CHAPTER 5 Plan-Space Planning

5.1 Introduction

In the previous chapter, we addressed planning as the search for a path in the graph Σ of a state-transition system. For state-space planning, the search space is given directly by Σ. Nodes are states of the domain; arcs are state transitions or actions; a plan is a sequence of actions corresponding to a path from the initial state to a goal state.

We shall be considering in this chapter a more elaborate search space that is not Σ anymore. It is a space where nodes are *partially specified plans*. Arcs are *plan refinement operations* intended to further complete a partial plan, i.e., to achieve an open goal or to remove a possible inconsistency. Intuitively, a refinement operation avoids adding to the partial plan any constraint that is not strictly needed for addressing the refinement purpose. This is called the *least commitment principle.* Planning starts from an initial node corresponding to an empty plan. The search aims at a final node containing a solution plan that correctly achieves the required goals.

Plan-space planning differs from state-space planning not only in its search space but also in its definition of a solution plan. Plan-space planning uses a more general plan structure than a sequence of actions. Here planning is considered as two separate operations: (1) the choice of actions, and (2) the ordering of the chosen actions so as to achieve the goal. A plan is defined as a set of planning operators together with ordering constraints and binding constraints; it may not correspond to a sequence of actions.

The search space is detailed in Section 5.2. Properties of solution plans are analyzed in Section 5.3; correctness conditions with respect to the semantics of state-transition systems are established. Algorithms for plan-space planning are proposed in Section 5.4. Several extensions are considered in Section 5.5. A comparison to state-space planning is offered in Section 5.6. The chapter ends with discussion and exercises.

5.2 The Search Space of Partial Plans

Generally speaking, a plan is a set of actions organized into some structure, e.g., a sequence. A partial plan can be defined as any subset of actions that keeps some useful part of this structure, e.g., a subsequence for state-space planning. All planning algorithms seen up to now extend step-by-step a partial plan. However, these were particular partial plans. Their actions are sequentially ordered. The total order reflects the intrinsic constraints of the actions in the partial plan and the particular search strategy of the planning algorithm. The former constraints are needed; a partial plan that is just an unstructured collection of actions would be meaningless because the relevance of a set of actions depends strongly on its organization. However, the constraints reflecting the search strategy of the algorithm are not needed. There can be advantages in avoiding them.

To find out what is needed in a partial plan, let us develop an informal planning step on a simple example. Assume that we already have a partial plan; let us refine it by adding a new action and let us analyze how the partial plan should be updated. We'll come up with four ingredients: adding actions, adding ordering constraints, adding causal relationships, and adding variable binding constraints.

Example 5.1 In the *DWR* domain, consider the problem where a robot r1 has to move a container c1 from pile p1 at location l1 to pile p2 and location l2 (see Figure 5.1). Initially r1 is unloaded at location l3. There are empty cranes k1 and k2 at locations l1 and l2, respectively. Pile p1 at location l1 contains only container c1; pile p2 at location l2 is empty. All locations are adjacent. ■

For Example 5.1, let us consider a partial plan that contains only the two following actions.

- take(k1,c1,p1,l1): crane k1 picks up container c1 from pile p1 at location l1.
- load(k1,c1,r1,l1): crane k1 loads container c1 on robot r1 at location l1.

Adding Actions. Nothing in this partial plan guarantees that robot r1 is already at location l1. Proposition at(r1,l1), required as a precondition by action load, is a *subgoal* in this partial plan. Let us add to this plan the following action.

- move(r1,*l*,l1): robot r1 moves from location l to the required location l1.

Adding Ordering Constraints. This additional move action achieves its purpose only if it is constrained to come *before* the load action. Should the move action come *before* or *after* the take action? Both options are feasible. The partial plan does not commit unnecessarily at this stage to either one. This follows the *least commitment principle* of not adding a constraint to a partial plan unless strictly needed. Here, the only needed constraint is to have move before load. Other constraints that may be

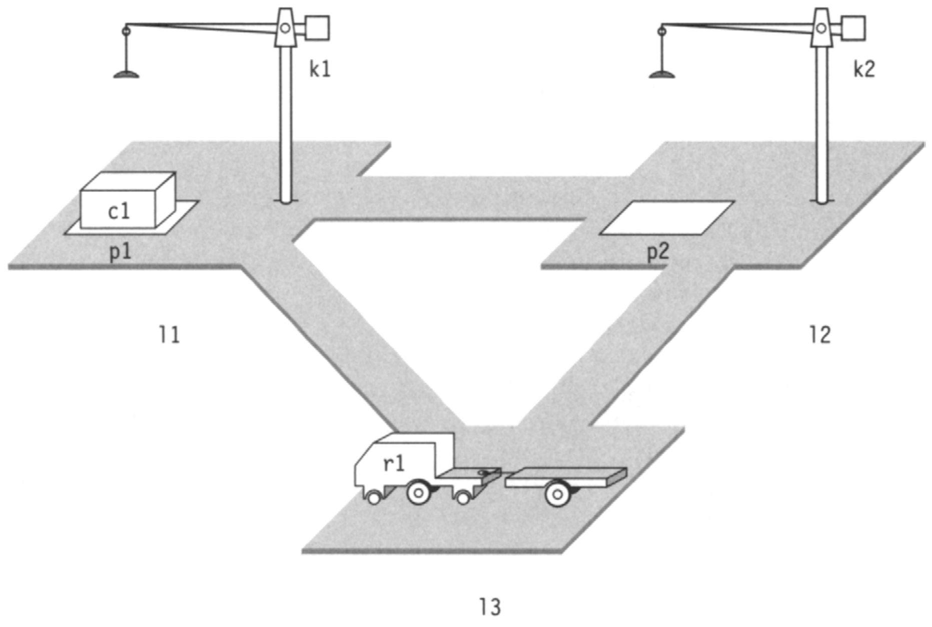

Figure 5.1 A simple DWR problem.

needed later while planning or even executing the plan may order move and take or may permit them to run concurrently.

Adding Causal Links. In the partial plan, we have added one action and one ordering constraint to another action already in the plan. Is that enough? Not quite. Because there is no explicit notion of a current state, an ordering constraint does not say that the robot stays at location l1 until the load action is performed. While pursuing the refinement of the partial plan, the planner may find other reasons to move the robot elsewhere, forgetting about the rationale for moving it to l1 in the first place. Hence, we'll be encoding explicitly in the partial plan the reason why action move was added: to satisfy the subgoal at(r1,l1) required by action load.

This relationship between the two actions move and load with respect to proposition at(r1,l1), is called a *causal link*. The former action is called the *provider* of the proposition, the later its *consumer*.[1] The precise role of a causal link is to state that a precondition of an action is *supported* by another action. Consequently, a precondition without a causal link is not supported. It is considered to be an open *subgoal* in the partial plan.

A provider has to be ordered before a consumer but not necessarily strictly before in a sequence: other actions may take place in between. Hence, a causal link does

1. Not in the sense of consuming a resource: load does not change this precondition.

not prevent interference between the provider and the consumer that may be due to other actions introduced later. Note that a causal link goes with an ordering constraint, but we may have ordering constraints without causal links. None of these relations is redundant with respect to the other.

Adding Variable Binding Constraints. A final item in the partial plan that goes with the refinement we are considering is that of variable binding constraints. Planning operators are generic schemas with variable arguments and parameters. Operators are added in a partial plan with systematic variable renaming. We should make sure that the new operator move concerns the same robot r1 and the same location l1 as those in operators take and load. What about location l the robot will be coming from? At this stage there is no reason to bind this variable to a constant. The same rationale of least commitment applies here as in the ordering between move and take. The variable l is kept unbounded. Later, while further refining the partial plan, we may find it useful to bind the variable to the initial position of the robot or to some other location.

To sum up, we have added to the partial plan an action, an ordering constraint, a causal link, and variable binding constraints. These are exactly the four ingredients of partial plans.

Definition 5.1 A *partial plan* is a tuple $\pi = (A, \prec, B, L)$, where:

- $A = \{a_1, \ldots, a_k\}$ is a set of partially instantiated planning operators.
- $\prec$ is a set of ordering constraints on A of the form $(a_i \prec a_j)$.
- B is a set of binding constraints on the variables of actions in A of the form $x = y$, $x \neq y$, or $x \in D_x$, D_x being a subset of the domain of x.
- L is a set of causal links of the form $\langle a_i \xrightarrow{p} a_j \rangle$, such that a_i and a_j are actions in A, the constraint $(a_i \prec a_j)$ is in $\prec$, proposition p is an effect of a_i and a precondition of a_j, and the binding constraints for variables of a_i and a_j appearing in p are in B.

■

A plan space is an implicit directed graph whose vertices are partial plans and whose edges correspond to refinement operations. An outgoing edge from a vertex π in the plan space is a *refinement operation* that transforms π into a successor partial plan π'. A refinement operation consists of one or more of the following steps.

- Add an action to A.
- Add an ordering constraint to $\prec$.
- Add a binding constraint to B.
- Add a causal link to L.

Planning in a plan space is a search in that graph of a path from an initial partial plan denoted π_0 to a node recognized as a solution plan. At each step, the planner has to choose and apply a refinement operation to the current plan π in order to achieve the specified goals. We now describe how goals and initial states are specified in the plan space.

Partial plans represent only actions and their relationships; states are not explicit. Hence, goals and initial states also have to be coded within the partial plan format as particular actions. Since preconditions are subgoals, the propositions corresponding to the goal g are represented as the preconditions of a dummy action, call it a_∞, that has no effect. Similarly, the propositions in the initial state s_0 are represented as the effects of a dummy action, a_0, that has no precondition. The initial plan π_0 is defined as the set $\{a_0, a_\infty\}$, together with the ordering constraint $(a_0 \prec a_\infty)$, but with no binding constraint and no causal link. The initial plan π_0 and any current partial plan represent goals and subgoals as preconditions without causal links.

Example 5.2 Let us illustrate two partial plans corresponding to Example 5.1 (see Figure 5.1). The goal of having container c1 in pile p2 can be expressed simply as in(c1,p2). The initial state is:

{adjacent(l1,l2), adjacent(l1,l3), adjacent(l2,l3),
adjacent(l2,l1), adjacent(l3,l1), adjacent(l3,l2),
attached(p1,l1), attached(p2,l2), belong(k1,l1), belong(k2,l2),
empty(k1), empty(k2), at(r1,l3), unloaded(r1), occupied(l3),
in(c1,p1), on(c1,pallet), top(c1,p1), top(pallet, p2)}

The first three lines describe rigid properties, i.e., invariant properties on the topology of locations, piles, and cranes; the last two lines define the specific initial conditions considered in this example.

A graphical representation of the initial plan π_0 corresponding to this problem appears in Figure 5.2. The partial plan discussed earlier with the three actions take, load, and move appears in Figure 5.3. In these figures, each box is an action

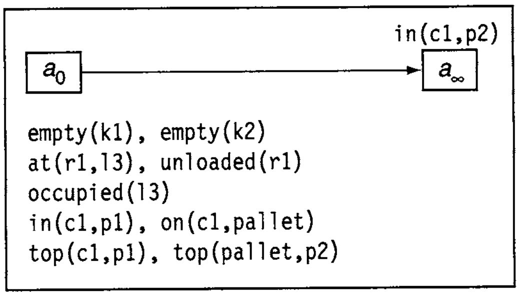

Figure 5.2 Initial plan π_0.

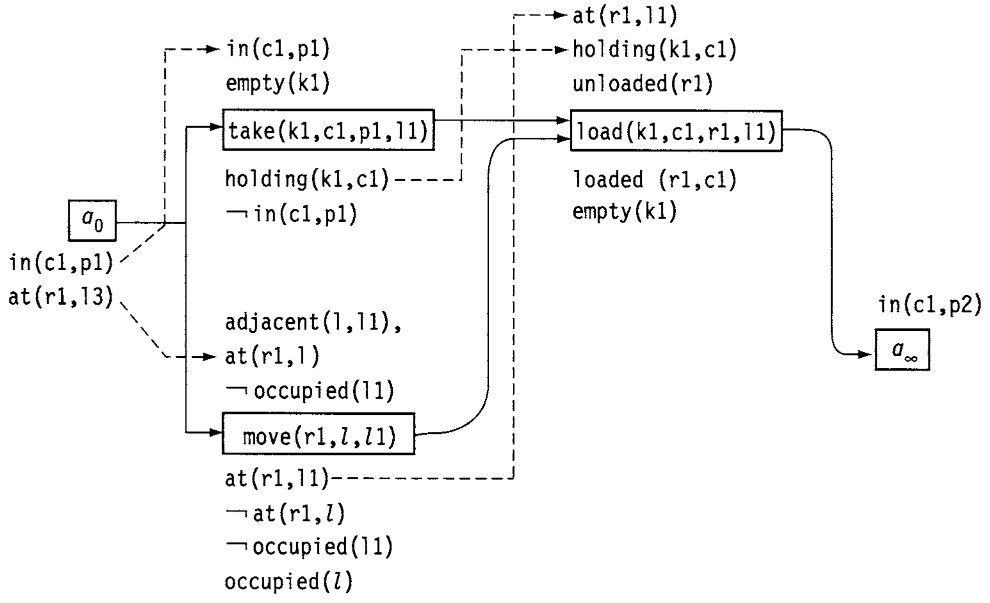

Figure 5.3 A partial plan.

with preconditions above and effects below the box. To simplify, rigid properties are not shown in the effects of a_0. Solid arrows are ordering constraints; dashed arrows are causal links; and binding constraints are implicit or shown directly in the arguments.

■

To summarize, a partial plan π is a structured collection of actions that provides the causal relationships for the actions in π, as well as their intrinsic ordering and variable binding constraints. A partial plan also provides subgoals in π as preconditions without causal links. A plan-space planner refines the partial plan by further ordering and constraining its actions or by adding new actions anywhere in the partial plan, as long as the constraints in π remain satisfiable. A partial plan enables us to neatly decouple two main issues in classical planning: (1) which actions need to be done in order to meet the goals, and (2) how to organize these actions.

It is convenient to view a partial plan π as representing a particular set of plans. It is the set of all sequences of actions corresponding to a path from the initial state to a goal state that can be obtained by refinement operations on π, i.e., by adding to π operators and ordering and binding constraints. A refinement operation on π reduces the set of plans associated with π to a smaller subset. This view will be formalized in the next section.

5.3 Solution Plans

In order to define planning algorithms in the plan space, we need to formally specify what is a solution plan in the plan space. A solution plan for a problem $P = (\Sigma, s_0, g)$ has been formally defined, with respect to the semantics of state-transition systems, as a sequence of ground actions from s_0 to a state in g (see Section 2.3.3). We now have to take into account that actions in a partial plan $\pi = (A, \prec, B, L)$ are only partially ordered and partially instantiated.

A consistent partial order $\prec$ corresponds to the set of all sequences of totally ordered actions of A that satisfy the partial order. Technically, these sequences are the topological orderings of the partial order $\prec$. There can be an exponential number of them. Note that a partial order defines a directed graph; it is consistent when this graph is loop free.

Similarly, for a consistent set of binding constraints B, there are many sequences of totally instantiated actions of A that satisfy B. These sequences correspond to all the ways of instantiating every unbounded variable x to a value in its domain D_x allowed by the constraints. The set B is consistent if every binding of a variable x with a value in the allowed domain D_x is consistent with the remaining constraints. Note that this is a strong consistency requirement.

Definition 5.2 A partial plan $\pi = (A, \prec, B, L)$ is a *solution plan* for problem $P = (\Sigma, s_0, g)$ if:

- its ordering constraints $\prec$ and binding constraints B are consistent; *and*
- every sequence of totally ordered and totally instantiated actions of A satisfying $\prec$ and B is a sequence that defines a path in the state-transition system Σ from the initial state s_0 corresponding to effects of action a_0 to a state containing all goal propositions in g given by preconditions of a_∞.

■

According to Definition 5.2, a solution plan corresponds to a set of sequences of actions, each being a path from the initial state to a goal state. This definition does not provide a computable test for verifying plans: it is not feasible to check all instantiated sequences of actions of A. We need a practical condition for characterizing the set of solutions. We already have a hint. Remember that a subgoal is a precondition without a causal link. Hence, a plan π meets its initial goals, and all the subgoals due to the preconditions of its actions, if it has a causal link for every precondition. However, this is not sufficient: π may not be constrained enough to guarantee that all possible sequences define a solution path in graph Σ.

Example 5.3 Consider Example 5.1, where we had a causal link from action move(r1,l,l1) to action load(k1,c1,r1,l1) with respect to proposition at(r1,l1). Assume that the final plan contains another action move(r,l',l''), without any ordering constraint that requires this action to be ordered before the previous move(r1,l',l1) or after

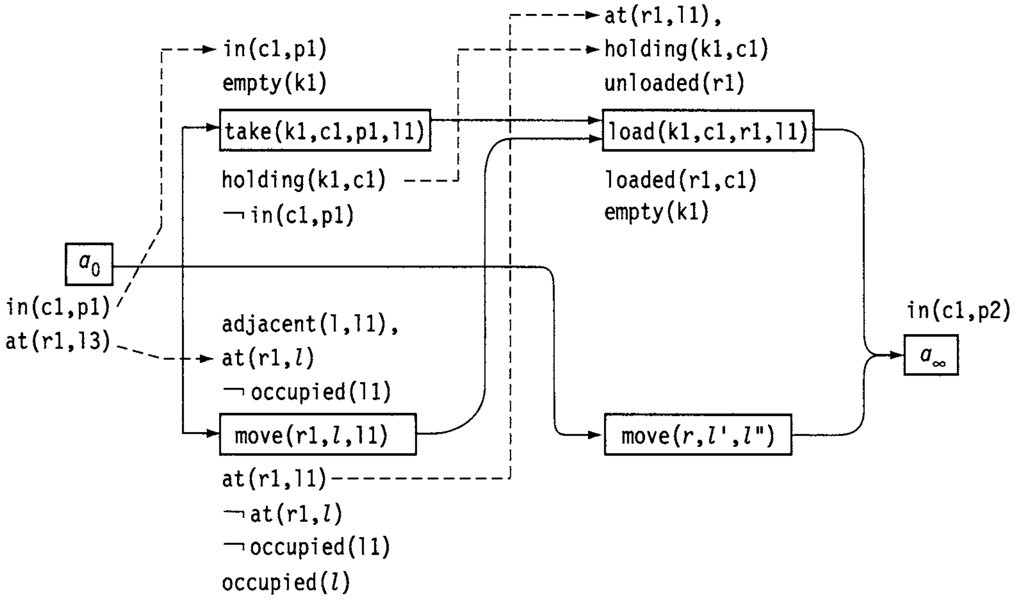

Figure 5.4 A plan containing an incorrect sequence.

action load, and without any binding constraint that requires r to be distinct from r1 or l' from l1 (Figure 5.4).

Among the set of totally ordered and instantiated sequences of actions of such a plan, there is at least one sequence that contains the subsequence

$\langle$move(r1,l, l1), ..., move(r1,l1,l''), ..., load(k1,c1,r1,l1)$\rangle$,

which is not a correct solution: the precondition at(r1,l1) is not satisfied in the state preceding action load. ■

Definition 5.3 An action a_k in a plan π is a *threat* on a causal link $\langle a_i \xrightarrow{p} a_j \rangle$ iff:

- a_k has an effect $\neg q$ that is possibly inconsistent with p, i.e., q and p are unifiable;
- the ordering constraints $(a_i \prec a_k)$ and $(a_k \prec a_j)$ are consistent with $\prec$; and
- the binding constraints for the unification of q and p are consistent with B. ■

Definition 5.4 A *flaw* in a plan $\pi = (A, \prec, B, L)$ is either:

- a subgoal, i.e., a precondition of an action in A without a causal link; or
- a threat, i.e., an action that may interfere with a causal link. ■

Proposition 5.1 *A partial plan $\pi = (A, \prec, B, L)$ is a solution to the planning problem $P = (\Sigma, s_0, g)$ if π has no flaw and if the sets of ordering constraints $\prec$ and binding constraints B are consistent.*

Proof Let us prove the proposition inductively on the number of actions in A.

Base step: For $A = \{a_0, a_1, a_\infty\}$, there is just a single sequence of totally ordered actions. Since π has no flaw, every precondition of a_1 is satisfied by a causal link with a_0, i.e., the initial state, and every goal or precondition of a_∞ is satisfied by a causal link with a_0 or a_1.

Induction step: Assume the proposition to hold for any plan having $(n-1)$ actions. Consider a plan π with n actions and without a flaw. Let $A_i = \{a_{i1}, \ldots, a_{ik}\}$ be a subset of actions whose only predecessor in the partial order $\prec$ is a_0. Every totally ordered sequence of actions of π satisfying $\prec$ starts necessarily with some action a_i from this subset. Since π has no flaw, all preconditions of a_i are met by a causal link with a_0. Hence every instance of action a_i is applicable to the initial state corresponding to a_0.

Let $[a_0, a_i]$ denote the first state of Σ after the execution of a_i in state s_0, and let $\pi' = (A', \prec', B', L')$ be the remaining plan. That is, π' is obtained from π by replacing a_0 and a_i with a single action (in the plan-space notation) that has no precondition and whose effects correspond to the first state $[a_0, a_i]$, and by adding to B the binding constraints due to this first instance of a_i. Let us prove that π' is a solution.

- $\prec'$ is consistent because no ordering constraint has been added from π to π'.
- B' is consistent because new binding constraints are consistent with B.
- π' has no threat because no action has been added in π, which had no threat.
- π' has no subgoal: every precondition of π that was satisfied by a causal link with an action $a \neq a_0$ is still supported by the same causal link in π'. Consider a precondition p supported by a causal link $\langle a_0 \xrightarrow{p} a_j \rangle$. Because a_i was not a threat in π, the effects of any consistent instance of a_i do not interfere with p. Hence condition p is satisfied in the first state $[a_0, a_i]$. The causal link in π' is now $\langle [a_0, a_i] \xrightarrow{p} a_j \rangle$.

Hence π' has $(n-1)$ actions without a flaw: by induction it is a solution plan. ■

Example 5.4 Let us augment Example 5.3 with actions:

```
move(r1,l1,l2)
unload(k1,c1,r1)
put(k1,c1,p2,l2)
```

The corresponding plan (Figure 5.5, where causal links are not shown) is a solution. ■

Finally, let us remark that Definition 5.2 and Proposition 5.1 allow for two types of flexibility in a solution plan: actions do not have to be totally ordered and they do not have to be totally instantiated. This remark applies also to a_∞ because all actions in a partial plan have the same syntactical status. In other words, the

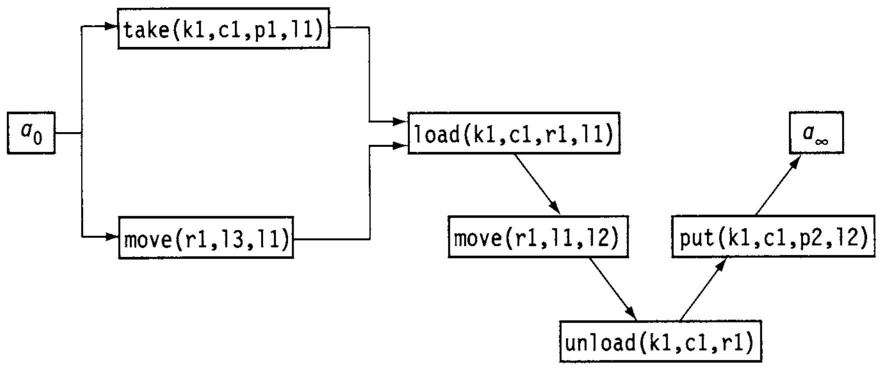

Figure 5.5 A solution plan.

flexibility introduced allows us to handle partially instantiated goals. Recall that variables in a goal are existentially quantified. For example, any state in which there is a container c in the pile p2 meets a goal such as in(c, p2) (see Exercise 5.4).

5.4 Algorithms for Plan-Space Planning

This section presents search procedures in the plan space. We first introduce a generic schema for planning in the plan space that clarifies the main ingredients of flaws and resolvers for flaws. We then introduce a particular control scheme that processes differently the two types of flaws in a partial plan.

5.4.1 The PSP Procedure

The characterization of the set of solution plans gives the elements needed for the design of planning algorithms in the plan space. Since π is a solution when it has no flaw, the main principle is to refine π, while maintaining $\prec$ and B consistent, until it has no flaw. The basic operations for refining a partial plan π toward a solution plan are the following.

- Find the flaws of π, i.e., its subgoals and its threats.
- Select one such flaw.
- Find ways to resolve it.
- Choose a resolver for the flaw.
- Refine π according to that resolver.

The resolvers of a flaw are defined so as to maintain the consistency of $\prec$ and B in any refinement of π with a resolver. When there is no flaw in π, then the conditions

```
PSP($\pi$)
    flaws ← OpenGoals($\pi$) ∪ Threats($\pi$)
    if flaws = ∅ then return($\pi$)
    select any flaw $\phi$ ∈ flaws
    resolvers ← Resolve($\phi, \pi$)
    if resolvers = ∅ then return(failure)
    nondeterministically choose a resolver $\rho$ ∈ resolvers
    $\pi'$ ← Refine($\rho, \pi$)
    return(PSP($\pi'$))
end
```

Figure 5.6 The PSP procedure.

of Proposition 5.1 are met, and π is a solution plan. Symmetrically, when a flaw has no resolver, then π cannot be refined into a solution plan.

Let us specify the corresponding procedure as a recursive nondeterministic schema, called PSP (for Plan-Space Planning). The pseudocode of PSP is given in Figure 5.6. The following variables and procedures are used in PSP.

- *flaws* denotes the set of all flaws in π provided by procedures OpenGoals and Threats; ϕ is a particular flaw in this set.
- *resolvers* denotes the set of all possible ways to resolve the current flaw ϕ in plan π and is given by procedure Resolve. The resolver ρ is an element of this set.
- π' is a new plan obtained by refining π according to resolver ρ, using for that procedure Refine.

The PSP procedure is called initially with the initial plan π_0. Each successful recursion is a refinement of the current plan according to a given resolver. The choice of the resolver is a *nondeterministic* step. The correct implementation of the nondeterministically choose step is the following.

- When a recursive call on a refinement with the chosen resolver returns a failure, then another recursion is performed with a new resolver.
- When all resolvers have been tried unsuccessfully, then a failure is returned from that recursion level back to a previous choice point. This is equivalent to an empty set of resolvers.

Note that the selection of a flaw (select step) is a *deterministic* step. All flaws need to be solved before reaching a solution plan. The order in which flaws are processed is very important for the efficiency of the procedure, but it is unimportant for its soundness and completeness. Before getting into the properties of PSP, let us detail the four procedures it uses.

OpenGoals (π). This procedure finds all subgoals in π. These are preconditions not supported by a causal link. This procedure is efficiently implemented with an *agenda* data structure. For each new action a in π, all preconditions of a are added to the agenda; for each new causal link in π, the corresponding proposition is removed from the agenda.

Threats (π). This procedure finds every action a_k that is a threat on some causal link $\langle a_i \xrightarrow{p} a_j \rangle$. This can be done by testing all triples of actions in π, which takes $O(n^3)$, n being the current number of actions in π. Here also an incremental processing is more efficient. For each new action in π, all causal links not involving that action are tested (in $O(n^2)$). For each new causal link, all actions in π, but not those of the causal link, are tested (in $O(n)$).

Resolve (ϕ, π). This procedure finds all ways to solve a flaw ϕ. If ϕ is a subgoal for a precondition p of some action a_j, then its resolvers are either of the following:

- A causal link $\langle a_i \xrightarrow{p} a_j \rangle$ if there is an action a_i already in π whose effect can provide p. This resolver contains three elements: the causal link, the ordering constraint $(a_i \prec a_j)$ if consistent with $\prec$, and the binding constraints to unify p with the effect of a_i.
- A new action a that can provide p. This resolver contains a together with the corresponding causal link and the ordering and binding constraints, including the constraints $(a_0 \prec a \prec a_\infty)$ required for a new action. Note that there is no need here to check the consistency of these constraints with $\prec$ and B.

If ϕ is a threat on causal link $\langle a_i \xrightarrow{p} a_j \rangle$ by an action a_k that has an effect $\neg q$, and q is unifiable with p, then its resolvers are any of the following:[2]

- The constraint $(a_k \prec a_i)$, if consistent with $\prec$, i.e., ordering a_k before the causal link.
- The constraint $(a_j \prec a_k)$, if consistent with $\prec$, i.e., ordering a_k after the causal link.
- A binding constraint consistent with B that makes q and p nonunifiable.

Note that another way to address a threat is to choose for a causal link an alternate provider a_i' that has no threat. Replacing a_i with a_i' as the provider for a_j can be done through backtracking.

2. These three resolvers are called respectively *promotion*, *demotion*, and *separation*.

Refine (ρ, π). This procedure refines the partial plan π with the elements in the resolver, adding to π an ordering constraint, one or several binding constraints, a causal link, and/or a new action. This procedure is straightforward: no testing needs to be done because we have checked, while finding a resolver, that the corresponding constraints are consistent with π. Refine just has to maintain incrementally the set of subgoals in the agenda and the set of threats.

The Resolve and Refine procedures perform several query and update operations on the two sets of constraints $\prec$ and B. It is preferable to have these operations carried out by specific *constraint managers*. Let us describe them briefly.

The Ordering constraint manager handles query and update operations. The former include operations such as querying whether a constraint $(a \prec a')$ is consistent with $\prec$ and asking for all actions in A that can be ordered after some action a. The update operations consist of adding a consistent ordering constraint and removing one or several ordering constraints, which is useful for backtracking on a refinement. The alternatives for performing these operations are either of the following:

- Maintain as an explicit data structure only input constraints. In that case, updates are trivial, in $O(1)$, whereas queries require a search in $O(n^2)$, n being the number of constraints in $\prec$.
- Maintain the transitive closure of $\prec$. This makes queries easy, in $O(1)$, but requires an $O(n^2)$ propagation for each new update; a removal is performed on the input constraints plus a complete propagation.

In planning there are usually more queries than updates, hence the latter alternative is preferable.

The Binding constraint manager handles three types of constraints on variables over finite domains: (1) unary constraints $x \in D_x$, (2) binary constraints $x = y$, and (3) $x \neq y$. Types 1 and 2 are easily managed through a Union-Find algorithm in time linear to the number of query and update operations, whereas type 3 raises a general NP-complete Constraint Satisfaction Problem (CSP).[3]

Figure 5.7 depicts the global organization of the PSP procedure. The correct behavior of PSP is based on the nondeterministic step for choosing a resolver for a flaw and for backtracking over that choice, when needed, through all possible resolvers. The order in which resolvers are tried is important for the efficiency of the algorithm. This choice has to be heuristically guided.

The order in which the flaws are processed (the step select for selecting the next flaw to be resolved) is also very important for the performance of the algorithm, even if no backtracking is needed at this point. A heuristic function is again essential for guiding the search.

3. For example, the well-known NP-complete graph coloring problem is directly coded with type 3 constraints.

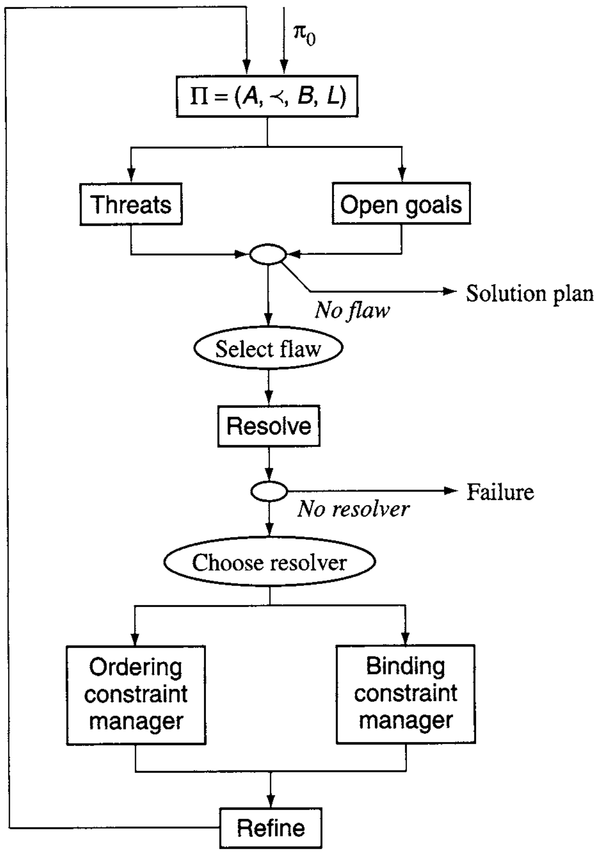

Figure 5.7 Organization of the PSP procedure.

Proposition 5.2 *The PSP procedure is sound and complete: whenever π_0 can be refined into a solution plan, PSP(π_0) returns such a plan.*

Proof Let us first prove the soundness, then the completeness.

In the initial π_0, the sets of constraints in $\prec$ and B are obviously consistent. The Resolve procedure is such that every refinement step uses a resolver that is consistent with $\prec$ and B. Consequently, the successful termination step of PSP returns a plan π that has no flaw and whose sets of constraints in $\prec$ and B are consistent. According to Proposition 5.1 (see page 93), π is a solution plan, hence PSP is sound.

In order to prove the completeness of this nondeterministic procedure, we must show that at least one of its execution traces returns a solution, when there is one. Let us prove it inductively on the length k of a solution plan.

Base step, for $k = 0$: If the empty plan is a solution to the problem, then π_0 has no flaw and PSP returns immediately this empty plan.

Induction step: Assume that PSP is complete for planning problems that have solutions of length $(k - 1)$, and assume that the problem at hand $P = (O, s_0, g)$ admits a solution $\langle a_1, \ldots, a_k \rangle$ of length k. Because a_k is relevant for the goal g, then there is at least one trace of PSP that chooses (and partially instantiates) an operator a'_k to address the flaws in π_0, such that a_k is an instance of a'_k. Let $[a'_k, a_\infty]$ be the set of goal propositions (in plan-space notation) corresponding to $\gamma^{-1}(g, a'_k)$. The next recursion of PSP takes place on a partial plan π_1 that has three actions $\{a_0, a'_k, a_\infty\}$. π_1 is equivalent to the initial plan of a problem from the state s_0 to a goal given by $[a'_k, a_\infty]$. This problem admits $\langle a_1, \ldots, a_{k-1} \rangle$ as a solution of length $(k - 1)$. By induction, the recursion of PSP on π_1 has an execution trace that finds this solution.

■

An important remark is in order. Even with the restrictive assumption of a finite transition system (assumption A0 in Section 1.5) the plan space is *not finite*. A *deterministic* implementation of the PSP procedure will maintain the completeness only if it guarantees to explore all partial plans, up to some length. It has to rely, e.g., on an iterative deepening search, such as IDA*, where a bound on the length of the sought solution is progressively increased.[4] Otherwise, the search may keep exploring deeper and deeper a single path in the search space, adding indefinitely new actions to the current partial plan and never backtracking. Note that a search with an A* algorithm is also feasible as long as the refinement cost from one partial plan to the next is not null.

5.4.2 The PoP Procedure

The PSP procedure is a generic schema that can be instantiated into several variants. Let us describe briefly one of them, algorithm PoP, which corresponds to a popular implementation for a planner in the plan space.

Figure 5.8 gives the pseudocode for PoP. It is specified as a nondeterministic recursive procedure. It has two arguments: (1) π, which is the current partial plan, and (2) *agenda*, which is a set of ordered pairs of the form $(a, p\hat{E})$, where a is an action in A and p is a precondition of a that is a subgoal. The arguments of the initial call are π_0 and an agenda containing a_∞ with all its preconditions.

PoP uses a procedure called Providers that finds all actions, either in A or in new instances of planning operators of the domain, that have an effect q unifiable with p. This set of actions is denoted *relevant* for the current goal.

There is a main difference between PSP and PoP. PSP processes the two types of flaws in a similar way: at each recursion, it selects heuristically a flaw from any type to pursue the refinement. The PoP procedure has a distinct control for subgoals and for threats. At each recursion, it first refines with respect to a subgoal, then it

4. See Appendix A.

```
PoP($\pi$, agenda)        ;; where $\pi = (A, \prec, B, L)$
    if $agenda = \emptyset$ then return($\pi$)
    select any pair $(a_j, p)$ in and remove it from agenda
    relevant $\leftarrow$ Providers($p, \pi$)
    if $relevant = \emptyset$ then return(failure)
    nondeterministically choose an action $a_i \in relevant$
    $L \leftarrow L \cup \{\langle a_i \xrightarrow{p} a_j \rangle\}$
    update $B$ with the binding constraints of this causal link
    if $a_i$ is a new action in $A$ then do:
        update $A$ with $a_i$
        update $\prec$ with $(a_i \prec a_j), (a_0 \prec a_i \prec a_\infty)$
        update agenda with all preconditions of $a_i$
    for each threat on $\langle a_i \xrightarrow{p} a_j \rangle$ or due to $a_i$ do:
        resolvers $\leftarrow$ set of resolvers for this threat
        if $resolvers = \emptyset$ then return(failure)
        nondeterministically choose a resolver in resolvers
        add that resolver to $\prec$ or to $B$
    return(PoP($\pi$, agenda))
end
```

Figure 5.8 The PoP procedure.

proceeds by solving all threats due to the resolver of that subgoal. Consequently, there are two nondeterministic steps: (1) the choice of a relevant action for solving a subgoal, and (2) the choice of an ordering constraint or a binding constraint for solving a threat.

Backtracking for these two nondeterministic steps is performed chronologically by going over all resolvers for a threat and, if none of them works, then backtracking to another resolver for the current subgoal. Note that a resolver for a threat is a single constraint to be added to either $\prec$ or B.

5.5 Extensions

The PoP procedure is a simplified version of a planner called UCPOP that implements many of the extensions of the classical representation introduced in Section 2.4. Let us mention here briefly how plan-space planning can handle some of these extensions.

Conditional Operators. Recall that a conditional operator has, in addition to normal preconditions and effects, some conditional effects that depend on whether an associated antecedent condition holds in the state s to which the operator is applied. In order to handle a conditional operator, two changes in the PoP procedure are required.

1. The Providers procedure may find an action that provides p conditionally. If this is the action chosen in *relevant*, then the antecedent on which p conditionally depends needs to be added to the *agenda*, along with the other unconditional preconditions of the action.
2. An action a_k can be a *conditional threat* on a causal link $\langle a_i \xrightarrow{p} a_j \rangle$ when the threatening effect is a conditional effect of a_k. In that case, there is another set of possible resolvers for this threat. In addition to ordering and binding constraints, here one may also solve the threat by adding to the *agenda*, as a precondition of a_k, the negation of a proposition in the antecedent condition of the threat. By making the antecedent condition false, one is sure that a_k cannot be a threat to the causal link. Note that there can be several such resolvers.

Disjunctive Preconditions. These correspond to a syntactical facility that allows one to specify in a concise way several operators into a single one. They are in principle easy to handle although they lead to an exponentially larger search space. Whenever the pair selected in the *agenda* corresponds to a disjunctive precondition, a nondeterministic choice takes place. One disjunct is chosen for which relevant providers are sought, and the procedure is pursued. The other disjuncts of the precondition are left as a possible backtrack point.

Quantified Conditional Effects. Under the assumption of a finite Σ, a quantified expression can be expanded into a finite ground expression. This expansion is made easier when each object in the domain is typed (see Section 2.4).

5.6 Plan-Space versus State-Space Planning

The search space is a critical aspect for the types of plans that can be synthesized and for the performance of a planner. Here are some of the ways in which plan-space planning compares to state-space planning.

- The state space is finite (under assumption A0), while the plan space is not, as discussed earlier.
- Intermediate states are explicit in state-space planning, while there are no explicit states in the plan space.
- A partial plan separates the choice of the actions that need to be done from how to organize them into a plan; it only keeps the intrinsic ordering and binding

constraints of the chosen actions. In state-space planning, the ordering of the sequence of actions reflects these constraints as well as the control strategy of the planner.

- The plan structure and the rationale for the plan's components are explicit in the plan space: causal links give the role of each action with respect to goals and preconditions. It is easy to explain a partial plan to a user. A sequence generated by a state-space planner is less explicit with regard to this causal structure.
- Nodes of the search space are more complex in the plan space than in states. Refinement operations for a partial plan take significantly longer to compute than state transitions or state regressions.

Despite the extra cost for each node, plan-space planning appeared for a while to be a significant improvement over state-space planning, leading to a search exploring smaller spaces for several planning domains [52, 544].

However, partial plans have an important drawback: the notion of explicit states along a plan is lost. Recent state-space planners have been able to significantly benefit from this advantage by making very efficient use of domain-specific heuristics and control knowledge. This has enabled state-space planning to scale up to very large problems. There are some attempts to generalize these heuristics and control techniques to plan-space planning [424]. However, it appears to be harder to control plan-space planners as efficiently as state-space ones because of the lack of explicit states.[5]

In summary, plan-space planners are not today competitive enough in classical planning with respect to computational efficiency. Nevertheless, plan-space planners have other advantages.

- They build partially ordered and partially instantiated plans that are more explicit and flexible for execution than plans created with state-space planners.
- They provide an open planning approach for handling several extensions to the classical framework, such as time, resources, and information gathering actions. In particular, planning with temporal and resource constraints can be introduced as natural generalization of the PSP schema (see Chapter 14).
- They allow distributed planning and multiagent planning to be addressed very naturally, since different types of plan-merging operations are easily defined and handled on the partial plan structure.

A natural question then arises: is it possible to blend state-space planning and plan-space planning into an approach that keeps the best of the two worlds?

5. For more detail, see Part III about heuristics and control issues, and in particular see Section 9.4 on heuristics for plan-space planning.

The planner called FLECS [527] provides an affirmative answer. The idea in FLECS is to combine the least commitment principle with what is called an *eager commitment strategy*. The former chooses new actions and constraints in order to solve flaws in a partial plan π, as is done PSP. The latter maintains a current state s; whenever there is in π an action a applicable to s that has no predecessor in $\prec$ except a_0, this strategy chooses to progress from s to the successor state $\gamma(s, a)$. Flaws are assessed with respect to this new current state. At each recursion, FLECS introduces a flexible choice point between eager and least commitment: it chooses nondeterministically either to progress on the current state or to solve some flaw in the current partial plan. The termination condition is an empty set of flaws. In a way, FLECS applies to plan-space planning the idea of the STRIPS procedure in state-space planning (see Exercise 5.10). However, unlike STRIPS, the procedure is sound and complete, provided that the flexible choice between eager and least commitment is a backtrack point. Several interesting heuristics for guiding this choice, and in particular an abstraction-driven heuristic, can be considered [524].

5.7 Discussion and Historical Remarks

The shift from state-space to plan-space planning is usually attributed to Sacerdoti [460], who developed a planner called NOAH [459] that has also been a seminal contribution to task reduction or Hierarchical Task Network (HTN) planning (see Chapter 11). Interestingly, the structure of a partial plan emerged progressively from another contribution to HTN planning, the Nonlin planner [503], which introduced causal links.[6] Nonlin raised the issue of *linear* versus *nonlinear* planning, which remained for a while an important and confusing debate issue in the field. The *linear* adjective for a planner referred confusingly in different papers either to the structure of the planner's current set of actions (a sequence instead of a partial order) or to a search strategy that addresses one goal after the previous one has been completely solved.

The issue of interacting goals in a planning problem and how to handle them efficiently has been a motivating concern for the study of the plan space. Starting from [502] and the so-called Sussman anomaly (see Example 4.3), several authors [51, 164, 267, 291] discussed this issue.

In the context of plan-space planning, Korf [336] introduced a distinction between problems with fully independent goals, serializable goals, and arbitrarily interacting goals. The first category is the easiest to handle. In the second category, there is an ordering constraint for solving the goals without violating the previously solved ones. If the goals are addressed in this correct order, then the planning complexity grows linearly in the number of goals. This goal dependence hierarchy is further refined in the context of plan-space planning by Barrett and

6. See [297] for a comparative analysis of plan-space and HTN planning.

Weld [52], where the authors introduce a planner-dependent notion of trivially and laboriously serializable goals. According to their analysis, plan-space planners have the advantage of more often leading to trivial serializable goals that are easily solved.

The truth criterion in a partial plan has been another major debate in the field. In state-space planning, it is trivial to check whether some condition is true or not in some current state. But plan-space planning does not keep explicit states. Hence it is less easy to verify whether a proposition is true before or after the execution of an action in a partially ordered and partially instantiated set of actions. The so-called Modal Truth Criterion (MTC) [120] provided a necessary and sufficient condition for the truth of a proposition at some point in a partial plan π. A planner called TWEAK relied at each recursion on the evaluation of this MTC criterion for synthesizing a plan in the plan space. It was shown that if π contains actions with conditional effects, then the evaluation of the MTC is an NP-hard problem. This complexity result led to a belief that plan-space planning with extended representation is impractical. However, this belief is incorrect because planning does not require a necessary and sufficient truth condition. It only has to enforce a sufficient truth condition, which basically corresponds in PSP to the identification and resolution of flaws, performed in polynomial time. A detailed analysis of the MTC in planning appears in Kambhampati and Nau [303].

The UCPOP planner of Penberthy and Weld [436, 544] has been a major contribution to plan-space planning. It builds on the advances brought by SNLP [381], an improved formalization of TWEAK, to propose a proven sound and complete planner able to handle most of the extensions to the classical representation introduced in the Action Description Language (ADL) of Pednault [433, 434]. The well-documented open-source Lisp implementation of UCPOP [50] offers several enhancements such as domain axioms and control rules. The latter even incorporate some of the learning techniques developed in the state-space planner PRODIGY [393, 524] for acquiring control knowledge.

The work on UCPOP opened the way to several other extensions such as handling incomplete information and sensing actions [183, 236, 438] and managing extended goals with protection conditions that guarantee a plan meeting some safety requirements [546] (see Part V). Issues such as the effects of domain features on the performance of the planner [327] and the role of ground actions (i.e., an early commitment as opposed to a late commitment strategy) [563] have been studied. Recent implementations, such as HCPOP [217], offer quite effective search control and pruning capabilities. The study of the relationship between state space and plan space exemplified in the FLECS planner [527] follows on several algorithmic contributions such as those by Minton *et al.* [392, 394] and on studies that relate the performance of PSP to specific features of the planning domain [327]. The work of Kambhampati *et al.* [296, 302] introduces a much wider perspective and a nice formalization that takes into account many design issues such as multiple contributors to a goal or systematicity [295], i.e., whether a planner is nonredundant and does not visit a partial plan in the space more than once.

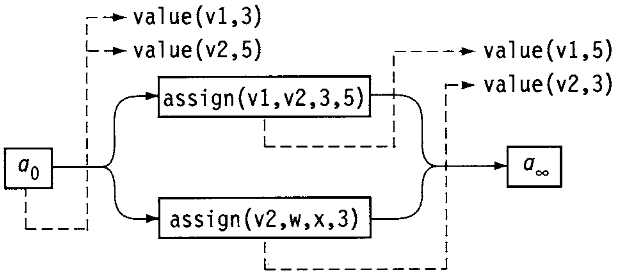

Figure 5.9 Partial plan for interchanging variables.

5.8 Exercises

5.1 Figure 5.9 shows a partial plan generated by PSP for the variable-interchange problem described in Exercise 4.1.

(a) How many threats are there?

(b) How many children (immediate successors) would this partial plan have?

(c) How many different solution plans can be reached from this partial plan?

(d) How many different nonredundant solution plans can be reached from this partial plan?

(e) If we start PSP running from this plan, can it ever find any redundant solutions?

(f) Trace the operation of PSP starting from this plan, along whichever execution trace finds the solution that has the smallest number of actions.

5.2 Trace the PSP procedure step-by-step on Example 5.1 (see page 86), from Figure 5.2 to Figure 5.5.

5.3 Trace the operation of PSP on the Sussman anomaly (Example 4.3; see page 86).

5.4 Trace the PSP procedure on the problem that has a single partially instantiated goal in(*c*,p2) with an initial state similar to that of Figure 5.1 except that location l1 has two piles p0 and p1, p0 has a single container c0, and p1 has a container c1.

5.5 Let $\mathcal{P}$ be a planning problem in which no operator has any negative effects.

(a) What part(s) of PSP will not be needed to solve $\mathcal{P}$?

(b) Suppose we run PSP deterministically, with a best-first control strategy. When, if at all, will PSP have to backtrack?

5.6 Consider the following "painting problem." We have a can of red paint (c1), a can of blue paint (c2), two paint brushes (r1, r2), and four unpainted blocks

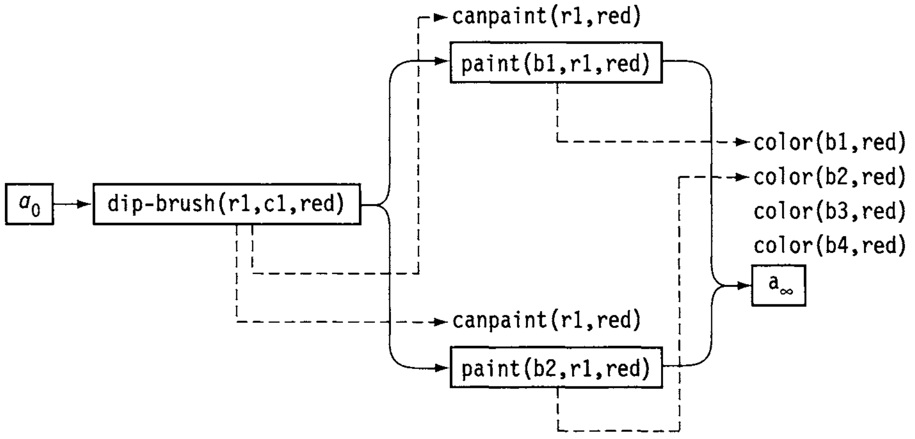

Figure 5.10 Partial plan for painting blocks.

(b1, b2, b3, b4). We want to make b1 and b2 red and make b3 and b4 blue. Here is a classical formulation of the problem:

s_0 = {can(c1), can(c2), color(c1,red), color(c2,blue), brush(r1), brush(r2), dry(r1), dry(r2), block(b1), block(b2), dry(b1), dry(b2), block(b3), block(b4), dry(b3), dry(b4)}

g = {color(b1,red), color(b2,red), color(b3,blue), color(b4,blue)}

dip-brush(r, c, k)

precond: brush(r), can(c), color(c, k)

effects: ¬dry(r), canpaint(r, k)

paint(b, r, k)

precond: block(b), brush(r), canpaint(r, k)

effects: ¬dry(b), color(b, k), ¬canpaint(r, k)

(a) In the paint operator, what is the purpose of the effect ¬canpaint(r,k)?

(b) Starting from the initial state, will PSP ever generate the partial plan shown in Figure 5.10? Explain why or why not.

(c) What threats are in the partial plan?

(d) Starting from the partial plan, resolve all of the open goals without resolving any threats. What threats are in the plan now?

(e) If we start PSP running with this plan as input, will PSP generate any successors? Explain why or why not.

(f) If we start PSP running with this plan as input, will PSP find a solution? Explain why or why not.

5.7 Dan wants to wash his clothes with a washing machine wm, wash his dishes in a dishwasher dw, and bathe in a bathtub bt. The water supply doesn't have enough pressure to do more than one of these activities at once. Here is a classical representation of the problem:

Initial state: status(dw,ready), status(wm,ready), status(bt,ready),
clean(dan,0), clean(clothes,0), clean(dishes,0),
loc(dishes,dw), loc(clothes,wm), loc(dan,bt), use(water,0)

Goal formula: clean(clothes,1), clean(dishes,1), clean(dan,1)

Operators:

```
start-fill(x)                                    end-fill(x)
 precond:  status(x,ready), use(water,0)          precond:  status(x,fill)
 effects:  status(x,fill),                        effects:  status(x,full),
           use(water,1)                                     use(water,0)

start-wash(x)                  end-wash(x, y)
 precond:  status(x,full)       precond:  status(x,wash)
 effects:  status(x,wash)       effects:  status(x,ready), clean(y,1)
```

(a) Let π_1 be the partial plan shown in Figure 5.11. What threats are in π_1?

(b) What flaws are in π_1 other than threats?

(c) How many different solutions can PSP find if we start it out with plan π_1 and do not allow it to add any new actions to the plan? Explain your answer.

(d) How many different solutions can PSP find if we do allow it to add new actions to π_1? Explain your answer.

5.8 Let $P = (O, s_0, g)$ and $P' = (O, s_0, g')$ be the statements of two planning problems having the same operators and initial state. Let B and B' be the search spaces for PSP on P and P', respectively.

(a) If $g \subseteq g'$, then is $B \subseteq B'$?

(b) If $B \subseteq B'$, then is $g \subseteq g'$?

(c) Under what conditions, if any, can we guarantee that B is finite?

(d) How does your answer to part (c) change if we run PSP deterministically with a breadth-first control strategy?

(e) How does your answer to part (c) change if we run PSP deterministically with a depth-first control strategy?

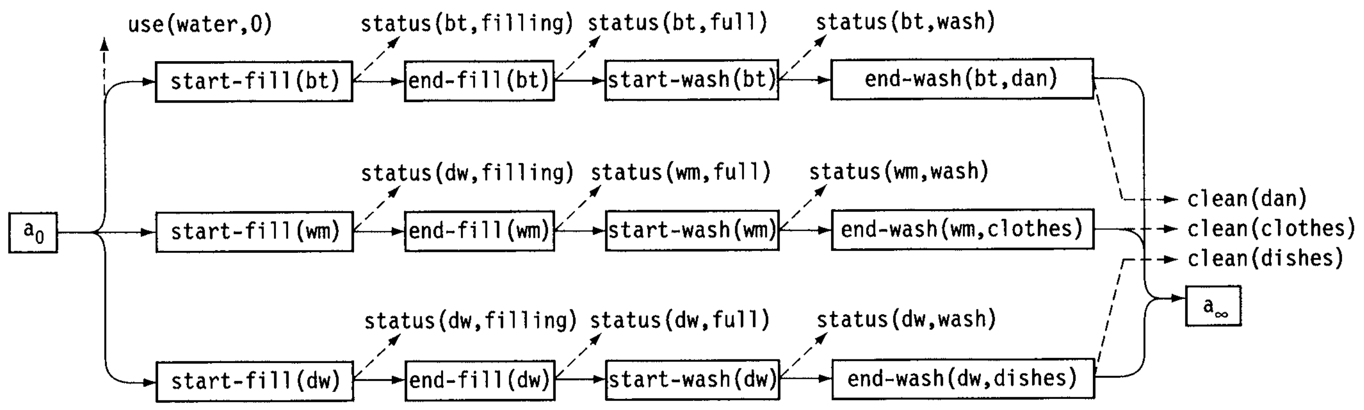

Figure 5.11 Partial plan for washing.

5.9 If $\mathcal{L}$ is a planning language in which the number of different ground atoms is n, then for any planning domain whose language is $\mathcal{L}$, the number of possible different states is at most 2^n. Use this upper bound on the number of states to create a simple loop-detection mechanism for PSP. Why is this loop-detection mechanism not very useful in practice?

5.10 Discuss the commonalities and differences of FLECS and STRIPS. Why is the former complete while the latter is not?

5.11 Download one of the public-domain plan-space planners, e.g., UCPOP[7] or HCPOP [217]. Test the full DWR example on several domains and problems with an increasing number of locations, robots, and containers. Discuss the practical range of applicability of this planner for the DWR application.

7. http://www.cs.washington.edu/ai/ucpop.html.