$\mathbf{Do}([A \mid B], s, s') \stackrel{def}{=} \mathbf{Do}(A, s, s') \vee \mathbf{Do}(B, s, s').$$

6. For a nondeterministic choice of a value for variable x, $[\pi x.A]$, we have

$$\mathbf{Do}([\pi x.A], s, s') \stackrel{def}{=} \exists x.\mathbf{Do}(A, s, s').$$

7. For an iteration of the form [*while* ϕ *do* A], we have[6]

$$\mathbf{Do}([\textit{while}\ \phi\ \textit{do}\ A], s, s') \stackrel{def}{=} \forall P\{\cdots \supset P(s, s')\}$$

where the ellipsis is an abbreviation for the conjunction of

$$\begin{array}{l}
\forall s_1.\ \neg\phi(s_1) \supset P(s_1, s_1) \\
\forall s_1, s_2, s_3.\ \phi(s_1) \wedge \mathbf{Do}(A, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3)
\end{array}$$

Similar rules can be given for recursive procedures, and even constructs involving concurrency and interrupts. The main point is that what it means to perform these complex actions can be fully specified in the language of the situation calculus. What we are giving, in effect, is a purely logical semantics for many of the constructs of traditional programming languages.

14.3.2 GOLOG

What we end up with, then, is a programming language, called GOLOG, that generalizes conventional imperative programming languages.[7] It includes the usual imperative constructs (sequence, iteration, *etc.*), as well as nondeterminism and other features. The main difference, however, is that the primitive statements of GOLOG are not operations on internal states, like assignment statements or pointer updates, but rather primitive actions in the world, such as picking up a block. Moreover,

[6] The rule for iteration involves *second-order quantification*: the P in this formula is a quantified predicate variable. The definition says that an iteration takes you from s to s' iff the smallest relation P satisfying certain conditions does so. The details are not of concern here.

[7] The name comes from "*Algol in logic*," after one of the original and influential programming languages.

what these primitive actions are supposed to do is not fixed in advance by the language designer, but is specified by the user separately by precondition and successor state axioms.

Given that the primitive actions are not fixed in advance or executed internally, it is not immediately obvious what it should mean to execute a GOLOG program A. There are two steps:

1. find a sequence of primitive actions $\vec{a}$ such that $\mathbf{Do}(A, S_0, do(\vec{a}, S_0))$ is entailed by the KB;
2. pass the sequence of actions $\vec{a}$ to a robot or simulator for actual execution in the world.

In other words, to execute a program we must first find a sequence of actions that would take us to a legal terminating situation for the program starting in the initial situation S_0, and then run that sequence.

Note that to find such a sequence, it will be necessary to reason using the given precondition and effect axioms, performing projection and legality testing. For example, suppose we have the program

$$[A \ ; \ \textit{if } \mathsf{Holding}(x) \textit{ then } B \textit{ else } C].$$

To decide between B and C, we need to determine whether or not $\mathsf{Holding}(x, s)$ would be true in the situation that results from performing action A.

14.3.3 An example

To see how this would work, consider a simple example in a robotics domain involving three primitive actions, $\mathsf{pickup}(x)$ (picking up a block), $\mathsf{putonfloor}(x)$ (putting a block on the floor), and $\mathsf{putontable}(x)$ (putting a block on the table), and three fluents $\mathsf{Holding}(x, s)$ (the robot is holding a block), $\mathsf{OnFloor}(x, s)$ (a block is on the floor), and $\mathsf{OnTable}(x, s)$ (a block is on the table).

The precondition axioms are the following:

- $Poss(\mathsf{pickup}(x), s) \equiv \forall z.\neg\mathsf{Holding}(z, s)$;
- $Poss(\mathsf{putonfloor}(x), s) \equiv \mathsf{Holding}(x, s)$;
- $Poss(\mathsf{putontable}(x), s) \equiv \mathsf{Holding}(x, s)$.

The successor state axioms are the following:

- $\mathsf{Holding}(x, do(a, s)) \equiv a = \mathsf{pickup}(x) \vee$
  $\mathsf{Holding}(x, s) \wedge a \neq \mathsf{putonfloor}(x) \wedge a \neq \mathsf{putontable}(x)$;
- $\mathsf{OnFloor}(x, do(a, s)) \equiv a = \mathsf{putonfloor}(x) \vee$
  $\mathsf{OnFloor}(x, s) \wedge a \neq \mathsf{pickup}(x)$;
- $\mathsf{OnTable}(x, do(a, s)) \equiv a = \mathsf{putontable}(x) \vee$
  $\mathsf{OnTable}(x, s) \wedge a \neq \mathsf{pickup}(x)$.

We might also have the following facts about the initial situation:

- $\neg\mathsf{Holding}(x, S_0)$;
- $\mathsf{OnTable}(x, S_0) \equiv (x = b_1) \vee (x = b_2)$.

So initially, the robot is not holding anything, and b_1 and b_2 are the only blocks on the table. Finally, we can consider two complex actions, removing a block, and clearing the table:

- *proc* $\mathsf{RemoveBlock}(x)$: $[\mathsf{pickup}(x) \ ; \ \mathsf{putonfloor}(x)]$;
- *proc* ClearTable : *while* $\exists x.\mathsf{OnTable}(x)$ *do*
  $\pi x[\mathsf{OnTable}(x)? \ ; \ \mathsf{RemoveBlock}(x)]$.

This completes the specification of the example.

To execute the GOLOG program ClearTable, it is necessary to first find an appropriate terminating situation, $do(\vec{a}, S_0)$, which determines the actions $\vec{a}$ to perform. To find this situation, we can use Resolution theorem-proving with answer extraction for the query

$$\mathrm{KB} \models \exists s.\ \mathbf{Do}(\mathsf{ClearTable}, S_0, s).$$

We omit the details of this derivation, but the result will yield a value for s like

$$s = do(\mathsf{putonfloor}(b_2), do(\mathsf{pickup}(b_2), do(\mathsf{putonfloor}(b_1), do(\mathsf{pickup}(b_1), S_0))))$$

from which the desired sequence starting from S_0 is

$$\langle \mathsf{pickup}(b_1), \mathsf{putonfloor}(b_1), \mathsf{pickup}(b_2), \mathsf{putonfloor}(b_2) \rangle.$$

In a more general setting, an answer predicate could be necessary. In fact, in some cases, it may not be possible to obtain a definite sequence of actions. This happens, for example, if what is known about the initial situation is that either block b_1 or block b_2 is on the table.

Observe that if what is known about the initial situation and the actions can be expressed as Horn clauses, the evaluation of GOLOG programs can be done directly in PROLOG. Instead of expanding $\mathit{Do}(A, s, s')$ into a long formula of the situation calculus and then using Resolution, we write PROLOG clauses such as

```
do(A,S1,S2) :-                          /* for primitive actions */
    prim_action(A), poss(A,S1), S2=do(A,S1).
do(seq(A,B),S1,S2) :-                   /* for sequences */
    do(A,S1,S3), do(B,S3,S2).
do(while(F,A),S1,S2) :-                 /* for while loops (test false) */
    not holds(F,S1), S2=S1.
do(while(F,A),S1,S2) :-                 /* for while loops (test true) */
    holds(F,S1), do(seq(A,while(F,A)),S1,S2).
```

and so on. Then the PROLOG goal

```
?- do(clear_table,s0,S).
```

would return the binding for the final situation.

This idea of using Resolution with answer extraction to derive a sequence of actions to perform will be taken up again in the next chapter on planning. When the problem can be reduced to PROLOG, we get a convenient and efficient way of generating a sequence of actions. This has proven to be an effective method of providing high-level control for a robot.

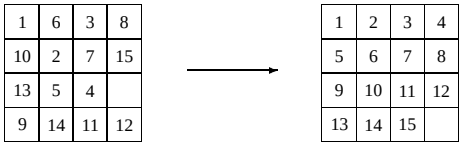
14.4 Bibliographic notes

14.5 Exercises

In the exercises below, and in the follow-up exercises of Chapter 15, we consider three application domains where we would like to be able to reason about action and change:

Pots of water: Consider a world with pots that may contain water. There is a single fluent, Contains, where Contains(p, w, s) is intended to say that a pot p contains w litres of water in situation s. There are only two possible actions, which can always be executed: empty(p) which discards all the water contained in the pot p, and transfer(p, p'), which pours as much water as possible without spilling from pot p to p', with no change when $p = p'$. To simplify

Figure 14.1: The 15-puzzle



the formalization, we assume that the usual arithmetic constants, functions and predicates are also available. (You may assume that axioms for these have already been provided or built-in.)

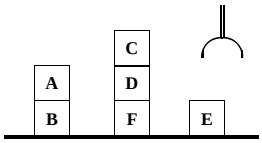
15 puzzle: The 15-puzzle consists of 15 consecutively numbered tiles located in a 4×4 grid. The object of the puzzle is to move the tiles within the grid so that each tile ends up at its correct location, as shown in Figure 14.1. The domain consists of *locations*, numbered 1 to 16, *tiles*, numbered 1 to 15, and of course, actions and situations. There will be a single action move(t, l) whose effect is to move tile t to location l, when possible. We will also assume a single fluent, which is a function loc, where loc(t, s) refers to the location of tile t in situation s. The only other non-logical terms we will use is the situation calculus predicate *Poss* and, to simplify the formalization, a predicate Adjacent(l_1, l_2) which holds when location l_1 is one move away from location l_2. For example, location 5 is adjacent only to locations 1, 6, and 9. (You may assume that axioms for Adjacent have already been provided.)

Note that in the text we concentrated on fluents that were predicates. Here we have a fluent that is a function. Instead of writing Loc(t, l, s), you will be writing loc$(t, s) = l$.

Blocks world: Imagine that we have a collection of blocks on a table, and that we have a robot arm that is capable of picking up blocks and putting them elsewhere as shown in Figure 14.2

We assume that the robot arm can hold at most one block at a time. We also assume that the robot can only pick up a block if there is no other

Figure 14.2: The blocks word



block on top of it. Finally, we assume that a block can only support or be supported by at most one other block, but that the table surface is large enough that all blocks can be directly on the table. There are only two actions available: puton(x, y) which picks up block x and moves it onto block y, and putonTable(x) which moves block x onto the table. Similarly, we have only two fluents: On(x, y, s) which holds when block x is on block y, and OnTable(x, s) which holds when block x is on the table.

For each application, the questions are the same:

1. Write the precondition axioms for the actions.
2. Write the effect axioms for the actions.
3. Show how successor state axioms for the fluents would be derived from these effect axioms. Argue that the successor state axioms are not logically entailed by the effect axioms, by briefly describing an interpretation where the effect axioms are satisfied but the successor state ones are not.
4. Show how frame axioms are logically entailed by the successor state axioms.

Chapter 15

Planning

When we explored reasoning about action in Chapter 14, we considered how a system could figure out what to do, given a complex nondeterministic action to execute, by using what it knows about the world and the primitive actions at its disposal. In this chapter, we consider a related but more fundamental reasoning problem: how to figure out what to do to make some arbitrary condition true. This type of reasoning is usually called *planning*. The condition that we want to achieve is called the *goal*, and the sequence of actions we seek that will make the goal true is called a *plan*.

Planning is one of the most useful ways that an intelligent agent can take advantage of the knowledge it has and its ability to reason about actions and their consequences. If we think of Artificial Intelligence as the study of intelligent behavior achieved through computational means, then planning is central to this study since it is concerned precisely with generating intelligent behavior, and in particular, with using what is known to find a course of action that will achieve some goal. The knowledge in this case involves information about the world, about how actions affect the world, about potentially complex sequences of events, and about interacting actions and entities, including other agents.

In the real world, because our actions are not totally guaranteed to have certain effects, and because we simply cannot know everything there is to know about a situation, planning is usually an uncertain enterprise, and it requires attention to many of the issues we have covered in earlier chapters, such as defaults and reasoning under uncertainty. Moreover, planning in the real world involves trying to determine what future states of the world will be like, but also observing the world as plans are being executed, and replanning as necessary. Nonetheless, the basic capabilities needed to begin considering planning are already available to us.

15.1 Planning in the situation calculus

Given its appropriateness for representing dynamically changing worlds, the situation calculus is an obvious candidate to support planning. We can use it to represent what is known about the current state of the world and the available actions.

The planning task can be formulated in the language of the situation calculus as follows:

> Given a formula, $Goal(s)$, of the situation calculus with a single free variable s, find a sequence of actions $\vec{a} = \langle a_1, \ldots a_n \rangle$, such that
>
> $$\text{KB} \models Goal(do(\vec{a}, S_0)) \wedge Legal(do(\vec{a}, S_0))$$
>
> where $do(\vec{a}, S_0)$ abbreviates $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)\ldots))$, and $Legal(do(\vec{a}, S_0))$ abbreviates $\bigwedge_{i=1}^{n} Poss(a_i, do(\langle a_1, \ldots a_{i-1} \rangle, S_0))$.

In other words, given a goal formula, we wish to find a sequence of actions such that it follows from what is known that

1. the goal formula will hold in the situation that results from executing the actions in sequence starting in the initial state, and
2. it is possible to execute each action in the appropriate situation (that is, each action's preconditions are satisfied).

Note that this definition says nothing about the structure of the KB—for example, whether or not it represents complete knowledge about the initial situation.

Having formulated the task this way, to do the planning, we can use Resolution theorem-proving with answer extraction for the following query:

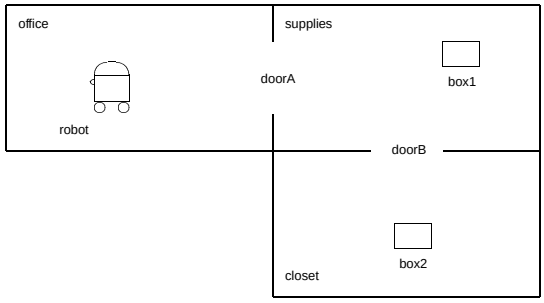
$$\text{KB} \models \exists s.\, Goal(s) \wedge Legal(s).$$

As with the execution of complex actions in Chapter 14, if the extracted answer is of the form $do(\vec{a}, S_0)$, then $\vec{a}$ is a correct plan. But as we will see in Section 15.4.2, there can be cases where the existential is entailed, but where the planning task is impossible because of incomplete knowledge. In other words, the goal can be achieved, but we can't find a specific way that is guaranteed to achieve it.

15.1.1 An example

Let us examine how this version of planning might work in the simple world depicted in Figure 15.1. A robot can roll from room to room, possibly pushing objects through doorways between the rooms. In such a world, there are two actions:

Figure 15.1: A simple robot world



pushThru(x, d, r_1, r_2), in which the robot pushes object x through doorway d from room r_1 to r_2, and goThru(d, r_1, r_2), in which the robot rolls through doorway d from room r_1 to r_2. To be able to execute either action, d must be the doorway connecting r_1 and r_2, and the robot must be located in r_1. After successfully completing either action, the robot ends up in room r_2. In addition, for the action pushThru, the object x must be located initially in room r_1, and will also end up in room r_2.

We can formalize these properties of the world in the situation calculus using the following two precondition axioms:

$$\textit{Poss}(\text{goThru}(d, r_1, r_2), s) \equiv \text{Connected}(d, r_1, r_2) \wedge \text{InRoom}(\text{robot}, r_1, s);$$

$$\textit{Poss}(\text{pushThru}(x, d, r_1, r_2), s) \equiv \text{Connected}(d, r_1, r_2) \wedge \text{InRoom}(\text{robot}, r_1, s) \wedge \text{InRoom}(x, r_1, s).$$

In this formulation, we use a single fluent, InRoom(x, r, s), with the following successor state axiom:

$$\text{InRoom}(x, r, \textit{do}(a, s)) \equiv \Pi(r) \vee (\text{InRoom}(x, r, s) \wedge \neg\exists r'.\, \Pi(r')),$$

where $\Pi(r)$ is the formula

$$\begin{array}{ll} & x = \text{robot} \wedge \exists d \exists r_1.\, a = \text{goThru}(d, r_1, r) \\ \vee & x = \text{robot} \wedge \exists d \exists r_1 \exists y.\, a = \text{pushThru}(y, d, r_1, r) \\ \vee & \exists d \exists r_1.\, a = \text{pushThru}(x, d, r_1, r). \end{array}$$

In other words, the robot is in room r after an action if that action was either a goThru or a pushThru to r, or the robot was already in r, and the action was not a goThru or a pushThru to some other r'. For any other object, the object is in room r after an action if that action was a pushThru to r for that object, or the object was already in r, and the action was not a pushThru to some other r' for that object.

Our KB should also contain facts about the specific initial situation depicted in Figure 15.1: there are three rooms, an office, a supply room, and a closet, two doors, two boxes, and the robot, with their locations as depicted. Finally, the KB needs to state that the robot and boxes are distinct objects and, for the solution to the frame problem presented in Chapter 14, that goThru and pushThru are distinct actions.

15.1.2 Using Resolution

Now suppose that we want to get some box into the office—that is, the goal we would like to achieve is

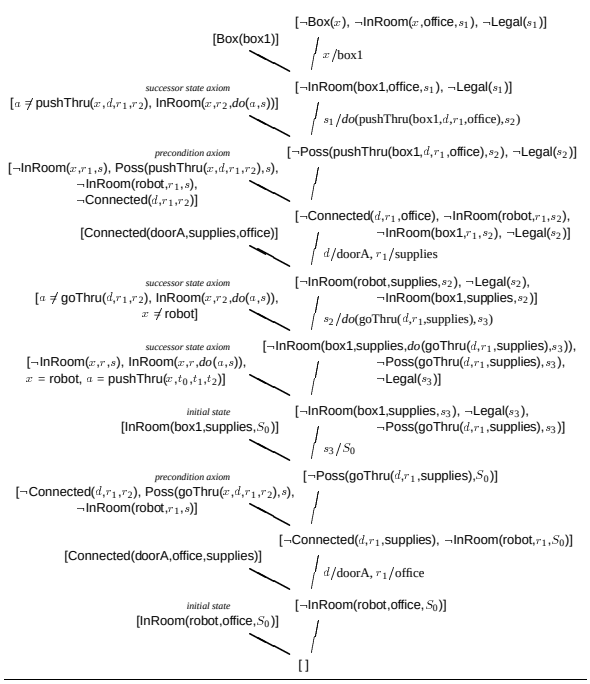
$$\exists x.\, \text{Box}(x) \wedge \text{InRoom}(x, \text{office}, s).$$

To use Resolution to find a plan to achieve this goal, we must first convert the KB to CNF. Most of this is straightforward, except for the successor state axiom, which expands to a set of clauses that includes the following (for one direction of the $\equiv$ formula only):

$$[x \neq \text{robot}, a \neq \text{goThru}(d, r_1, r_2), \text{InRoom}(x, r_2, \textit{do}(a, s))]$$
$$[x \neq \text{robot}, a \neq \text{pushThru}(y, d, r_1, r_2), \text{InRoom}(x, r_2, \textit{do}(a, s))]$$
$$[a \neq \text{pushThru}(x, d, r_1, r_2), \text{InRoom}(x, r_2, \textit{do}(a, s))]$$
$$[\neg\text{InRoom}(x, r, s), x = \text{robot}, a = \text{pushThru}(x, t_0, t_1, t_2), \text{InRoom}(x, r, \textit{do}(a, s))]$$
$$[\neg\text{InRoom}(x, r, s), a = \text{goThru}(t_3, t_4, t_5), a = \text{pushThru}(t_6, t_7, t_8, t_9), \text{InRoom}(x, r, \textit{do}(a, s))].$$

The t_i here are Skolem terms of the form $f_i(x, r, a, s)$ arising from the existentials in the subformula $\Pi(r)$.

Figure 15.2: Planning using Resolution



The Resolution proof tree for this planning problem is sketched in Figure 15.2. The formulas on the left are taken from the KB, while those on the right start with the negation of the formula to be proved:

$$\exists s_1 \exists x.\ \mathsf{Box}(x) \wedge \mathsf{InRoom}(x, \mathsf{office}, s_1) \wedge \mathit{Legal}(s_1).$$

Notice that whenever a *Legal* literal is derived, it is expanded to a clause containing *Poss*, or to the empty clause in the case of $\neg \mathit{Legal}(S_0)$. For example, in the second step of the derivation, s_1 is replaced by a term of the form $\mathit{do}(\ldots, s_2)$, and so $\neg \mathit{Legal}(s_1)$ expands to a clause containing $\neg \mathit{Poss}(\ldots, s_2)$ and $\neg \mathit{Legal}(s_2)$. Also observe that the successor state axioms in the KB use equality, which would require some additional machinery (as explained in Chapter 4), and which we have omitted from the diagram here for simplicity.

To keep the diagram simple, we have also not included an answer predicate in this derivation. Looking at the bindings on the right side, it can be seen that the correct substitution for s_1 is

$$\mathit{do}(\mathsf{pushThru}(\mathsf{box1}, \mathsf{doorA}, \mathsf{supplies}, \mathsf{office}), \\ \mathit{do}(\mathsf{goThru}(\mathsf{doorA}, \mathsf{office}, \mathsf{supplies}), S_0)).$$

and so the plan is to first perform the goThru action and then the pushThru one.

All but one of the facts in this derivation (including a definition of *Legal*) can be expressed as Horn clauses. The final use of the successor state axiom has two positive equality literals. However, by using negation as failure to deal with the inequalities, we can use a PROLOG program directly to generate a plan, as shown in Figure 15.3. The goal would be

```
?- box(X), inRoom(X,office,S), legal(S).
```

and result of the computation would then be

```
X = box1
S = do(pushThru(box1,doorA,supplies,office),
       do(goThru(doorA,office,supplies),s0))
```

as it was above. Using PROLOG in this way is very delicate, however. A small change in the ordering of clauses or literals can easily cause the depth-first search strategy to go down an infinite branch.

In fact, more generally, using Resolution theorem-proving over the situation calculus for planning is rarely practical for two principal reasons. First of all, we are required to explicitly draw conclusions about what is not changed by doing actions. We saw this in the derivation above (in the final use of the successor state axiom), where we concluded that the robot moving from the office to the supply room did not change the location of the box (and so the box was still ready to be pushed into the office). In this case, there was only one action and one box to worry about; in a larger setting, we may have to reason about the properties of many objects remaining unaffected after the performance of many actions.

Figure 15.3: Planning using Prolog

```
inRoom(robot,office,s0).
box(box1).  inRoom(box1,supplies,s0).
box(box2).  inRoom(box2,closet,s0).

connected(doorA,office,supplies).
connected(doorA,supplies,office).
connected(doorB,closet,supplies).
connected(doorB,supplies,closet).

poss(goThru(D,R1,R2),S) :-
   connected(D,R1,R2), inRoom(robot,R1,S).
poss(pushThru(X,D,R1,R2),S) :-
   connected(D,R1,R2), inRoom(robot,R1,S),
   inRoom(X,R1,S).

inRoom(X,R2,do(A,S)) :-
   X=robot, A=goThru(D,R1,R2).
inRoom(X,R2,do(A,S)) :-
   X=robot, A=pushThru(Y,D,R1,R2).
inRoom(X,R2,do(A,S)) :-
   A=pushThru(X,D,R1,R2).
inRoom(X,R,do(A,S)) :- inRoom(X,R,S),
   not (X=robot),
   not (A=pushThru(X,T0,T1,T2)).
inRoom(X,R,do(A,S)) :- inRoom(X,R,S),
   not (A=goThru(T3,T4,T5)),
   not (A=pushThru(T6,T7,T8,T9)).

legal(s0).
legal(do(A,S)) :- poss(A,S), legal(S).
```

Secondly, and more seriously, the search for a sequence of actions using Resolution (or the PROLOG variant) is completely unstructured. Notice, for example, that in the derivation above, the first important choice that was made was to bind the x to box1. If your goal is to get some box into the office, it is silly to first decide on a box and then search for a sequence of actions that will work for that box. Much better would be to decide on the box opportunistically based on the current situation and what else needs doing. In some cases the search should work backwards from the goal; in others, it should work forward from the current state. Of course, all of this search should be quite separate from the search that is needed to reason about what does or does not hold in any given state.

In the next section, we deal with the first of these issues. We deal with searching for a plan effectively in Section 15.3.

15.2 The STRIPS Representation

STRIPS is an alternative representation to the pure situation calculus for planning. It derives from work on a mobile robot (called "Shakey") at SRI International in the 1960's. In STRIPS, we assume that the world we are trying to deal with satisfies the following:

- only one action can occur at a time;
- actions are effectively instantaneous;
- nothing changes except as the result of planned actions.

In this context, the above has been called the "STRIPS assumption," but it clearly applies just as well to our version of the situation calculus. What really distinguishes STRIPS from the situation calculus is that knowledge about the initial state of the world is required to be complete, and knowledge about the effects and non-effects of actions is required to be in a specific form. In what follows, we use a very simple version of the representation, although many of the advantages we claim for it hold more generally.

In STRIPS, we do not represent histories of the world like we do in the situation calculus, but rather we deal with a single world state at a time. The world state is represented by what is called a *world model*, which is a set of ground atomic formulas, similar to a database of facts in the PLANNER system of Chapter 6, and the working memory of a production system of Chapter 7. These facts can be thought of as ground fluents (with the situation argument suppressed) under closed-world, unique-name, and domain-closure assumptions (as in Chapter 11). For the example depicted in Figure 15.1, we would have the following initial world model, DB_0:

InRoom(box1,supplies)	Box(box1)
InRoom(box2,closet)	Box(box2)
InRoom(robot,office)	
Connected(doorA,office,supplies)	Connected(doorA,supplies,office)
Connected(doorB,closet,supplies)	Connected(doorA,supplies,closet)

In this case there is no need to distinguish between a fluent (like InRoom) and a predicate that is unaffected by any action (like Box).

Further, in STRIPS, actions are not represented explicitly as part of the world model, which means that we cannot reason about them directly. Instead, actions are thought of as *operators*, which syntactically transform world models. An operator takes the world model database for some state, and transforms it into a database representing the successor state. The main benefit of this way of representing and reasoning about plans is that it avoids frame axioms: an operator will change what it needs to in the database, and thereby leave the rest unaffected.

STRIPS operators are specified by pre- and postconditions. The preconditions are sets of atomic formulas of the language that need to hold before the operator can apply. The postconditions come in two parts: a *delete list*, which is a set of atomic formulas to be removed from the database; and an *add list*, which is a set of atomic formulas to be added to the database. The delete list represents properties of the world state that no longer hold after the operator is applied, and the add list represents new properties of the world state that will hold after the operator is applied. For the example above, we would have the following two operators:

pushThru(x, d, r_1, r_2)
Precondition: InRoom(robot, r_1), InRoom(x, r_1), Connected(d, r_1, r_2)
Delete list: InRoom(robot, r_1), InRoom(x, r_1)
Add list: InRoom(robot, r_2), InRoom(x, r_2)

goThru(d, r_1, r_2)
Precondition: InRoom(robot, r_1), Connected(d, r_1, r_2)
Delete list: InRoom(robot, r_1)
Add list: InRoom(robot, r_2)

Note that the arguments of operators are variables that can appear in the the pre- and postcondition formulas.

A STRIPS problem, then, is represented by an initial world model database, a set of operators, and a goal formula. A solution to the problem is a set of operators that can be applied in sequence starting with the initial world model without violating any of the preconditions, and which results in a world model that satisfies the goal formula.

More precisely, a STRIPS problem is characterized by $\langle \mathrm{DB}_0, \mathit{Operators}, \mathit{Goal} \rangle$ where DB_0 is a list of ground atoms, *Goal* is a list of atoms (whose free variables are understood existentially), and *Operators* is a list of operators of the form $\langle \mathit{Act}, \mathit{Pre}, \mathit{Add}, \mathit{Del} \rangle$ where *Act* is the name of the operator, and *Pre*, *Add*, and *Del* are lists of atoms. A solution is a sequence

$$\langle \mathit{Act}_1\theta_1, \ldots, \mathit{Act}_n\theta_n \rangle$$

Figure 15.4: A depth-first progressive planner

Input: a world model and a goal formula
Output: a plan or fail

ProgPlan[DB,*Goal*] =
  If *Goal* $\subseteq$ DB then return the empty plan
  For each operator $\langle \mathit{Act}, \mathit{Pre}, \mathit{Add}, \mathit{Del} \rangle$ such that *Pre* $\subseteq$ DB do
    Let $\mathrm{DB}' = \mathrm{DB} + \mathit{Add} - \mathit{Del}$
    Let *Plan* = ProgPlan[$\mathrm{DB}', \mathit{Goal}$]
    If *Plan* $\neq$ fail then return $\mathit{Act} \cdot \mathit{Plan}$
  end for
  Return fail

where Act_i is the name of an operator in the list (with Pre_i, Add_i, and Del_i as the other corresponding components) and θ_i is a substitution of constants for the variables in that operator, and where the sequence satisfies the following:

- for all $1 \leq i \leq n$, $\mathrm{DB}_i = \mathrm{DB}_{i-1} + \mathit{Add}_i\theta_i - \mathit{Del}_i\theta_i$;
- for all $1 \leq i \leq n$, $\mathit{Pre}_i\theta_i \subseteq \mathrm{DB}_{i-1}$;
- for some θ, $\mathit{Goal}\theta \subseteq \mathrm{DB}_n$.

The + and $-$ in this definition refer to the union and difference of lists respectively.

15.2.1 Progressive planning

The characterization of a solution to the STRIPS planning problem above immediately suggests the planning procedure shown in Figure 15.4. For simplicity, we have left out the details concerning the substitutions of variables. This type of planner is called a *progressive* planner, since it works by progressing the initial world model forward until we obtain a world model that satisfies the goal formula.

Consider once again the planning problem in Figure 15.1. If called with the initial world model above (DB_0), and goal

$$\mathrm{Box}(x),\ \mathrm{InRoom}(x, \mathrm{office}),$$

the progressive planner would first confirm that the goal is not yet satisfied, and then within the loop, eventually get to the operator goThru(doorA,office,supplies)

whose precondition is satisfied in the DB. It then would call itself recursively with the following progressed world model:

InRoom(box1,supplies)	Box(box1)
InRoom(box2,closet)	Box(box2)
InRoom(robot,supplies)	
Connected(doorA,office,supplies)	Connected(doorA,supplies,office)
Connected(doorB,closet,supplies)	Connected(doorA,supplies,closet)

The goal is still not satisfied, and the procedure then continues and gets to the operator pushThru(box1,doorA,supplies,office) whose precondition is satisfied in the progressed DB. It would then call itself recursively with a new world model:

InRoom(box1,office)	Box(box1)
InRoom(box2,closet)	Box(box2)
InRoom(robot,office)	
Connected(doorA,office,supplies)	Connected(doorA,supplies,office)
Connected(doorB,closet,supplies)	Connected(doorA,supplies,closet)

At this point, the goal formula is satisfied, and the procedure unwinds successfully and produces the expected plan.

15.2.2 Regressive planning

In some applications, it may be advantageous to use a planner that works backwards from the goal rather than forward from the initial state. The process of working backwards, repeatedly simplifying the goal until we obtain one that is satisfied in the initial state is called *goal regression*. A regressive planner is shown in Figure 15.5. In this case, the first operator we consider is the last one in the plan. This operator obviously must not delete any atomic formula that appears in the goal. Furthermore, to be able to use this operator, we must ensure that its preconditions will be satisfied; so they become part of the next goal. However, the formulas in the add list of the operator we are considering will be handled by that operator, so they can be removed from the goal as we regress it.

If called with the initial world model from Figure 15.1 and goal

$$\mathsf{Box}(x),\ \mathsf{InRoom}(x, \mathsf{office}),$$

the regressive planner would first confirm that the goal is not yet satisfied, and then within the loop, eventually get to pushThru(box1,doorA,supplies,office) whose delete list does not intersect with the goal.[1] It then would call itself recursively with

[1] As before, we are omitting details about variable bindings. A more realistic version would certainly leave the x in the goal unbound at this point, for example.

Figure 15.5: A depth-first regressive planner

Input: a world model and a goal formula
Output: a plan, or fail

RegrPlan[DB,*Goal*] =
  If $Goal \subseteq$ DB then return the empty plan
  For each operator $\langle Act, Pre, Add, Del\rangle$ such that $Del \cap Goal = \{\}$ do
    Let $Goal' = Goal + Pre - Add$
    Let $Plan$ = RegrPlan[DB, $Goal'$]
    If $Plan \neq$ fail then return $Plan \cdot Act$
  end for
  Return fail

the following regressed goal:

$$\mathsf{Box}(\mathsf{box1}),\ \mathsf{InRoom}(\mathsf{robot}, \mathsf{supplies}),\ \mathsf{InRoom}(\mathsf{box1}, \mathsf{supplies}),\ \mathsf{Connected}(\mathsf{doorA}, \mathsf{supplies}, \mathsf{office}).$$

The goal is still not satisfied in the initial world model, so the procedure continues and within the loop, eventually gets to the operator goThru(doorA,office,supplies) whose delete list does not intersect with the current goal. It would then call itself recursively with a new regressed goal:

$$\mathsf{Box}(\mathsf{box1}),\ \mathsf{InRoom}(\mathsf{robot}, \mathsf{office}),\ \mathsf{InRoom}(\mathsf{box1}, \mathsf{supplies}),\ \mathsf{Connected}(\mathsf{doorA}, \mathsf{supplies}, \mathsf{office}),\ \mathsf{Connected}(\mathsf{doorA}, \mathsf{office}, \mathsf{supplies}).$$

At this point, the goal formula is satisfied in the initial world model, and the procedure unwinds successfully and produces the expected plan.

15.3 Planning as a reasoning task

While the two planners above (or their breadth-first variants) work much better in practice than the Resolution-based planner considered earlier, neither of them works very well on large problems. This is not too surprising since it can be shown that the planning task is NP-hard, even for the simple version of STRIPS we have considered, and even when the STRIPS operators have no variables. It is therefore

extremely unlikely that there is *any* procedure that will work well in all cases, as this would immediately lead to a corresponding procedure for satisfiability.[2]

As with deductive reasoning, there are essentially two options we can consider: we can do our best to make the search as effective as possible, especially by avoiding redundancy in the search, or we can make the planning problem easier by allowing the user to provide control information.

15.3.1 Avoiding redundant search

One major source of redundancy is the fact that actions in a plan tend to be independent and can be performed in different orders. If the goal is to get both box1 and box2 into the office, we can push box1 first or push box2 first. The problem is that when searching for a sequence of actions (either progressing a world model or regressing a goal), we consider totally ordered sequence of actions. Before we can rule out a collection of actions as inappropriate for some goal, we end up considering many permutations of those same actions.

To deal with this issue, let us consider a new type of plan, which is a finite set of actions that are only partially ordered. Because such a plan is not a linear sequence of actions, it is sometimes called a *nonlinear* plan. In searching for such a plan, we order one action before another only if we are required to do so. For getting the two boxes into the office, for example, we would want a plan with two parallel branches, one for each box. Within each branch, however, the moving actions(s) of the robot to the appropriate room would need to occur strictly before the corresponding pushing action(s).

To generate this type of plan, a different sort of planner, called a *partial-order planner*, is often used. In a partial order planner, we start with an incomplete plan, consisting of the initial world model at one end and the goal at the other end. At each step, we insert new actions into the plan, and new constraints on when that action needs to take place relative to the other actions in the plan, until we have filled all the gaps from one end to the other. It is worth noting, however, that the efficacy of this approach to planning is still somewhat controversial because of the amount of extra bookkeeping it appears to require.

A second source of redundancy concerns applying sequence of actions repeatedly. Consider, for example, getting a box into the office. This always involves the same operators: some number of goThru actions followed by a corresponding number of pushThru actions. Furthermore, this sequence as a whole has a fixed

[2] One popular planning method involves encoding the task directly as a satisfiability problem, and using satisfiability procedures to find a plan.

precondition and postcondition that can calculated once and for all from the component operators. The authors of STRIPS considered an approach to the reuse of such sequences of actions, and created a set of macro-operators, or "MACROPS," which were parameterized and abstracted sequences of operators. While adding macro-operators to a planning problem means that a larger number of operators will need to be considered, if they are chosen wisely, the resulting plans can be much shorter. Indeed, many of the practical planning systems work primarily by assembling precompiled plan fragments from a library of macro-operators.

15.3.2 Application-dependent control

Even with careful attention to redundancy in the search, planning remains impractical for many applications. Often the only way to make planning effective is to make the problem easier, for example, by giving the planner explicit guidance on how to search for a solution. We can think of the macro-operators, for example, as *suggesting* to the planner a sequence to use to get a box into a room. But in some cases, we can be more definite. Suppose, for example, we wish to reorganize all of the boxes in a certain distant room. We might tell the planner that it should handle this by *first* planning on getting to the distant room (ignoring any action dealing with the boxes) and *only then* planning on reorganizing the boxes (ignoring any action involving motion to other rooms). As with the procedural control of Chapter 6, constraints of this sort clearly simplify the search by ruling out various sequences of action.

In fact, we can imagine two extreme versions of this guidance. At one extreme, we let the planner search for any sequence of actions, with no constraints; at the other extreme, the guidance we give to a planner would specify a complete sequence of actions, where no search would be required at all. This idea does not require us to use STRIPS, of course, and the situation calculus, augmented with the GOLOG programming language, provides a convenient notation for expressing application-dependent search strategies.

Consider the following highly nondeterministic GOLOG program:

$$\mathit{while}\ \neg \mathit{Goal}\ \ \mathit{do}\ \{\pi a.\ a\}.$$

The body of the loop says that we should pick an action a nondeterministically, and then do a. To execute the entire program, we need to find a sequence of actions corresponding to performing the loop body repeatedly, ending up in a final situation s where $\mathit{Goal}(s)$ is true. But this is no more and no less than the planning task. So using GOLOG, we can represent guidance to a planner at various levels of specificity.

The program above provides no guidance at all; on the other hand, the deterministic program

$$\{ \quad \mathsf{goThru}(\mathsf{doorA}, \mathsf{office}, \mathsf{supplies}) \; ; \\ \mathsf{pushThru}(\mathsf{box1}, \mathsf{doorA}, \mathsf{supplies}, \mathsf{office}) \; \}$$

requires no search at all. In between, however, we would like to provide some application-dependent guidance, leaving a more manageable search problem.

One convenient way to control the search process during planning is by using what is called *forward filtering*. The idea is to modify very slightly the above program so that not every action a whose precondition is satisfied can be selected as the next action to perform in the sequence, but only those actions that also satisfy some application-dependent criterion:

$$\mathit{while}\ \neg \mathit{Goal}\ \ \mathit{do}\ \{\pi a.\, \mathit{Acceptable}(a)?\ ;\ a\}.$$

The intent is that the fluent $\mathit{Acceptable}(a, s)$ should be defined by the user to filter out actions which may be legal but are not useful at this point in the plan. For example, if we want to tell the planner that it first needs to get to the closet and only then consider moving any boxes, we might have the something like the following in the KB:

$$\mathit{Acceptable}(a, s) \equiv \mathsf{InRoom}(\mathsf{robot}, \mathsf{closet}, s) \wedge \mathsf{BlockAction}(a) \\ \vee \neg \mathsf{InRoom}(\mathsf{robot}, \mathsf{closet}, s) \wedge \mathsf{MoveAction}(a),$$

for some suitable BlockAction and MoveAction predicates. Of course, defining *Acceptable* properly for any particular application is not easy, and requires a deep understanding of how to solve planning problems in that application.

We can use the idea of forward filtering to define a complete progressive planner in GOLOG. The procedure DFPlan below is a recursive variant of the loop above that takes as an argument a bound on the length of the action sequence it will consider. It then does a depth-first search for a plan of that length or shorter:

$$\mathit{proc}\ \mathsf{DFPlan}(n): \\ \quad \mathit{Goal}? \mid \{(n > 0)?\ ;\ \pi a\, (\mathit{Acceptable}(a)?\ ;\ a)\ ;\ \mathsf{DFPlan}(n-1)\}$$

Of course, the plan it finds need not be the shortest one that works. To get the shortest plan, it would be necessary to first look for plans of a certain length, and only then look for longer ones:

$$\mathit{proc}\ \mathsf{IDPlan}(n):\ \mathsf{IDPlan}'(0, n)$$

$$\mathit{proc}\ \mathsf{IDPlan}'(m, n):\ \mathsf{DFPlan}(m) \mid \{(m < n)?\ ;\ \mathsf{IDPlan}'(m+1, n)\}$$

The procedure IDPlan does a form of search called *iterative deepening*. It uses depth-first search (that is, DFPlan) at ever larger depths as a way of providing many of the advantages of breadth-first search.

15.4 Beyond the basics

In this final section, we briefly consider a small number of more advanced topics in planning.

15.4.1 Hierarchical planning

The basic mechanisms of planning that we have covered so far, even including attempts to simplify the process with macro-operators, still preserve all detail needed to solve a problem all the way through the process. In reality, attention to too much detail can derail a planner to the point of uselessness. It would be much better, if possible, to first search through an *abstraction space*, where unimportant details were suppressed. Once a solution in the abstraction space were found, then all we would have to do would be to account for the details of the linkup of the steps.

In an attempt to separate levels of abstraction of the problem in the planning process, the STRIPS team invented the ABSTRIPS approach. The details are not important here, but we can note a few of the elements of this approach. First, preconditions in the abstraction space have fewer literals than those in the ground space, thus they should be less taxing on the planner. For example, in the case of pushThru, at the highest level of abstraction, the operator is applicable whenever an object is pushable and a door exists; without those basic conditions, the operator is not even worth considering. At a lower level of abstraction, like the one we used in our earlier example, the robot and object have to be in the same room, which must be connected by a door to the target room. At an even finer-grained level of detail, it would be important to ascertain whether or not the door was open (and attempt to open it if not). But that is really not relevant until we have a plan that involves going through the door with the object. Finally, in the least abstract representation, it would be important to get the robot right next to the object, and both the robot and object right next to the doorway, so that they could move through it.

15.4.2 Conditional planning

In very many applications, there may not be enough information available to plan a full course of action to achieve some goal. For example in our robot domain, imagine that each box has a printed label on it that says either office or closet, and suppose our goal is to get box1 into the room printed on its label. With no further information, the full, advance planning task is impossible since we have no way of knowing where the box should end up. However, we do know that there exists a sequence of actions that will achieve the goal, namely, to go into the supply room,

and push the box either to the office or to the closet. If we were to use Resolution with answer extraction for this example, the existential query would succeed, but we would end up with a clause with two answer literals, corresponding to the two possible sequences of action.

But now imagine that our robot is equipped with a sensor of some sort that tells it whether or not there is a box located in the same room, with a label on it that says office. In this case, we would now like to say that the planning task, or a generalization of it, is possible. The plan that we expect, however, is not a linear sequence of actions, but is tree-structured, based on the outcome of sensors: go to the supply room, and if the sensor indicates the presence of a box labeled office, then push box1 into the office, and otherwise push box1 into the closet. This type of branching plan is a called a *conditional plan*, and a planner that can generate one is called a *conditional planner*.

There are various ways of making this notion precise, but perhaps the simplest is to extend the language of situation calculus so that instead of just having terms S_0 and *do*(a, s) denoting situations, we also have terms of the form *cdo*(p, s), where p is a tree-structured conditional plan of some sort. The situation denoted by this term would depend on the outcome of the sensors involved, which of course would need to be specified. To describe, for example, the sensor mentioned above, we might state something like the following:

$$\begin{aligned}\mathsf{Fires}(\mathsf{sensor1}, s) \;\equiv\; &\exists x \exists r.\, \mathsf{InRoom}(\mathsf{robot}, r, s) \wedge \\ &\mathsf{Box}(x) \wedge \mathsf{InRoom}(x, r, s) \wedge \mathsf{Label}(x, \mathsf{office})\end{aligned}$$

With terms like *cdo*(p, s) in the language, we could once again use Resolution with answer extraction to do planning. How to do conditional planning *efficiently*, on the other hand, is a much more difficult question.

15.4.3 "Even the best-laid plans..."

Situation calculus representations, and especially STRIPS, make many restrictive assumptions. As we discussed in our section on complex actions, there are many aspects of action that bear investigation and may potentially impact the ability of an AI agent to reason appropriately about the world. Among the many issues in real-world planning that are currently under active investigation we find things like simultaneous interacting actions (e.g., lifting a piano, opening a doorlatch where the key must be turned and knob turned at the same time), external events, nondeterministic actions or those with probabilistic outcomes, non-instantaneous actions, non-static predicates, plans that explicitly include time, and reasoning about termination.

An even more fundamental challenge for planning is the suggestion made by some that explicit, symbolic production of formal plans is something to be avoided altogether. This is generally a reaction to the computational complexity of the underlying planning task. Some advocate instead the idea of a more "reactive" system, which observes conditions and just "reacts" by deciding—or looking up—what to do next. This one-step-at-a-time-like process is more robust in the face of unexpected changes in the environment. A reactive system could be implemented with a kind of "universal plan"—a large lookup table (or boolean circuit) that tells you exactly what to do based on conditions. In some cases where they have been tried, reactive systems have had impressive performance on certain low-level problems, like learning to walk; they have even appeared intelligent in their behavior. At the current time, though, it is very unclear how far one can go with such an approach and what its intrinsic limitations are.

15.5 Bibliographic notes

15.6 Exercises

The exercises below are continuations of the exercises from Chapter 14. For each application, we consider a planning problem involving an initial setup and a goal.

Pots of water: Imagine that in the initial situation, we have two pots, a 5-litre one filled with water, and an empty 2-litre one. Our goal is to obtain 1 litre of water in the 2-litre pot.

15 puzzle: Assume that every tile is initially placed in its correct position, except for tile 9 which is in location 13, tile 13 in location 14, tile 14 in location 15, and tile 15 in location 16. The goal, of course, is to get every tile placed correctly.

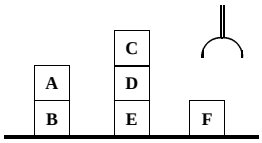
Blocks world: In the initial situation, the blocks are arranged as in Figure 14.2 of Chapter 14. The goal is to get them arranged as in Figure 15.6.

For each application, the questions are the same:

1. Write a sentence of the situation calculus of the form $\exists s.\alpha$ which asserts the existence of the final goal situation.

2. Write a ground situation term e (that is, a term that is either S_0 or of the form *do*(a, e') where a is a ground action term and e' is itself a ground situation term) such that e denotes the desired goal situation.

Figure 15.6: The blocks word goal



3. Explain how you could use Resolution to automatically solve the problem for any initial state: how would you generate the clauses, and assuming the process stops, how would you extract the necessary moves? *(Do not attempt to write down a derivation!)* Explain why you need to use the successor state axioms, and not just effect axioms.

4. Suppose we were interested in formalizing the problem using a STRIPS representation. Decide what the operators should be, and then write the precondition, add list, and delete list for each operator. You may change the language as necessary.

5. Consider the database corresponding to the initial state of the problem. For each STRIPS operator, and each binding of its variables such that the precondition is satisfied, state what the database progressed through this operator would be.

6. Consider the final goal state of the problem. For each STRIPS operator, describe the bindings of its variables for which the operator can be the final action of a plan, and in those cases, what the goal regressed through the operator would be.

7. Without any additional guidance, a very large amount of search is usually required to solve planning problems. There are often, however, application-dependent heuristics that can be used to reduce the amount of search. For example,

   - for the 15-puzzle, we should get the first row and first column of tiles into their correct positions (tiles 1, 2, 3, 4, 5, 9, 13); then recursively solve the remaining 8-puzzle without disturbing these outside tiles;

   - for the blocks world, we should never move a block that is in its *final position*, where a block x is considered to be in its final position iff either (a) x is on the table and x will be on the table in the goal state or (b) x is on another block y, x will be on y in the goal state, and y is also in its final position.

Explain how the complex actions of GOLOG from Chapter 14 can be used to define a more restricted search problem which incorporates heuristics like these. Sketch briefly what the GOLOG program would look like.