15 Planning

In this chapter, you learn about two distinct approaches to planning a sequence of actions to achieve some goal. One way, the STRIPS approach, uses *if-add-delete operators* to work on a single collection of assertions. The other way, using logic, requires the introduction of *situation variables* and *frame axioms* that deal with linked collections of logical statements.

Both approaches are presented in this chapter; their differences and complementary strengths are highlighted through juxtaposition. Fortunately, however, you can study them independently, if you wish.

By way of illustration, you see how to plan simple blocks-world movement sequences.

First, you learn about the much more transparent STRIPS approach, and how you can use notions like *establishes* and *threatens* to prevent potentially interfering actions from destroying each other's work. Next, you learn that logic requires you to use *frame axioms* whenever you want to deal with a changing world.

PLANNING USING IF-ADD-DELETE OPERATORS

In the physical world, a **plan** is a prescription for a sequence of actions that, if followed, will change the relations among objects so as to achieve a desired goal. One way to represent a plan is by way of a sequence of assertion additions and deletions that reflect physical movements. Thus,

the deletion of ON(A, C) and addition of ON(A, B) means move object A from object C to object B.

One way to create a plan is to search for a sequence of operators that lead from the assertions that describe the initial state of the world to assertions that describe your goal. In this section, you learn how to perform such a search.

Operators Specify Add Lists and Delete Lists

Figure 15.1 shows an example of an initial situation and a goal situation in which block A is on block B, and block B is on block C. A little more precisely, part of the initial situation description consists of the following axioms:

ON(A, C)
ON(C, Table)
ON(D, B)
ON(B, Table)

In addition, you need a way of expressing that two of the objects support nothing. One way is to introduce the idea that a block is clear, and hence is capable of having something placed on it only if there is nothing on it to start with. In contrast to the blocks world discussed in Chapter 3, here no block can directly support more than one other block. The axioms are

CLEAR(A)
CLEAR(D)

The goal is to arrange for the following to be part of the situation description:

ON(A, B)
ON(B, C)

For this particular example, you are to assume that nothing can be moved if it is supporting something; said another way, you cannot move x unless x's top is clear.

To plan, you need to capture what the various actions accomplish when they are actually deployed. One approach is to specify each action in terms of when it can be used, what new assertions it makes true, and what previous assertions it withdraws. These specifications can be packaged into if–add–delete rules, introduced previously in Chapter 7. In this chapter, some labels change in honor of the language that is used in the planning literature: In the context of planning, if–add–delete rules are called **operators** and the *if* conditions of each operator are called its **prerequisites**. Here is the description of an operator that describes the action of moving a block:

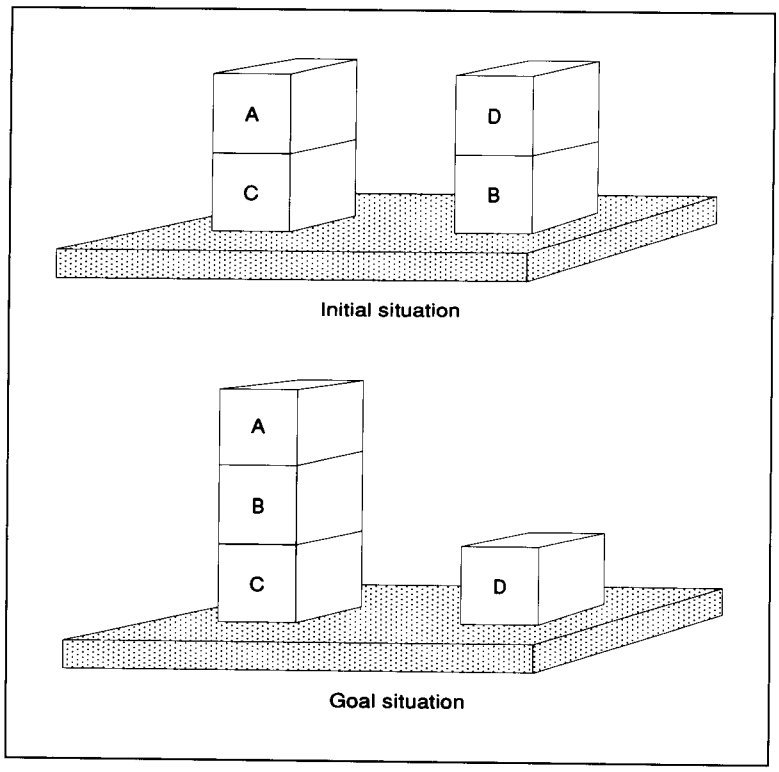

Figure 15.1 An initial situation and a goal situation.

Operator 1: Move block x from block y to block z

If:	ON(x, y)
	CLEAR(x)
	CLEAR(z)
Add list	ON(x, z)
	CLEAR(y)
Delete list:	ON(x, y)
	CLEAR(z)

Now suppose you instantiate the move-block-from-block-to-block operator with x bound to block A, y to block C, and z to block D:

Operator 1: Move block A from block C to block D

If:	ON(A, C)
	CLEAR(A)
	CLEAR(D)
Add list	ON(A, D)
	CLEAR(C)
Delete list:	ON(A, C)
	CLEAR(D)

Then, if you deploy this instantiated operator in the example's initial situation, you will alter the situation description such that it appears as follows, with two deleted assertions struck out and two new assertions at the end:

~~ON(A, C)~~
ON(C, Table)
ON(D, B)
ON(B, Table)
CLEAR(A)
~~CLEAR(D)~~
ON(A, D)
CLEAR(C)

You can, of course, move a block onto the table even though there are other blocks on the table already. Also, the table does not become clear when a block is moved off it. Accordingly, you need two extra operators specialized for movements in which the table is involved:

Operator 2: Move block x from block y to Table

If:	ON(x, y)
	CLEAR(x)
Add list	ON$(x,$ Table$)$
	CLEAR(y)
Delete list:	ON(x, y)

Operator 3: Move block x from Table to block z

If:	ON$(x,$ Table$)$
	CLEAR(x)
	CLEAR(z)
Add list	ON(x, z)
Delete list:	ON$(x,$ Table$)$
	CLEAR(z)

You Can Plan by Searching for a Satisfactory Sequence of Operators

Once you have a problem description, planning is, in a certain silly sense, simple. You need only to try everything, using breadth-first search, where *try everything* means try every operator, instantiated in every possible way, as long as the operators' prerequisites are satisfied. If you use this strategy for the example, you discover a satisfactory plan after four moves, as demonstrated by the breadth-first search tree shown in figure 15.2.

Unfortunately, this approach forces you to fight a battle against exponential tree growth—a battle you would probably lose for any problem of practical size. For the example, you can manage, because there is a maximum of 12 branches under any node and because there is a solution after

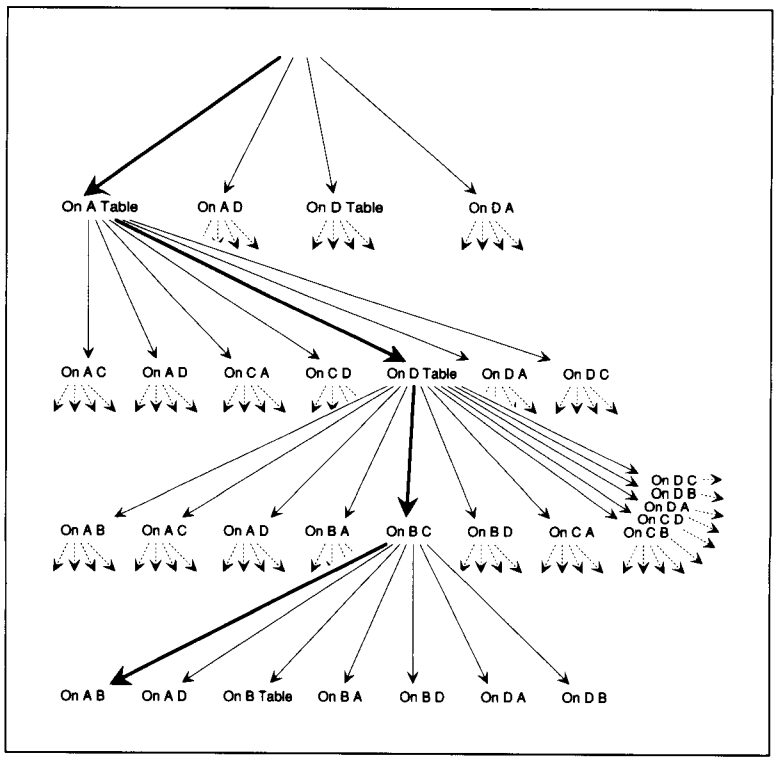

Figure 15.2 A simple, but bad approach to planning. You try every possible operator sequence, using breadth-first search. The heavy line marks a plan that reaches the goal.

just four moves, but if you were to scatter, say, eight irrelevant two-block towers on the table, then there would be 100 alternatives for the first move alone.

Plainly, you need to be smarter about how to conduct a search for a suitable operator sequence.

Backward Chaining Can Reduce Effort

Usually, the goal is described by only a few assertions that must hold for a problem to be solved, whereas any full description of a situation involves many more assertions. Accordingly, you may want to chain backward through operators, rather than forward. To backward chain, you look for an operator that, when instantiated, has an *add* pattern that matches your goal. Then, if that operator's prerequisites are not satisfied, you backward chain through other operators until you reach assertions that describe the initial state of the world.

For example, consider that part of the example problem that requires block A to be on block B. For the moment, forget about getting block B on block C.

At the top of figure 15.3, you see the initial state on the left, the single goal assertion on the right, and an attempt to start bridging the

gap between the two by moving block A from block C to block B. More specifically, an Establishes link extends from the ON(A, B) assertion in the add list of the instantiated movement operator to the ON(A, B) assertion in the goal description.

Moving block A from block C to block B is not the only choice, of course; it is just one of three possibilities, because block A could, in principle, come from block D or the table. If moving it to its goal position from block C does not work out, those other possibilities could be checked by breadth-first search.

Given that moving block A from block C to block B is the choice to be considered, two prerequisites have to be satisfied: both block A and block B must be clear. One way to clear block B is to move block D to the table. As shown in the middle of figure 15.3, this move links together two instantiated operators; an Establishes link joins the CLEAR(B) assertion in the add list of operator 2's instantiation to the CLEAR(B) prerequisite in operator 1's instantiation.

Again, satisfying the CLEAR(B) prerequisite using an instantiated operator that moves block D to the table is just one of many choices. In principle, any other block could be in the way.

Note that the Establishes link means that the operator that establishes an assertion must appear in the final plan before the operator that needs the assertion. The establisher does not need to appear right before the establishee, however. There may be other operators in between, as long as none of those interposed operators add or delete the established assertion.

Once block B is clear, no other operator is needed, for the remaining prerequisites of both operators are satisfied in the initial situation, as shown at the bottom of figure 15.3. Once again, this way of dealing with those prerequisites is not the only one. It is the direct way, and it is a reasonable way under the circumstances, but in principle, you could, for example, uselessly stall by moving block D on and off block B many times to create a longer bridge between the initial state and the movement of block A to block B.

Alternative choices have been mentioned frequently in this discussion to emphasize that search is involved. What you see in figure 15.3 is just one path through a search tree.

In general, a partial path may involve choices that cannot possibly lead to a workable plan because of devastating operator interactions. To see why, consider the full example, with a goal involving two assertions, ON(A, B) and ON(B, C), rather than only ON(A, B). If you temporarily ignored the second of the goal assertions, you could certainly make the choices that previously led to the placement of block A on block B, leading to the partial plan shown at the top of figure 15.4.

With the problem of moving A onto B solved, you could then attempt to enable the movement of B onto C by moving B from the table onto C.

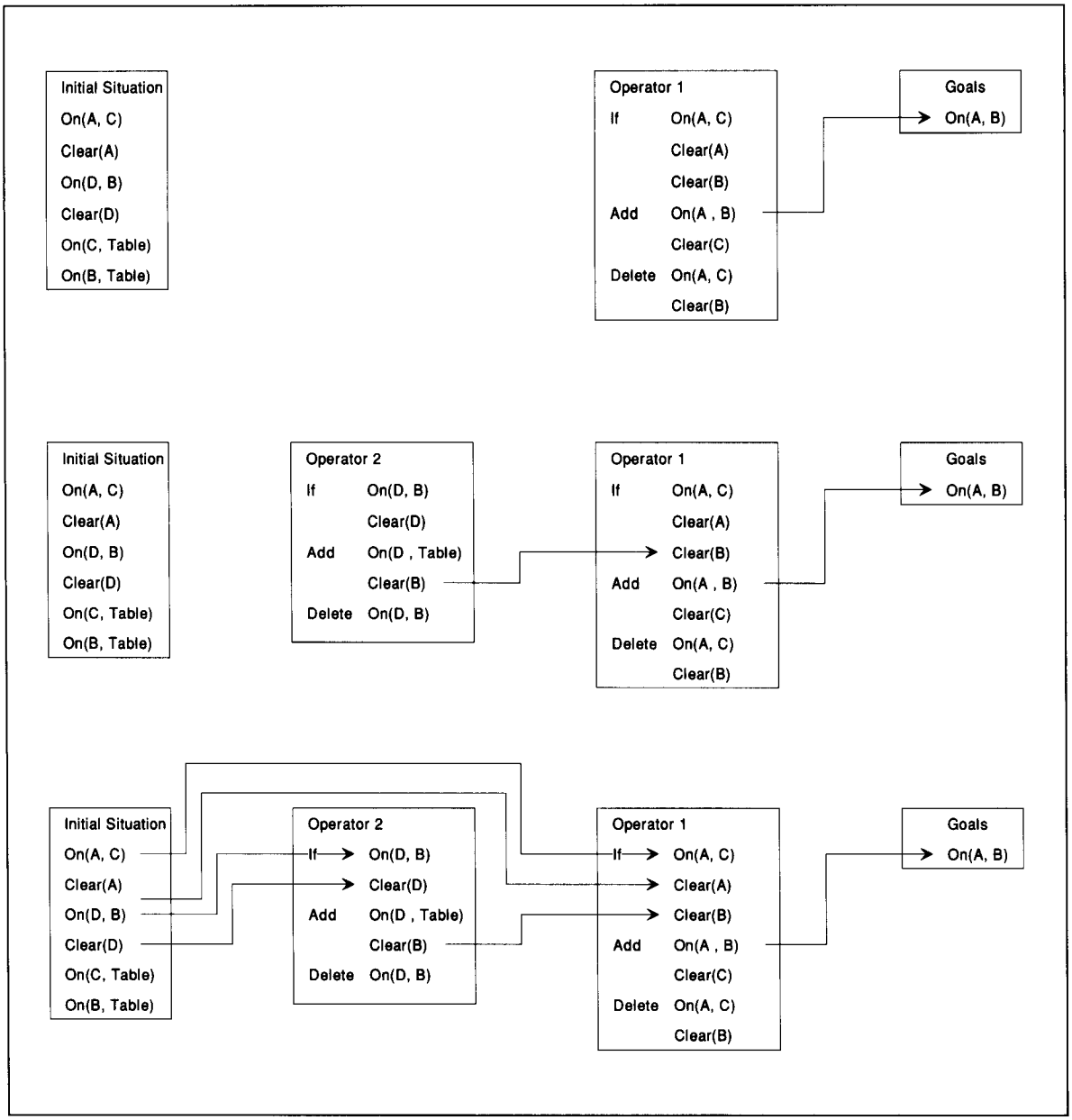

Figure 15.3 Three steps toward constructing a plan for achieving a simple goal using backward chaining. All links shown are Establishes links. Most of them tie an addition in one instantiated operator to a prerequisite in another instantiated operator, thus creating an ordering constraint between the two operators. Other Establishes links are connected to initial-situation assertions or goal assertions.

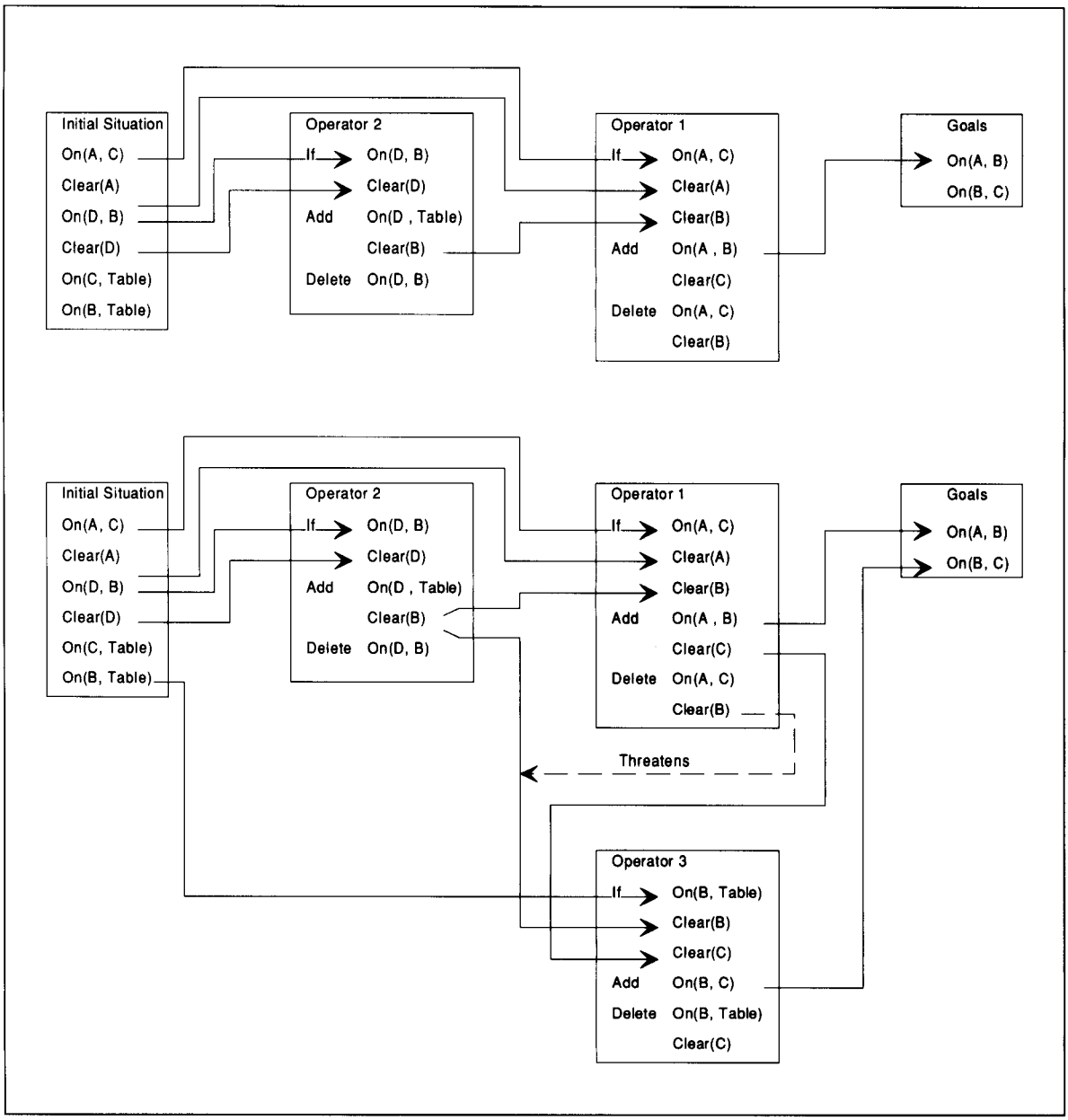

Figure 15.4 Construction of a plan for achieving two goal assertions using backward chaining. All links shown are Establishes links. The top portion deals with moving block A onto block B; the bottom portion adds an operation intended to move block B onto block C. The plan is flawed because operator 1 interferes with an Establishes link between operator 2 and operator 3.

Then, to satisfy the prerequisites required to move block B from the table onto block C, you have to be sure that both blocks are clear. At first glance, looking only at the diagram and forgetting about real blocks, you might think you could satisfy these prerequisites by taking advantage of the initial situation and the additions to it made by other operators. Using Establishes links to establish order, you might think you could move block D onto the table, move block A onto block B, and finally move block B onto block C. But too bad for first-glance looks: Moving block A onto block B interferes with moving block B anywhere.

Thus, you are faced with a major problem. As you try to add an assertion demanded by the prerequisites of one operator, you may withdraw an assertion that you added previously to satisfy a goal or to establish a prerequisite for another operator. Clearly, you need a method for detecting such interference as soon as possible, so that you can terminate useless partial plans without further work.

Impossible Plans Can Be Detected

Fortunately, it is easy to identify potential problems in the plan itself, without appealing to your knowledge of blocks. You simply monitor each Establishes link, checking for operators that could invalidate the link. When you find such a link-invalidating operator, you impose an additional ordering constraint that prevents the operator from doing any damage.

In the bottom of figure 15.4, for example, there is an Establishes link between operator 2 and operator 3, because operator 2 adds CLEAR(B) and operator 3 needs CLEAR(B). Operator 1 deletes CLEAR(B), however, and if operator 1 were allowed to occur after operator 2 and before operator 3, then operator 1 would invalidate the Establishes link. Accordingly, operator 1 is said to be a **threat** to the link.

To contain such threats, you need only to introduce an additional ordering constraint: The threatening operator must appear before the establisher or after the establishee, and may not appear between them. In the example, this constraint means that operator 1 must appear in the plan either before operator 2 or after operator 3.

Unfortunately, neither choice works, as shown by figure 15.5, in which implied Before links replace Establishes links, creating a kind of simple **ordering net**. As shown in the left portion of the figure, operator 1 cannot appear before operator 2, because the Establishes link from operator 2 to operator 1 has already established that operator 2 appears before operator 1. Similarly, as shown in the right portion of the figure, operator 3 cannot appear before operator 1, because an Establishes link from operator 1 to operator 3 has already established that operator 1 appears before operator 3. The only possible conclusion is that the choices made so far cannot lead to a workable plan.

The possibility of interference is not the certainty of interference, however. Consider figure 15.6. In the top portion, you see the initial state on

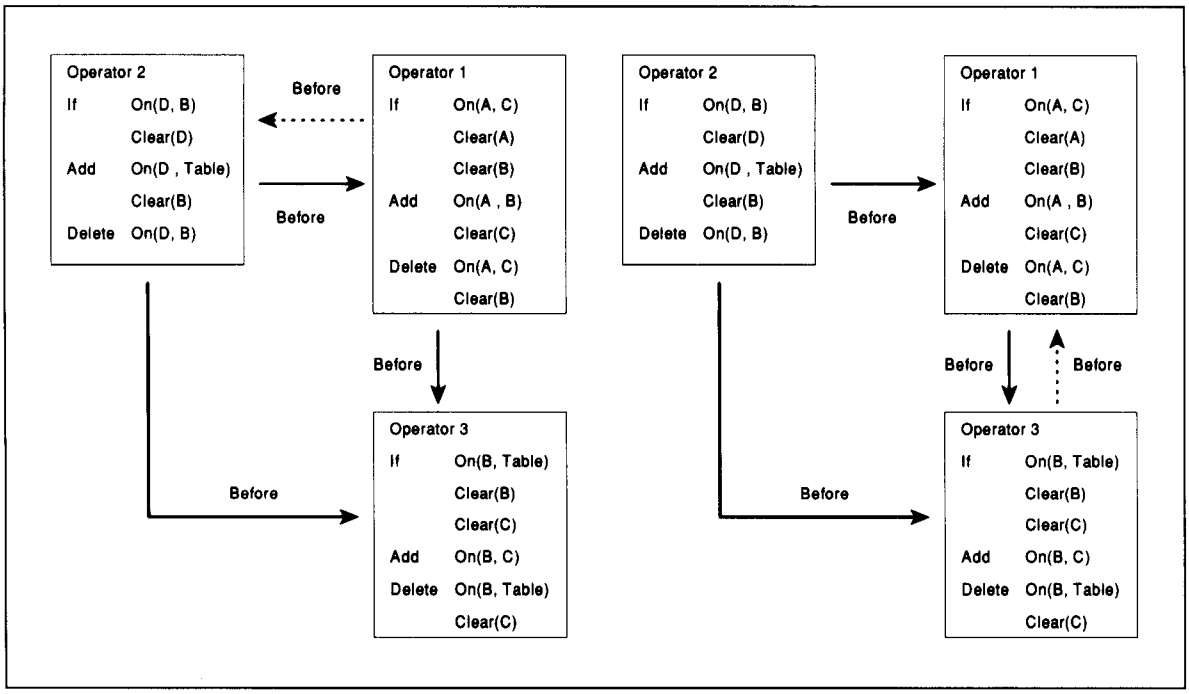

Figure 15.5 You can protect a threatened Establishes link by installing an additional Before link. Here, however, operator 1 cannot appear before operator 2, as in the left portion, or after operator 3, as in the right portion, without creating a Before loop.

the left, the goal assertions on the right. This time, the choice is to move block A from the table to block B, instead of from its initial resting place on block C. In the bottom portion, you see how the prerequisites of the operator that moves block A from the table can be satisfied by the assertions in the initial state augmented by assertions placed by one instantiation of operator 2 that moves block A from block C to the table, and by another instantiation of operator 2 that moves block D from block B to the table. At this point, there are no threats.

Although there are no threats, there is a question of how to get block B onto block C. One choice is to instantiate operator 3 again, as shown in the top portion of figure 15.7, thereby moving block B from the table, where it lies initially. Following that choice, you need only to decide how to satisfy the prerequisites that require block B to be on the table and both block B and block C to have clear tops. One set of choices is shown in the bottom portion of figure 15.7.

Note, however, that there is, once again, a threat to which you must attend. Operator 3a, the first of the two instantiations of operator 3, deletes the CLEAR(B) assertion that operator 2a adds and that operator 3b needs. Thus, operator 3a threatens the Establishes link between operator 2a and operator 3b. Evidently, operator 3a must be placed before operator 2a or after operator 3b. Plainly, operator 3a cannot be placed before operator 2a, because operator 2a establishes one of operator 3a's prerequisites.

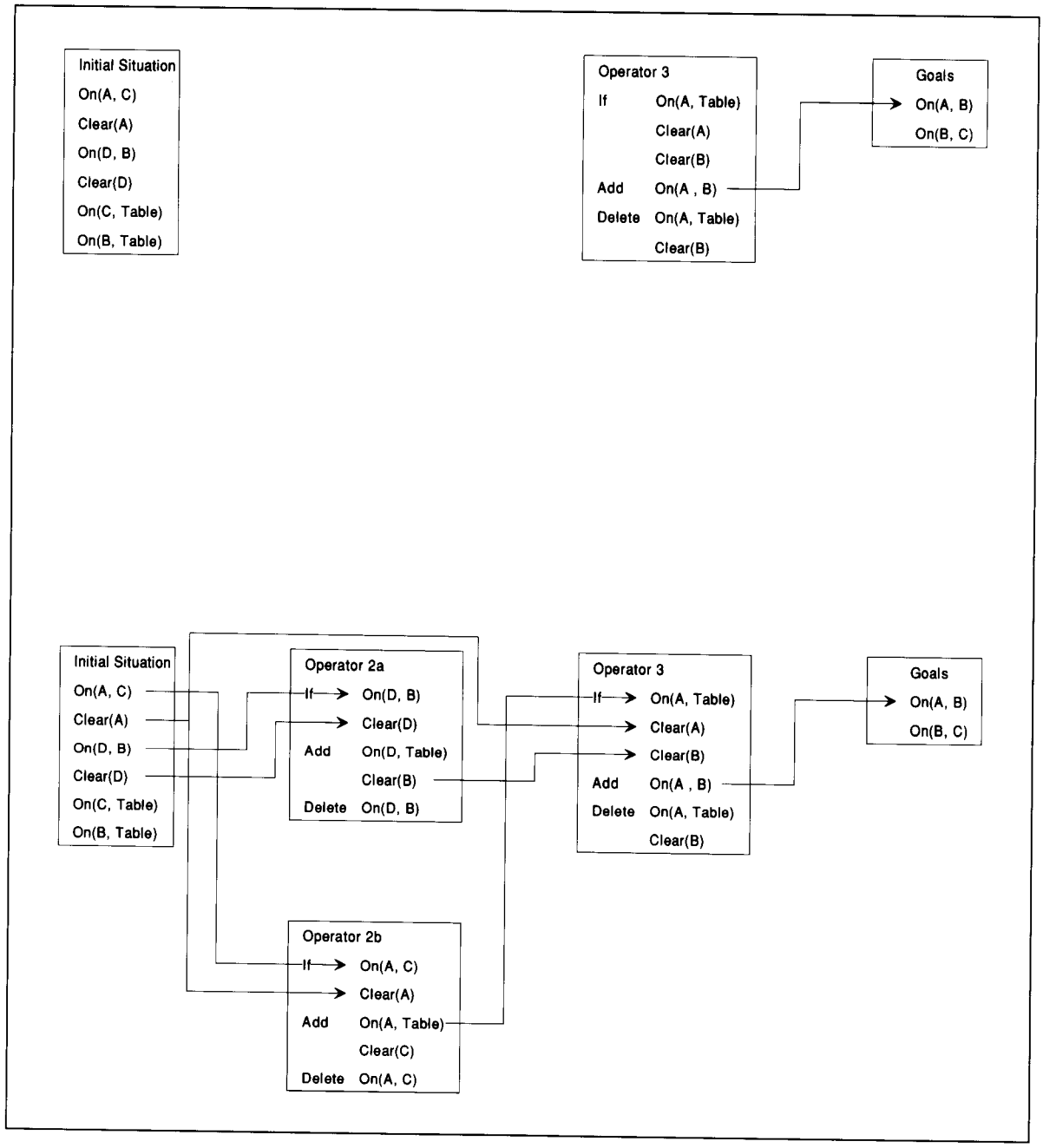

Figure 15.6 In the top portion, planning begins with an instantiation of operator 3; in the bottom, two instantiations of operator 2 are added to deal with operator 3's prerequisites.

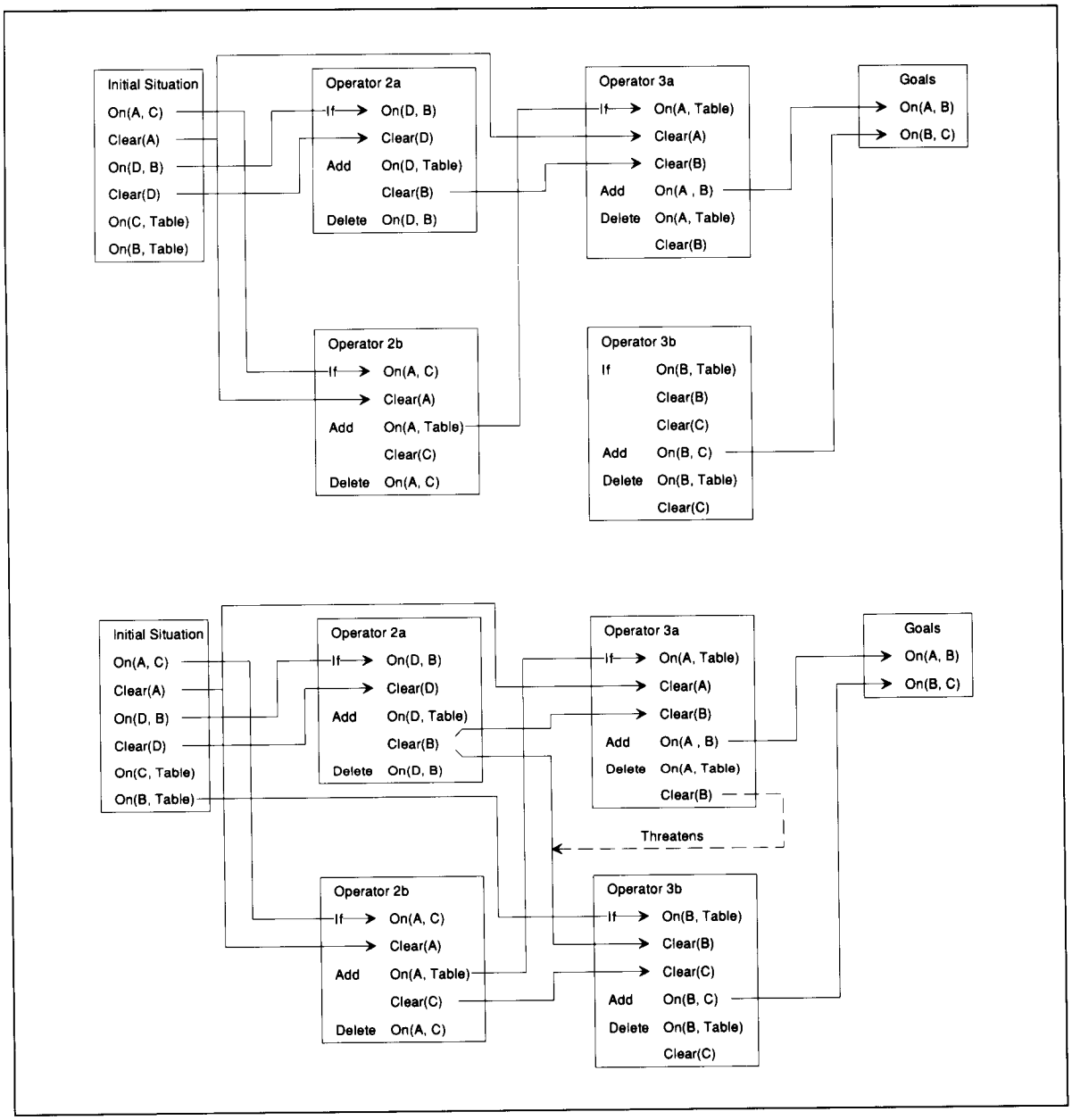

Figure 15.7 Completion of a plan for achieving two goal assertions. The top portion shows how work might start on the problem of getting block B onto block C. The bottom portion shows how the plan might be completed by addition of operator 3b, which is a second instantiation of operator 3. Because of the addition, operator 3a threatens the Establishes link that extends from operator 2a to operator 3b.

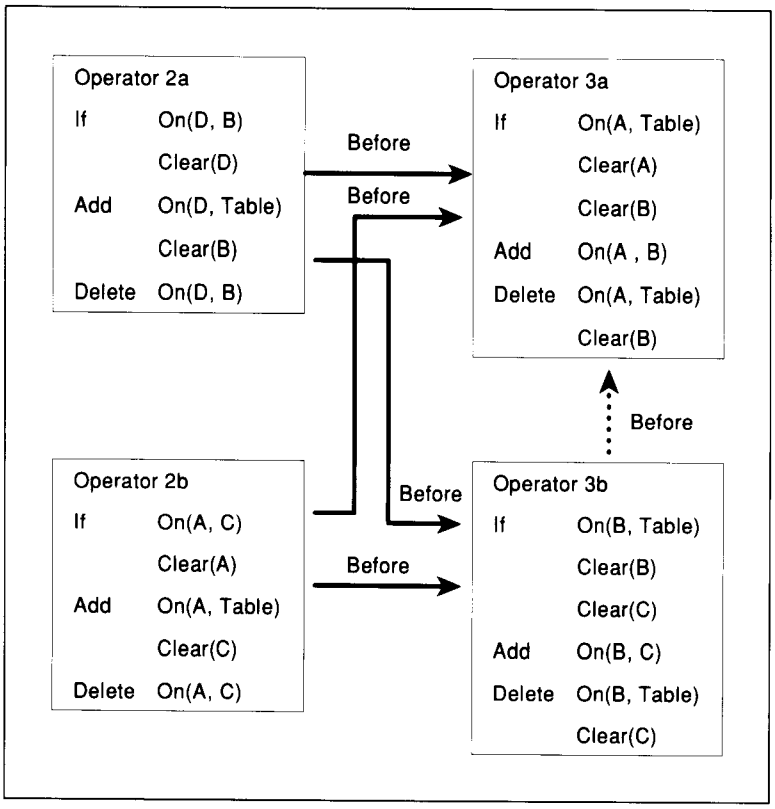

Figure 15.8 You can protect the threatened Establishes link shown in figure 15.7 by installing an additional Before link. Here, a new Before link forcing operator 3b to occur before operator 3a prevents trouble.

Fortunately, as shown in figure 15.8, operator 3a can be after operator 3b without creating any impossible chain of Before links.

The plan shown in the bottom of figure 15.8 is called a **complete plan** because all goal assertions are asserted, all operator prerequisites are satisfied, and the implied Before links form no loops. A **partial plan** is any other plan in which the implied Before links form no loops.

Note that a complete plan does not completely specify an exact ordering of steps. Block A can be moved to the table before block D, or vice versa, whichever you like:

Plan 1	Plan 2
Move A to the table	Move D to the table
Move D to the table	Move A to the table
Move B to C	Move B to C
Move A to B	Move A to B

At this point, you might wonder whether there is always at least one way to write out a sequence of operators consistent with the ordering constraints implied by a complete plan's Establishes links, forming a **linear plan**. Be comforted: A version of the topological sorting procedure, introduced in Chapter 9, always can produce at least one linear plan from each complete plan.

In summary, the procedure for constructing a plan is a search procedure that extends partial plans until one is complete:

To construct a plan,

- ▷ Extend a partial plan containing no operators until you discover a satisfactory complete plan.

To extend a partial plan,

- ▷ If the plan has a Before loop, announce failure.
- ▷ If the plan is a complete plan, announce success.
  - ▷ If there is an operator that threatens an Establishes link between operators o_1 and o_2, call the threat o_t, and do one of the following:
    - ▷ Place a Before link from o_t to o_1 and extend the plan.
    - ▷ Place a Before link from o_2 to o_t and extend the plan.
  - ▷ Otherwise, pick an unsatisfied prerequisite, and do one of the following:
    - ▷ Find an existing operator that adds the unsatisfied prerequisite, install an Establishes link, and extend the plan.
    - ▷ Instantiate and add an operator that adds the unsatisfied prerequisite and extend the plan.

Partial Instantiation Can Help Reduce Effort Too

Even with backward chaining, extra blocks scattered here and there are a nuisance. In the example, block A does not need to be parked on the table while you arrange for block B to be on block C. It could be parked on any block other than block B or block C, thus increasing the branching involved in your search with each addition of a new block.

One way to cope with this sort of branching is to instantiate operators only insofar as necessary. To see how you do that, suppose that you have one additional block on the table, block E. Further suppose that you decide to move block A from a parking block to block B, rather than from the table. This decision can lead to another decision between two full instantiations of operator 1: