Chapter 12 Control Strategies in Deductive Planning

12.1 Introduction

In deductive planning, a planning problem is seen as a deduction problem, i.e., as a problem of proving a theorem in a logical framework. The main difference between classical planning and deductive planning is that in the former each action is specified with a triple of preconditions, positive effects, and negative effects, whereas in the latter actions are specified with logical formulas. In classical planning, state transitions are computed directly by adding positive effects and deleting negative ones. Deductive planning uses deduction for computing state transitions. Plan generation is done by logical deduction, i.e., by applying the inference rules of the logic, rather than by state-space or plan-space search. Given a description of the planning domain as a set of formulas and a goal as a formula, then a solution plan is generated by a proof in the logic.

The consequence is twofold. From the positive side, the main advantage of deductive planning is its expressiveness. Depending on the logic that is chosen, most of the restrictive assumptions of classical planning can be released. For instance, both of the approaches presented in this chapter, situation calculus and dynamic logic, can easily represent infinite state-transition systems, nondeterminism, partial observability, conditional and iterative plans, and extended goals.

From the negative side, planning-specific search techniques have been shown to be more practical than deduction to solve classical planning problems. The main bottleneck in deductive planning is the lack of automatic procedures to generate plans. The reactions to this problem have been along two main lines. One approach is to limit the expressiveness of the logic to special cases for which there are efficient decision procedures, such as the "planning as satisfiability" techniques described in Chapter 7. Another approach, which is the focus of the current chapter, is to keep the logic expressive but allow the user to write domain-dependent control strategies that can reduce the burden of searching for a proof. We focus on two main paradigms.

1. *Plans as programs.* In this approach, the user does not specify a goal and ask the theorem prover to find a plan by deduction. One rather writes programs in a logical language that are *possibly incomplete* specifications of plans. This can reduce significantly the search for a proof because the theorem prover needs to find a proof only for those parts of the plan that are not fully specified. For instance, rather than giving to the planner the goal "the robot should be loaded at a target location," the user can write a program "move the robot to an intermediate location, find a plan for loading the robot and move the robot to the target location."
2. *Tactics.* Tactics are user-defined programs that specify which inference rules should be applied in a deduction process and how inference rules should be combined to generate a proof. This approach is conceptually different than the previous one. The user specifies the goal and then writes a program that guides the search for a proof of that goal. Thus, for instance, if the goal is "the robot should be loaded at a target location," and this is represented as a conjunction of two formulas, "the robot is loaded" and "the robot is at target location," a tactic can specify that the theorem prover should search for two proofs for the two conjuncts and then apply a simple inference rule that proves the conjunction.

Very expressive deductive planning frameworks have been proposed that support either the approach based on plans as programs or the one based on tactics. In this chapter, we describe the "plans as programs" paradigm in Section 12.2 and tactics in Section 12.3. For both of them, we focus on a basic and simple framework mostly for classical planning. This framework has been extended significantly in the literature to deal with different kinds of planning problems (see Section 12.4).

12.2 Situation Calculus

Situation calculus is a first-order language for representing states and actions that change states. Section 12.2.1 introduces the notion of *situation*, Section 12.2.2 shows how actions are represented, and Section 12.2.3 describes planning domains, problems, and solutions in situation calculus. Section 12.2.4 introduces the "plans as programs" paradigm in situation calculus. These sections provide an informal introduction to the topic. See the discussion and historical remarks in Section 12.4 for some main references to formal accounts.

12.2.1 Situations

In classical planning, each state s corresponds to a *different* logical theory whose axioms are the atoms in s (see Chapter 2). Rather than atoms, we could even have axioms that are of a more general form, e.g., any kind of formula in first-order logic (see Section 2.4).

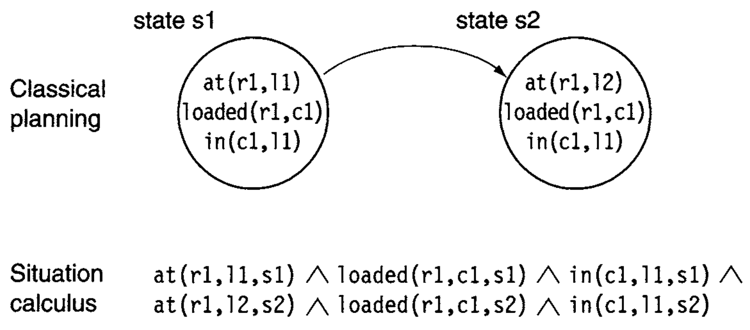


Figure 12.1 Situation calculus versus classical planning.

In situation calculus there is only *one* logical theory, and different states are represented by including the "name" of each state as an extra argument to each atom that is true in the state. For example, to say that s is a state in which at(r1,l1) holds, one would write at(r1,l1,s). The term s is called a *situation*. Figure 12.1 shows the conceptual difference in representation between classical planning and situation calculus.

The language $\mathcal{L}$ of situation calculus is a first-order language that includes terms that are situations. All the expressive power of a first-order language can then be used to describe a planning domain. This means that atoms holding in some states need not be mentioned explicitly; they just need to be entailed by the theory. Thus, for instance, we can state that:

$$\forall r \forall l \forall c \forall s (\text{at}(r, l, s) \wedge \text{loaded}(r, c) \rightarrow \text{in}(c, l, s))$$

Given at(r1,l1,s1) and loaded(r1,c1), then in(c1,l1,s1) is entailed by the theory, and this is true for all the possible instantiations of the variables r, l, c, and s.

Terms and formulas containing a situation are called *fluents*. For example, at(r1,l1,s1) and at(r,l,s), are fluents.

12.2.2 Actions

In classical planning, atoms in different states are described in a restricted first-order language L, and actions are described by operators that are elements of a separate "planning language" based on L. In contrast to classical planning, in situation calculus actions are represented as terms in the same first-order language $\mathcal{L}$ in which states are represented. For instance, the terms move(r1,l1,l2) and move(r,l,l') of $\mathcal{L}$ denote actions. The situation resulting after the execution of an action is represented by means of the special function symbol do. If α is a variable denoting an action, then $\mathsf{do}(\alpha,s)$ is the situation resulting from the execution of α in situation s. For instance, $\mathsf{do}(\mathsf{move}(r,l,l'),s)$ is the situation after executing the action denoted by $\mathsf{move}(r,l,l')$ in situation s.

Action preconditions in situation calculus are represented in the same language $\mathcal{L}$. We use the predicate Poss to represent action preconditions. For instance, the formula

$$\forall r \forall l \forall l' \forall s(\mathsf{Poss}(\mathsf{move}(r,l,l'),s) \leftrightarrow \mathsf{at}(r,l,s)) \tag{12.1}$$

states that action $\mathsf{move}(r,l,l')$ is applicable to a situation s iff the robot is at location l in s ($\mathsf{at}(r,l,s)$). These formulas are called *action precondition axioms*. They are generalizations of action preconditions in classical planning because they can be any first-order formula. For instance, preconditions can be defined with existential quantifiers, disjunctions, etc., as in the following precondition of action transport,

$$\forall c \forall l(\mathsf{Poss}(\mathsf{transport}(c,l),s) \leftrightarrow \exists l' \exists r(\mathsf{adjacent}(l,l') \wedge \mathsf{at}(r,l') \wedge \mathsf{loaded}(r,c))) \tag{12.2}$$

which states that a container can be transported to a location if there is a robot loaded with that container in an adjacent location.

In situation calculus, action effects are described by formulas of $\mathcal{L}$. For instance, the formula

$$\forall r \forall l \forall l' \forall s(\mathsf{Poss}(\mathsf{move}(r,l,l'),s) \rightarrow \mathsf{at}(r,l',\mathsf{do}(\mathsf{move}(r,l,l'),s))) \tag{12.3}$$

states that the effect of moving a robot from l to l' is that the robot is at l'. Formula 12.3 describes a *positive effect*, i.e., the fact that a fluent becomes true. We also have to take into account *negative effects*, i.e., the case where an action causes a fluent to become false. For instance, the formula

$$\forall r \forall l \forall l' \forall s(\mathsf{Poss}(\mathsf{move}(r,l,l'),s) \rightarrow \neg\mathsf{at}(r,l,\mathsf{do}(\mathsf{move}(r,l,l'),s))) \tag{12.4}$$

describes a negative effect. The action effects we have described so far are analogous to the effects represented in classical planning. In situation calculus, however, we can use all the expressiveness of first-order logic to describe effects. For instance, we can also have *conditional effects*, i.e., effects that depend on the situation. For instance, the formula

$$\forall r \forall l \forall l' \forall c \forall s((\mathsf{Poss}(\mathsf{move}(r,l,l'),s) \wedge \mathsf{loaded}(r,c,s)) \rightarrow \mathsf{in}(c,l',\mathsf{do}(\mathsf{move}(r,l,l'),s))) \tag{12.5}$$

states that if the robot is loaded with a container and the robot moves to a destination, then the container is at the destination.

Different than classical planning, situation calculus must take into account the *frame problem*, i.e., the problem of describing what actions do not change.

For instance, in classical planning, if loaded(r1,c1) is true in a state s, and the robot moves to a different location, then loaded(r1,c1) is still true in the new state s'. In situation calculus, because state transitions are computed by deduction, if loaded(r1,c1,s) is true, we cannot conclude that loaded(r1,c1,s') is true unless there is an axiom saying so. For instance, the formula

$$\forall r \forall l \forall l' \forall s (\mathsf{Poss}(\mathsf{move}(r,l,l'),s) \rightarrow (\mathsf{loaded}(r,\mathsf{do}(\mathsf{move}(r,l,l'),s) \leftrightarrow \mathsf{loaded}(r,s))) \tag{12.6}$$

states that, if the action move(r,l,l') is applicable in s, then, if the robot is loaded in s, the robot will still be loaded after moving, and it will not be loaded if it is not loaded in s. These kinds of formulas must be axioms of the logical theory and are called *frame axioms.*

12.2.3 Planning Domains, Problems, and Solutions

Given the proper constants, function symbols, and predicate symbols, a *planning domain* in situation calculus is described by $D = (Ax_p, Ax_e, Ax_f)$, where the elements are defined as follows.

- Ax_p is a set of action precondition axioms.
- Ax_e is a set of action effect axioms.
- Ax_f is a set of frame axioms.

We need to define what *plans* are in situation calculus. In order to do this we introduce a function symbol ; in $\mathcal{L}$. We use an infix notation for ; rather than ordinary prefix notation, i.e., we write $x;y$ in place of $;(x,y)$.[1] Then any term denoting an action is a *plan.* If π_1 and π_2 are plans, then $\pi_1;\pi_2$ is also a plan. The intended meaning of ; is sequential composition: in $\pi_1;\pi_2$ we first execute π_1 and then π_2. This definition of *plan* implies totally ordered plans.

The execution of a plan π is defined through the symbol Exec. Intuitively, Exec(π,s,s') holds if the plan π leads from situation s to situation s'. Let α be a variable denoting an action and π_1 and π_2 be variables denoting plans. Then we can use the following notation to abbreviate some of the expressions of $\mathcal{L}$[2]:

$$\mathsf{Exec}(\alpha,s,s') \text{ means } \mathsf{Poss}(\alpha,s) \wedge s' = \mathsf{do}(\alpha,s)$$

$$\mathsf{Exec}(\pi_1;\pi_2,s,s') \text{ means } \exists s''(\mathsf{Exec}(\pi_1,s,s'') \wedge \mathsf{Exec}(\pi_2,s'',s')$$

1. We should also introduce the fundamental axioms that allow us to prove that $(x;y);z = x;(y;z)$. We refer to Levesque *et al.* [360] for a formal account.
2. Formally, the abbreviation Exec(α,s,s') should take into account the need for restoring the situation arguments to any functional fluent of α in Poss(α,s).

In order to define a classical planning problem, we need to represent initial and goal states. Initial states are denoted by a constant s0 in the language called the *initial situation*. The initial situation is described by a formula called the *initial situation axiom* Ax_0. It is a formula of $\mathcal{L}$ that either does not contain any situation or contains only the situation s0. Thus, for instance, the formula $\text{at(r1,l1,s0)} \wedge \neg\text{loaded(r1,s0)}$ describes an initial situation where the robot r1 is at location l1 and is not loaded. Goal states are described by a formula $\Phi_g(s)$ called the *goal formula*. $\Phi_g(s)$ is a formula whose only free variable is s. Thus, for instance, the formula $\text{at(r1,l2,}s) \wedge \text{loaded(r1,}s)$ describes the goal of reaching a state where the robot r1 is at location l2 and is loaded.

A *planning problem* $(D, Ax_0, \Phi_g(s))$ is a planning domain D, an initial situation Ax_0, and a goal formula $\Phi_g(s)$. A *solution* to a planning problem $(D, Ax_0, \Phi_g(s))$ is a plan π such that

$$\exists s(\text{Exec}(\pi, \text{s0}, s) \wedge \Phi_g(s))$$

is entailed[3] by D and Ax_0.[4]

Example 12.1 Consider the DWR example where we have one robot r1, two adjacent locations l1 and l2, and the actions move(r1,l1,l2), move(r1,l2,l1), and load(r1). The action precondition axioms are:

$$\forall r \forall l \forall l' \forall s(\text{Poss(move(}r,l,l'),s) \leftrightarrow \text{at}(r,l,s))$$

$$\forall r \forall l \forall l' \forall s(\text{Poss(load(}r),s) \leftrightarrow \neg\text{loaded}(r,s))$$

The action effect axioms are:

$$\forall r \forall l \forall l' \forall s(\text{Poss(move(}r,l,l'),s) \rightarrow \text{at}(r,l',\text{do(move(}r,l,l'),s)))$$

$$\forall r \forall l \forall l' \forall s(\text{Poss(move(}r,l,l'),s) \rightarrow \neg\text{at}(r,l,\text{do(move(}r,l,l'),s)))$$

$$\forall r \forall l \forall l' \forall s(\text{Poss(load(}r),s) \rightarrow \text{loaded}(r,\text{do(load(}r),s)))$$

The frame axioms are:

$$\forall r \forall l \forall l' \forall s(\text{Poss(move(}r,l,l'),s) \rightarrow (\text{loaded}(r,\text{do(move(}r,l,l'),s)) \leftrightarrow \text{loaded}(r,s)))$$

$$\forall r \forall l \forall l' \forall s(\text{Poss(load(}r,s)) \rightarrow (\text{at}(r,l,l',\text{do(load(}r),s)) \leftrightarrow \text{at}(r,l,l',s)))$$

The initial situation axiom is:

$$\text{at(r1,l1,s0)} \wedge \neg\text{loaded(r1,s0)}$$

The goal formula is:

$$\text{at(r1,l2,}s) \wedge \text{loaded(r1,}s)$$

3. See Appendix B for a formal notion of entailment in first-order logic.
4. In this informal presentation, we skip the discussion of *fundamental axioms*, i.e., domain independent axioms, such as unique name axioms for actions and situations. See Levesque *et al.* [360] for a detailed description of fundamental axioms.

We can then use a first-order logic theorem prover to obtain the plan that achieves the goal described by the goal formula. ■

12.2.4 Plans as Programs in Situation Calculus

This section describes how to generalize the definition of a plan to allow plans that are not necessarily sequential. We discuss this extension briefly and informally through some examples; see Levesque *et al.* [360] for formal definitions.

Test actions are introduced through the symbol ?:

$$\text{Exec(at(r1,l1)?},s,s') \text{ means } \text{at(r1,l1},s) \wedge s = s'$$

The formula Exec(at(r1,l1)?,s,s') is true only if the test at(r1,l1)? succeeds in s and the resulting state is s'. Note, however, that the test at(r1,l1)? does not *change* s. Thus Exec(at(r1,l1)?,s,s') is true if and only if at(r1,l1,s) is true and $s = s'$.

Nondeterministic choices are expressed through the symbol |:

$$\text{Exec(move(r1,l1,l2) | move(r1,l1,l3)},s,s') \text{ means}$$
$$(\text{Exec(move(r1,l1,l2)},s,s') \vee \text{Exec(move(r1,l1,l3)},s,s'))$$

The formula Exec(move(r1,l1,l2) | move(r1,l1,l3),s,s') is true if s' is the resulting state of executing either move(r1,l1,l2) or move(r1,l1,l3) in s.

Nondeterministic iterations are expressed through the symbol $*$. For instance, Exec(load(r1,c1)$*$,s,s') means that executing load(r,c) zero or more times in s produces s'.

Nondeterministic iteration and test actions can be used to define the usual programming constructs for conditionals and loops. Let ϕ be a formula, and let π, π_1, and π_2 be plans. Then:

$$\text{if } \phi \text{ then } \pi_1 \text{ else } \pi_2 \text{means } (\phi?;\pi_1)|(\neg\phi?;\pi_2)$$
$$\text{while } \phi \text{ do } \pi \text{ means } (\phi?;\pi)*;\neg\phi?$$

if ϕ then π_1 can be defined analogously. The above planning constructs can then be used by the user to write programs that specify plans.

The logical language described here can be viewed as a high-level programming language. Planning in this framework is equivalent to executing the logical program. The program in the following example is a complete specification of a plan.

Example 12.2 In the DWR domain, we can instantiate the following formula

```
∀l∀l' if at(r1,l') then
    while ∃c in(c,l) do
        move(r1,l',l);load(r1,c);move(r1,l;l');unload(r1,c)
```

to a plan that moves iteratively the robot r1 to transport all the containers in a given location to the original location where the robot comes from. ■

The language allows for incomplete specifications of plans, in which case ordinary first-order theorem proving is used to generate the plan.

Example 12.3 In situation calculus, it is possible to write the following program

```
∀l∀l′ if at(r1,l′) then
    while ∃c in(c,l) do
        move(r1,l′,l);π;move(r1,l;l′);unload(r1,c)
```

with a condition that states that π should lead from a situation where r1 is unloaded in l' to a situation where r1 is loaded in l' with a container c. In this case, the plan π has to be generated by deduction. Fortunately, this is a rather easy task because the plan is just one action, load(r1,c).[5] ■

Example 12.4 As an extreme case, we may leave the theorem prover a harder task and write the following program with a proper initial and goal condition for π.

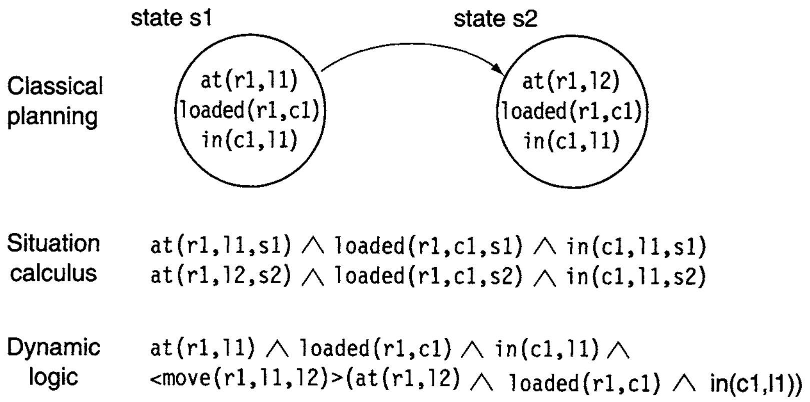
```
∀l∀l′ if at(r1,l′) then
while ∃c in(c,l) do π
```
■

12.3 Dynamic Logic

Dynamic logic, like situation calculus, is a logic for reasoning about states and actions that change states. In dynamic logic, states are represented in *one* logical theory as well. However, contrary to situation calculus, states are not represented explicitly in the language of the logic. Which atoms hold in different states is stated through modal operators, similar to those of temporal logics, like LTL (see Chapter 10) and CTL (see Chapter 17). While temporal logics have been devised to reason about temporal evolutions, dynamic logic is used to reason about actions (or programs) that change the state. For this reason, modal operators are "parameterized" with actions. If a is an action, then $[a]$ and $\langle a\rangle$ are modal operators that state what is true after action a is executed. For example, to say that at(r1,l2) is true in a state after the execution of action move(r1,l1,l2), one would write move(r1,l1,l2) > at(r1,l2). Figure 12.2 shows the conceptual difference in representation between classical planning, situation calculus, and dynamic logic.

In this section, we first define the language of a very simple fragment of dynamic logic (Section 12.3.1), its semantics (Section 12.3.2), and its deductive machinery,

5. In order to present all of this formally, we would need the introduction of procedures in the logical framework [360].



at(rl,1l,sl) ∧ loaded(rl,cl,sl) ∧ in(cl,1l,sl)
at(rl,12,s2) ∧ loaded(rl,cl,s2) ∧ in(cl,1l,s2)

at(rl,1l) ∧ loaded(rl,cl) ∧ in(cl,1l) ∧
<move(rl,1l,12)>(at(rl,12) ∧ loaded(rl,cl) ∧ in(c1,l1))

Figure 12.2 Classical planning, situation calculus, and dynamic logic.

i.e., axioms and inference rules (Section 12.3.3). These are the basic ingredients of planning domains, problems, and solutions in dynamic logic (Section 12.3.4). We start with a rather simple framework, in which plans are restricted to be sequences of actions, and we then extend it to represent plans involving control structures, such as conditional and iterative plans (Section 12.3.5). We finally give some guidelines on how deductive planning can be guided by user-defined control strategies (Section 12.3.6).

12.3.1 The Language of the Logic

The set of plans Π includes the set of basic actions Π_0. For instance, the basic action move(r1,l1,l2) is a plan. Furthermore, for every pair of plans π_1 and π_2, the plan $\pi_1;\pi_2$ is also in Π. The set of formulas Φ is inductively defined starting from a set of atomic propositions Φ_0 and the set of plans Π.

- True and False are formulas in Φ.
- $\Phi_0 \subseteq \Phi$.
- If p and q are formulas in Φ, then $\neg p$ and $p \wedge q$ are also formulas in Φ.
- If p is a formula in Φ and π is a plan in Π, then $\langle\pi\rangle p$ is a formula in Φ.

The formula $\langle\pi\rangle p$ states that p is true in *at least one* of the possible states that result from the execution of π. For example, the intended meaning of the formula

$$\langle \text{move(r1,l1,l2); load(r1)} \rangle (\text{at(r1,l2)} \wedge \text{loaded(r1)})$$

is that if we first move the robot r1 to location l2 and then we load it, then the robot is at l2 and is loaded.

Let $[\pi]p$ be an abbreviation of $\neg\langle\pi\rangle\neg p$. Then $[\pi]p$ states that p holds in *all* the states after executing π.

From this discussion, it should be clear that dynamic logic can represent nondeterministic actions. Indeed, as we will see in the next section, actions and plans can be nondeterministic.

12.3.2 The Semantics

We provide semantics to plans with a function $\rho{:}\Pi \rightarrow 2^{S\times S}$, which maps a plan π to a set of pairs of states (s, s'). An action a is said to be *applicable* to a state s if there exists a state s' such that $(s, s') \in \rho(a)$. Intuitively, s is the state where π is executed and s' is a state after the execution of π. When we concatenate sequentially two actions, we get the pairs with the initial states of the first action and the final states of the second action, provided that the first one leads to states where the second one is applicable:

$$\rho(\pi_1;\pi_2) = \{(s,t) \in S \times S \mid \exists s' \in S.\,(s,s') \in \rho(\pi_1) \wedge (s',t) \in \rho(\pi_2)\}$$

Let a proposition p be true in a state s. Then we write $p \in s$. The set of states in which $\langle\pi\rangle p$ is true is the following:

$$\{s \in S \mid \exists s'.\,(s,s') \in \rho(\pi) \wedge p \in s'\}$$

Figure 12.3 describes the semantics of formulas of the form $\langle\pi\rangle p$. If $\langle a\rangle p$ holds in state s_1, i.e., $s_1 \in \tau(\langle a\rangle p)$, and a leads from s_1 to s_2, i.e., $(s_1, s_2) \in \rho(a)$, then p holds in s_2, i.e., $s_2 \in \tau(p)$. Notice that in the example in Figure 12.3, we suppose that s_2 is the only state such that $(s_1, s_2) \in \rho(a)$, i.e., a is deterministic when applied to s_1. If there are more than one pair, then p holds in at least one of the resulting states. If $\langle a;b\rangle p$ holds in s_1, i.e., $s_1 \in \tau(\langle a;b\rangle p)$, and a leads from s_1 to s_2, i.e., $(s_1, s_2) \in \rho(a)$, and b leads from s_2 to s_3, i.e., $(s_2, s_3) \in \rho(b)$, then p holds in s_3, i.e., $s_3 \in \tau(p)$.

An *interpretation* is a triple $M = (S, \tau, \rho)$. A formula $p \in \Phi$ is *valid in an interpretation* $M = (S, \tau, \rho)$ (written $M \models p$) if and only if $\tau(p) = S$; p is *valid* (written $\models p$) if it is valid in every interpretation.

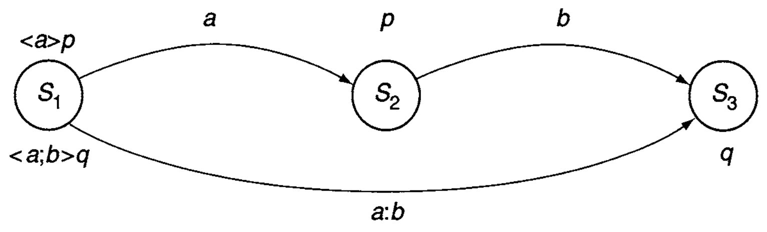


Figure 12.3 Semantics of formulas and plans.

12.3.3 The Deductive Machinery

The deductive machinery of the logic is based on a set of axioms and a set of inference rules. Any axiom of the propositional calculus is an axiom of the logic.[6] Moreover, we have the following axioms:

$$\langle \pi_1; \pi_2 \rangle p \leftrightarrow \langle \pi_1 \rangle (\langle \pi_2 \rangle p)$$
$$\langle \pi \rangle (p \vee q) \leftrightarrow (\langle \pi \rangle p \vee \langle \pi \rangle q)$$

It is easy to show that these formulas are valid. We have two inference rules.

1. From $p, p \rightarrow q$, derive q. This is the usual modus ponens rule.
2. From p, derive $\neg\langle a \rangle \neg p$. This rule, which is called necessitation, states that if p holds in all states, then the negation of p cannot hold in any of the states reached by a.

If a formula p follows from the axioms under the rules of inference, we say it is *provable* and write $\vdash p$. If p follows from a formula q, we say that it is *derivable* from q and we write $q \vdash p$.

12.3.4 Planning Domains, Problems, and Solutions

A *planning domain* is a triple $(\Phi_0, \Pi_0, Ax_{\Pi_0})$, where Φ_0 is a set of atomic propositions, Π_0 is the set of actions, and Ax_{Π_0} is the union of the following three disjoint sets.

1. A set of propositional formulas called *domain axioms*.
2. A set of formulas called *action axioms* that describe the action preconditions and effects. They are of the form $p \rightarrow \langle a \rangle q$ or of the form $p \rightarrow [a]q$, where a is any basic action in Π_0, and p and q are any propositional formulas. We call p the preconditions and q the effects.
3. *Frame axioms*, which describe what actions do not change. See Section 12.2 for a discussion of why frame axioms are needed.

Example 12.5 Consider a DWR example where we have one robot r1, two adjacent locations l1 and l2, and the actions move(r1,l1,l2), move(r1,l2,l1), and load(r1). The set of basic propositions and actions are:

$$\Phi_0 = \{\text{at(r1,l1)}, \text{at(r1,l2)}, \text{loaded(r1)}\}$$
$$\Pi_0 = \{\text{move(r1,l1,l2)}, \text{move(r1,l2,l1)}, \text{load(r1)}\}$$

6. See Appendix B for a description of the axioms of propositional and first-order logic and for a definition of interpretation.

The set Ax_{Π_0} contains the following formulas:

$$\begin{aligned}
\neg\mathtt{at(r1,l2)} &\leftrightarrow \mathtt{at(r1,l1)}\ \mathtt{(domain)}\\
\mathtt{at(r1,l1)} &\rightarrow \langle\mathtt{move(r1,l1,l2)}\rangle\mathtt{at(r1,l2)}\ \mathtt{(move\ l2)}\\
\mathtt{at(r1,l2)} &\rightarrow \langle\mathtt{move(r1,l2,l1)}\rangle\mathtt{at(r1,l1)}\ \mathtt{(move\ 2l)}\\
\neg\mathtt{loaded(r1)} &\rightarrow \langle\mathtt{load(r1)}\rangle\mathtt{loaded(r1)}\ \mathtt{(load)}\\
\mathtt{loaded(r1)} &\rightarrow \langle\mathtt{move(r1,l1,l2)}\rangle\mathtt{loaded(r1)}\ \mathtt{(frame\ 1)}\\
\mathtt{loaded(r1)} &\rightarrow \langle\mathtt{move(r1,l2,l1)}\rangle\mathtt{loaded(r1)}\ \mathtt{(frame\ 2)}\\
\neg\mathtt{loaded(r1)} &\rightarrow \langle\mathtt{move(r1,l1,l2)}\rangle\neg\mathtt{loaded(r1)}\ \mathtt{(frame\ 3)}\\
\neg\mathtt{loaded(r1)} &\rightarrow \langle\mathtt{move(r1,l2,l1)}\rangle\neg\mathtt{loaded(r1)}\ \mathtt{(frame\ 4)}\\
\mathtt{at(r1,l1)} &\rightarrow \langle\mathtt{load(r1)}\rangle\mathtt{at(r1,l1)}\ \mathtt{(frame\ 5)}\\
\mathtt{at(r1,l2)} &\rightarrow \langle\mathtt{load(r1)}\rangle\mathtt{at(r1,l2)}\ \mathtt{(frame\ 6)}\\
\neg\mathtt{at(r1,l1)} &\rightarrow \langle\mathtt{load(r1)}\rangle\neg\mathtt{at(r1,l1)}\ \mathtt{(frame\ 7)}\\
\neg\mathtt{at(r1,l2)} &\rightarrow \langle\mathtt{load(r1)}\rangle\neg\mathtt{at(r1,l2)}\ \mathtt{(frame\ 8)}
\end{aligned}$$

The domain axiom (domain) states that a robot is in only one of the two possible locations at a time. The formulas (move12), (move21), and (load) describe the preconditions and effects of the actions. The other formulas, (frame1) through (frame8), are called *frame axioms.* They describe what does not change after an action is executed. ■

A *planning problem* is (D, Φ_0, Φ_g), where D is a planning domain, Φ_0 is a propositional formula describing the initial state, and Φ_g is a propositional formula describing the goal state. A *solution* to a planning problem is a plan $\pi \in \Pi$ such that

$$Ax_{\Pi_0} \vdash \Phi_0 \rightarrow \langle\pi\rangle\Phi_g$$

Example 12.6 Consider Example 12.5. Let Φ_0 be at(r1,l1) $\wedge$ ¬loaded(r1) and Φ_g be at(r1,l2) $\wedge$ loaded(r1). From (move12), (load), and (domain) we derive:

$$(\mathtt{at(r1,l1)} \wedge \neg\mathtt{loaded(r1)}) \rightarrow \langle\mathtt{move(r1,l1,l2);load(r1)}\rangle\mathtt{at(r1,l2)} \wedge \mathtt{loaded(r1)}$$

■

12.3.5 Extensions

In the previous sections we have discussed deductive planning within a very small subset of Propositional Dynamic Logic (PDL) [259]. PDL can express plans that include control structures like conditionals and loops, defined on the basis of nondeterministic choices, tests, and repetitions. Plans and formulas are defined inductively as follows.

- *True*, *False* $\in \Phi$, $\Phi_0 \subseteq \Phi$.
- If $p, q \in \Phi$, then $\neg p \in \Phi$ and $p \wedge q \in \Phi$.
- If π_1 and π_2 are plans in Π, then the following hold.
  - $\pi_1; \pi_2$ is a plan in Π (sequence).
  - $\pi_1 \cup \pi_2$ is a plan in Π (nondeterministic choice).
  - π^* is a plan in Π (nondeterministic repetition).
- If $p \in \Phi$, then $p? \in \Pi$ (test).
- If $\alpha \in \Pi$, then $\langle\alpha\rangle p \in \Phi$.

According to the definition above, plans can be obtained as sequences of actions, $\alpha; \beta$(do α and then β); nondeterministic choices, $\alpha \cup \beta$(do α or β, nondeterministically); nondeterministic repetitions, α^*(do α zero or more times); and tests, $p?$ (if p is true, then proceed with the execution of the plan; otherwises, stop execution). This allows us to define conditionals and loops. For any $p \in \Phi$ and $\pi, \pi_1, \pi_2 \in \Pi$, we define:

- **if** p **then** π_1 **else** $\pi_2 = p?; \pi_1 \cup (\neg p)?; \pi_2$
- **while** p **do** $\pi = (p?; \alpha)^*; (\neg p)?$

12.3.6 User-Defined Control Strategies as Tactics

In deductive planning, plan generation is done by searching for a proof. A lot of effort has been devoted to finding domain-independent heuristics for this search problem. However, automatic theorem proving is well known to be very hard, e.g., semidecidable in first-order logic. It is unlikely that automatic theorem proving can be used to solve planning problems of realistic complexity. For this reason, an alternative approach has been devised where the user can specify control knowledge for the search of a proof. This approach is called *tactical theorem proving*, and the user-defined control strategies are called *tactics*.

Tactics are programs that control a theorem prover by telling which are the inference rules to use, to which formulas they should be applied, and in which order. Tactics are either primitive or compound. *Primitive tactics* are programs that apply basic inference rules, i.e., basic proof steps. If a rule is applicable, then the corresponding tactic returns the formula that is the conclusion of the rule; if it is not applicable, it returns failure. This is how tactics deal with "wrong" attempts to generate a proof, i.e., attempts to apply inference rules that are not applicable. Indeed, when writing a tactic that has to deal with a large or complex proof, the user may not know whether a rule is applicable at a certain point in the proof search. As an example, consider the basic rule of modus ponens: from a formula A and a formula $A \rightarrow B$, the rule derives B. The corresponding primitive tactic `modus-ponens-tac`

```
modus-ponens-tac($\phi_1$,$\phi_2$)          ;; primitive tactic for modus ponens
    if premise($\phi_1$,$\phi_2$)            ;; rule preconditions
        then return conclusion($\phi_2$)     ;; rule application
        else exit with "failure"             ;; failure generation
end
```

Figure 12.4 A primitive tactic.

is shown in Figure 12.4. The primitive tactic modus-ponens-tac tests whether the rule is applicable. The function premise(ϕ_1,ϕ_2) tests whether ϕ_1 is a premise of ϕ_2, namely, whether ϕ_2 is $\phi_1 \rightarrow \phi_3$, where ϕ_3 is any formula. If it is, then the tactic returns the conclusion of the rule, i.e., ϕ_3. It fails otherwise.

Compound tactics are compositions of primitive tactics through operators called *tacticals*. Some of the possible tacticals are the following.

- then(tac1,tac2) applies tac1. If tac1 fails, then the tactical fails; otherwise, it applies tac2.
- orelse(tac1,tac2) applies tac1. If tac1 fails, then the tactical applies tac2.
- try(tac1) applies tac1. If tac1 fails, then the tactical returns the original formula it is applied to.
- repeat(tac1) applies tac1 until the tactical fails.

Example 12.7 Consider the tactic conjunction-tac(ϕ,ψ) that always returns $\phi \wedge \psi$. Then

$$\text{orelse(modus-ponens-tac,conjunction-tac)}(B, A \rightarrow B)$$

returns $B \wedge (A \rightarrow B)$.

As a further example, the application of the following tactic

$$\text{repeat(tac1)}(A, A \rightarrow (A \rightarrow (A \rightarrow B)))$$

returns B.

■

12.4 Discussion and Historical Remarks

In this section, we discuss some relations with the other approaches that are described in this part of the book, we discuss the main pros and cons of deductive planning, and we provide some historical remarks.

The "plans as programs" approach is somewhat analogous to HTN planning (see Chapter 11). In HTN planning, each method corresponds roughly to an incomplete plan. The main difference is that in the "plans as program" approach, one can do operations that are analogous to HTN decomposition and also operations that are analogous to state-space search. This is both an advantage and a disadvantage. The advantage is the expressiveness in specifying control strategies, the disadvantage in the loss of efficiency.

The "plans as programs" approach has been proposed in other logical frameworks than situation calculus, e.g., in dynamic logic [496], in process logic [510], and in programming logics that have been extended with temporal operators [499].

The main conceptual difference of the tactics-based approach with respect to other approaches, such as plans as programs and HTN planning, is that tactics are programs whose basic constructs are the basic routines of the theorem prover. They are not specific to planning but are general techniques for theorem proving.

The main advantage of deductive planning is its expressiveness. Compared with classical planning, deductive planning allows for relaxing most of the restrictions described in Chapter 1. This is true even in the very simple logical frameworks we have presented in this chapter. We can use logics that are expressive enough to represent nonfinite state-transition systems, nondeterminism, partial observability, and extended goals.

Nonfinite state-transition systems can be of interest, e.g., in the case new objects are introduced in the domain. For instance, we may not know a priori how many containers are in a DWR domain because new containers can be delivered by boats. Nonfinite state systems are also useful when we have variables that range over real values.

Both situation calculus and dynamic logic can represent and reason about actions with nondeterministic effects, which can be represented with disjunctive formulas. Dynamic logic has modal operators that can distinguish when an effect holds in all the resulting states ($[a]p$) and in at least one state ($\langle a \rangle p$). Partial observability has also been represented through knowledge or sensing actions, e.g., in the situation calculus approach [359] and in the dynamic logic approach [496]. Extended goals can be expressed, e.g., in frameworks that combine temporal and programming logic [499].

Another advantage of deductive planning is that further functionalities like plan verification are given for free. Given a plan, one can prove properties of the plan, e.g., that it achieves a desired goal, that it preserves a given property, and so on. While this feature is not much relevant in classical planning, it is important in cases where plan validation is not trivial, as in planning under uncertainty with extended goals. However, it should be noted that other frameworks for planning under uncertainty, such as those based on model checking, allow for automatic plan validation (see Chapter 17).

In spite of all these potential advantages, the planning technique matured outside of the deductive approach. Indeed, most of the issues discussed previously (e.g., nondeterminism, plan verification) have been tackled with different techniques. Indeed, the major problem of deductive planning is the lack of procedures that generate plans automatically. This is due mainly to the expressiveness of the logic. The work described in this chapter is an attempt to deal with this problem by allowing for user-defined strategies in the logical framework. Important sources in this area are the work on tactics-based deductive planning [54, 71], and on GOLOG [360].

The situation calculus approach was the first to be used for planning. In his seminal work, Green [249] proposes a situation calculus for modeling planning domains and a general purpose resolution theorem prover for generating plans. Manna and Waldinger [377] proposed a theory where recursive plans are generated by deduction in a resolution-based and induction-based tableau calculus. The framework allows for plans involving programming control structures, such as conditionals, loops, and recursion. The early work on program synthesis [378] is very close to this idea.

The "plans as programs" approach relies on modal logic and specifically on dynamic and temporal logic. The modal logic approach was first proposed in early work by Rosenschein [457], within a framework based on Propositional Dynamic Logic [259], and then extended by Kautz to first-order dynamic logic [318]. In these papers the plan generation process is still algorithmic rather than deductive.

A more recent approach to deductive planning has been introduced by W. Stephan and S. Biundo [499]. In this work, actions are composed out of add and delete operations (somehow similar to Strips), plans are combined with control structures including nondeterministic choice and recursion. This approach was the first to provide an efficient solution to the frame problem in the "plans as programs" paradigm. The framework has been implemented within a theorem prover for dynamic logic where users can specify strategies, called tactics in theorem proving. The idea of using hand-coded and reusable theorem-proving strategies (tactics) was first proposed in the PHI system [54, 71], a theorem prover based on a modal interval-based logic, called LLP (Logical Language for Planning), which combines dynamic logic and temporal logic. Interestingly enough, not only does the logical framework allow for plan generation by deduction but it also provides the ability to do plan recognition by means of abduction and to combine plan recognition with plan generation [329].

A different line of research pursues planning based on linear logic [69], which has received some recent attention in the work by Cresswell [134]. Moreover, there is an active line of research on the use of situation calculus as a programming language, in the style of logic programming. GOLOG [360] is a programming language based on situation calculus. The GOLOG approach was first used in cognitive robotics [359] and is recently being used in planning for information gathering and planning for the web.

12.5 Exercises

12.1 Formalize the planning problem described in Example 7.1 in situation calculus and in dynamic logic. Provide a sketch of the proofs that generate a solution plan.

12.2 Write a tactic that automates the generation of a solution plan for a dynamic logic formulation of the planning problem in Example 7.1.

12.3 Write a plan-as-program in situation calculus that automates the generation of a solution plan for a situation calculus formulation of the planning problem in Example 7.1.

12.4 Complicate Example 7.1 with a loading and unloading operation. Formalize the example in situation calculus and in dynamic logic. Provide a sketch of the proofs that generate a solution plan. Write a tactic and a plan-as-program that generate a solution plan.

12.5 Formalize the full DWR domain in situation calculus and in dynamic logic. Formalize some planning problems.

12.6 In Example 7.1, suppose you do not know in which location the robot is initially. Suppose the planning problem is to find a plan such that, no matter where the robot is initially, the plan leads you to the same goal of Example 7.1 (the robot must be in l2). Formalize the planning problem in situation calculus and in dynamic logic, and provide a sketch of the proofs that generate solution plans.

12.7 Provide a situation calculus formulation of the DWR domain with the following modifications.

- The number of containers is c, with $c > 0$.
- The number of robots is r, with $r > 0$.
- The number of locations is l, with $l \geq r$.

Then formalize a planning problem where in the initial state all the containers are in location l_i and in the goal state all the containers must be in location l_j, with $l_j \neq l_i$. Provide a high-level sketch of the proof that generates a solution plan.

12.8 Provide a dynamic logic formulation of the DWR domain with the following modification: If we have containers in more than one pile in the same location, then the loading operation may load a wrong container, i.e., it may load a container different from the designated one (e.g., the container loaded is one at the top of a different pile in the same location). Consider then the goal where you have to move a container to a different location. Does a solution exist? Show with a proof that there is or is not a solution.