# This Is a Publication of
# The American Association for Artificial Intelligence

# Refinement Planning as a Unifying Framework for Plan Synthesis

### *Subbarao Kambhampati*

■ Planning—the ability to synthesize a course of action to achieve desired goals—is an important part of intelligent agency and has thus received significant attention within AI for more than 30 years. Work on efficient planning algorithms still continues to be a hot topic for research in AI and has led to several exciting developments in the past few years. This article provides a tutorial introduction to all the algorithms and approaches to the planning problem in AI. To fulfill this ambitious objective, I introduce a generalized approach to plan synthesis called refinement planning and show that in its various guises, refinement planning subsumes most of the algorithms that have been, or are being, developed. It is hoped that this unifying overview provides the reader with a brand-name–free appreciation of the essential issues in planning.

**P***lanning* is the problem of synthesizing a course of action that, when executed, takes an agent from a given initial state to a desired goal state. Automating plan synthesis has been an important research goal in AI for more than 30 years. A large variety of algorithms, with differing empirical trade-offs, have been developed over this period. Research in this area is far from complete, with many exciting new algorithms continuing to emerge in recent years.

To a student of planning literature, the welter of ideas and algorithms for plan synthesis can at first be bewildering. I remedy this situation by providing a unified overview of the approaches for plan synthesis. I describe a general and powerful plan-synthesis paradigm called *refinement planning* and show that a majority of the traditional, as well as the newer, plan-synthesis approaches are special cases of this paradigm. It is my hope that this unifying treatment separates the essential trade-offs from the peripheral ones (for example, brand-name affiliations) and provides the reader with a firm understanding of the existing work as well as a feel for the important open research questions.

In this article, I briefly discuss the (classical) planning problem in AI and provide a chronology of the many existing approaches. I then propose refinement planning as a way of unifying all these approaches and present the formal framework for refinement planning. Next, I describe the refinement strategies that correspond to existing planners. I then discuss the trade-offs among different refinement-planning algorithms. Finally, I describe some promising new directions for scaling up refinement-planning algorithms.

## Planning and Classical Planning

Intelligent agency involves controlling the evolution of external environments in desirable ways. Planning provides a way in which the agent can maximize its chances of achieving this control. Informally, a plan can be seen as a course of action that the agent decides on based on its overall goals, information about the current state of the environment, and the dynamics of the evolution of its environment (figure 1).

The complexity of plan synthesis depends on a variety of properties of the environment and the agent, including whether (1) the environment evolves only in response to the agent's actions or also independently, (2) the state of the environment is observable or partially hidden, (3) the sensors of the agent are powerful enough to perceive the state of the environment, and (4) the agent's actions have deterministic or stochastic effects on the state
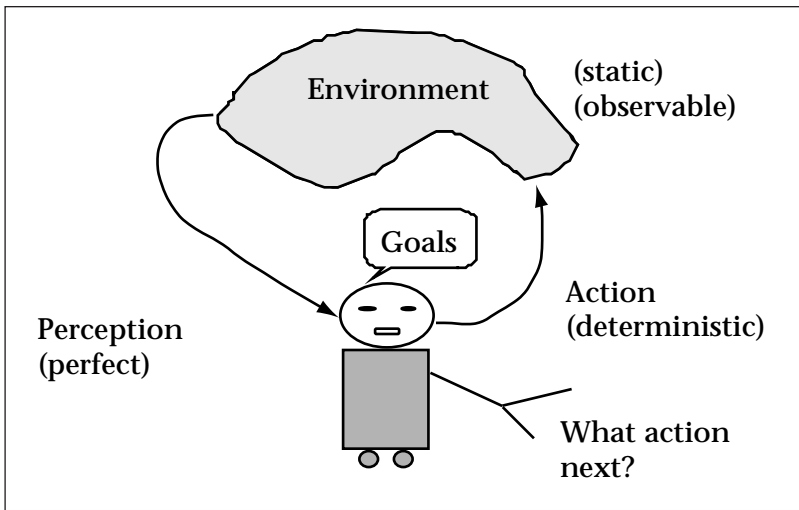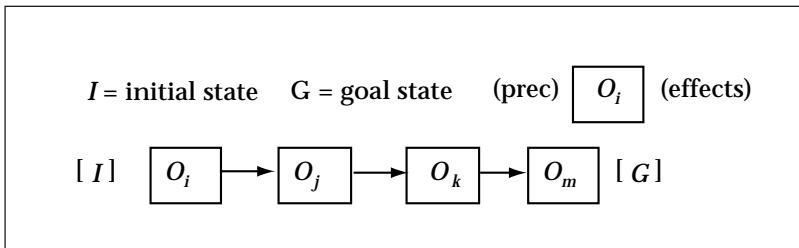
*Figure 1. Role of Planning in Intelligent Agency.*
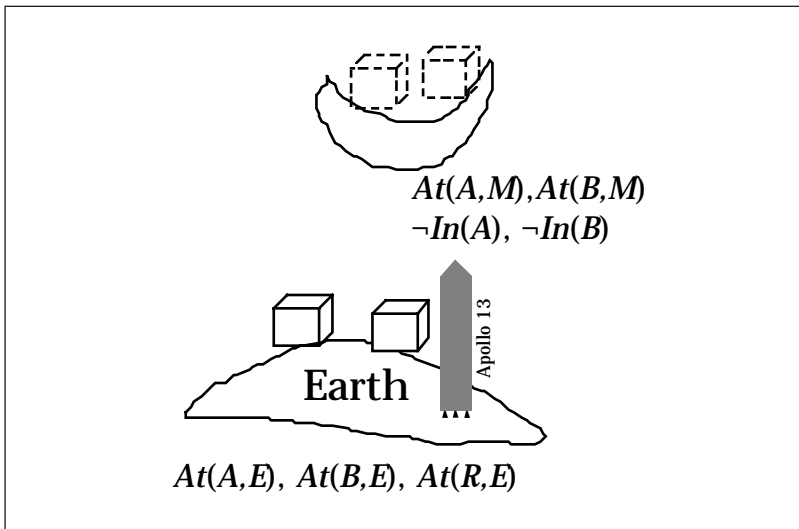


*Figure 2. Classical Planning Problem.*



*Figure 3. Rocket Domain.*

of the environment. Perhaps the simplest case of planning occurs when the environment is static (in that it changes only in response to the agent's actions) and observable, and the agent's actions have deterministic effects on the state of the environment. Plan synthesis under these conditions has come to be known as the *classical planning problem.*

The classical planning problem is thus specified (figure 2) by describing the initial state of the world, the desired goal state, and a set of deterministic actions. The objective is to find a sequence of these actions, which, when executed from the initial state, lead the agent to the goal state.

Despite its apparent simplicity and limited scope, the classical planning problem is still important in understanding the structure of intelligent agency. Work on classical planning has historically also helped our understanding of planning under nonclassical assumptions. The problem itself is computationally hard—P-space hard or worse (Erol, Nau, and Subrahmanian 1995)—and a significant amount of research has gone into efficient search-based formulations.

## Modeling Actions and States

We now look at the way the classical planning problem is modeled. Let us use a simple example scenario—that of transporting two packets from the earth to the moon, using a single rocket. Figure 3 illustrates this problem.

States of the world are conventionally modeled in terms of a set of binary state variables (also referred to as conditions). The initial state is assumed to be specified completely; so, *negated conditions* (that is, state-variables with false values) need not be shown. Goals involve achieving the specified (true-false) values for certain state variables.

Actions are modeled as state-transformation functions, with preconditions and effects. A widely used action syntax is Pednault's (1988) action description language (ADL), where preconditions and effects are first-order quantified formulas (with no disjunction in the effects formula because the actions are deterministic).

We have three actions in our rocket domain (figure 4): (1) *Load,* which causes a package to be in the rocket; (2) *Unload,* which gets it out; and (3) *Fly,* which takes the rocket and its contents to the moon. Notice the quantified and negated effects in the case of *Fly.* Its second effect says that every box that is in the rocket—before the *Fly* action is executed—will be at the moon after the action. It might be worth noting that the implication in the second effect of *Fly* is not a strict logical implication but, rather, a shorthand notation for saying that when $In(x)$ holds for any
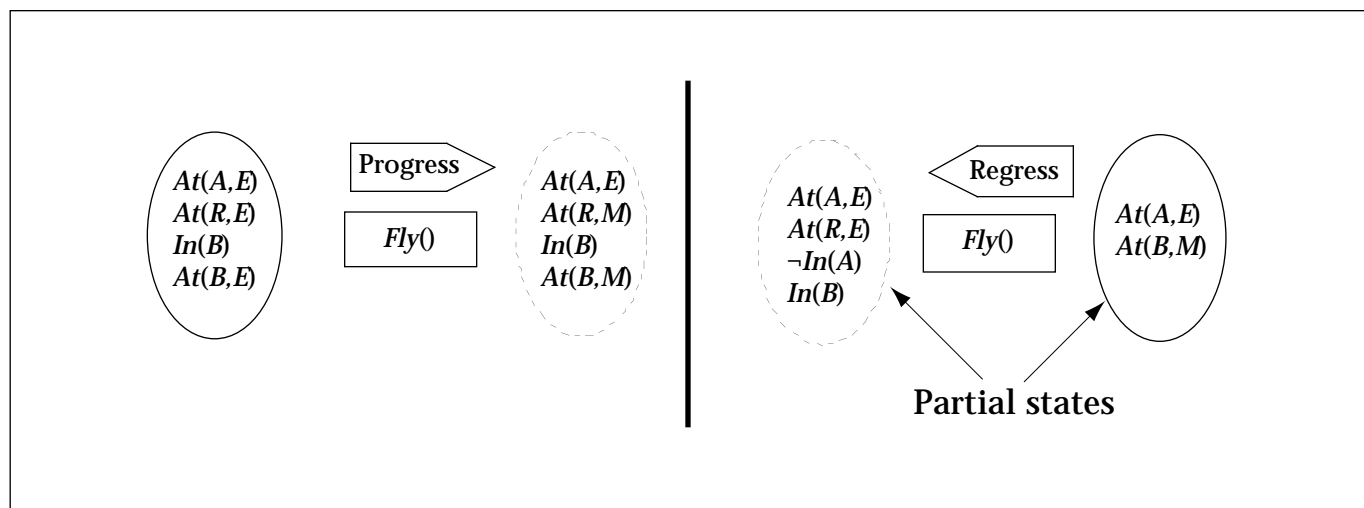
*Figure 4. Progression and Regression of World States through Actions.*

*x* in the state in which *Fly* is executed, *At(x,M)* and ¬*At(x,E)* will be true in the state resulting after the execution.

## Action Application

As mentioned earlier, actions are seen as state-transformation functions. In particular, an action can be executed in any state where its preconditions hold; on execution, the state is modified such that state variables named in the effects have the specified values, and the remaining variables retain their values. The state after the execution is undefined if the preconditions of the action do not hold in the current state. The left-hand side of figure 4 shows the result of executing the *Fly*() action in the initial state of the rocket problem. This process is also sometimes referred to as *progressing* a state through an action.

It is also useful to define the notion of regressing a state through an action. *Regressing* a state *s* through an action *a* gives the weakest conditions that must hold before *a* was executed, such that all the conditions in *s* hold after the execution. A condition *c* regresses over an action *a* to *c* if *a* has no effect corresponding to *c*, regresses to true if *a* has an effect *c*, and regresses to *d* if *a* has a conditional effect *d* => *c*. It regresses to false if *a* has an effect ¬*c* (implying that there is no state of the world where *a* can be executed to give rise to *c*). Regression of a state over an action involves regressing the individual conditions over the action and adding the preconditions of the action to the combined result. The right-hand side of figure 4 illustrates the process of regressing the final (goal) state of the rocket problem through the action *Fly*().

## Verifying a Solution

Having described how actions transform states, I can now provide a straightforward way of checking if a given action sequence is a solution to the planning problem under consideration. We start by simulating the application of the first action of the action sequence in the initial state of the problem. If the action applies successfully, the second action is applied in the resulting state, and so on. Finally, we check to see if the state resulting from the application of the last action of the action sequence is a goal state (that is, whether the state variables named in the goal specification of the problem occur in the state with the specified values). An action sequence fails to be a solution if either some action in the sequence cannot be executed in the state immediately preceding it or if the final state is not a goal state. An alternate way of verifying if the action sequence is a solution is to start by regressing the goal state over the last action of the sequence, regressing the resulting state over the second to the last action, and so on. The action sequence is a solution if all the conditions in the state resulting after regression over the first action of the sequence are present in the initial state, and none of the intermediate states are inconsistent (have the condition false in them).

## Chronology of Classical Planning Approaches

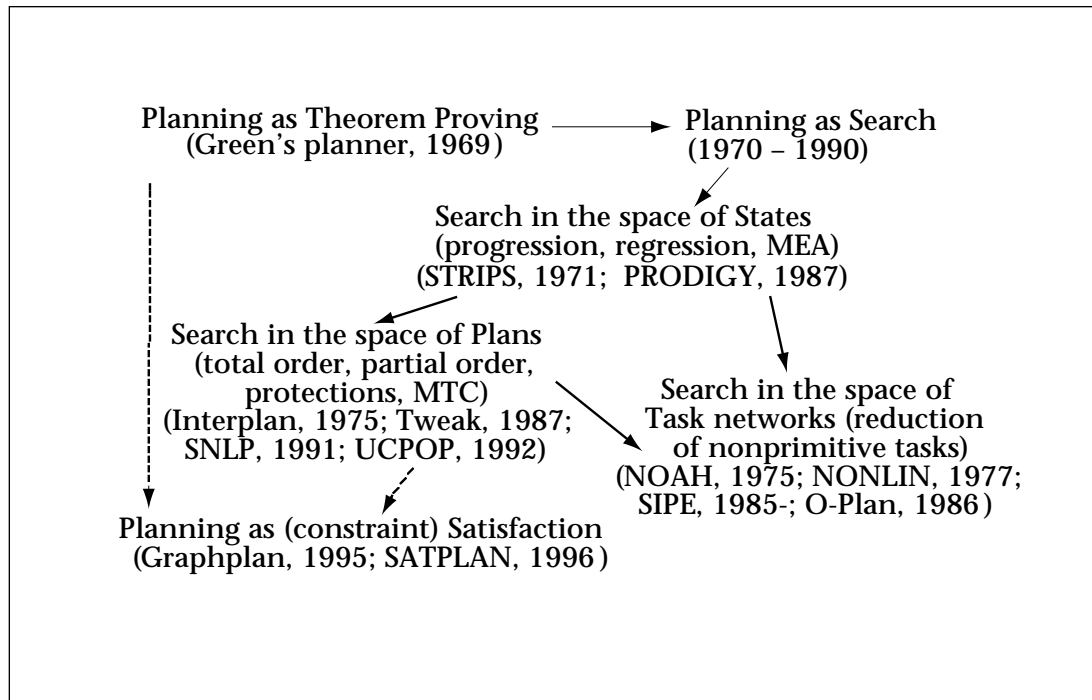Plan generation under classical assumptions has received widespread attention, and a

Planning as Theorem Proving
(Green's planner, 1969) ⟶ Planning as Search
(1970 – 1990)

Search in the space of States
(progression, regression, MEA)
(STRIPS, 1971; PRODIGY, 1987)

Search in the space of Plans
(total order, partial order,
protections, MTC)
(Interplan, 1975; Tweak, 1987;
SNLP, 1991; UCPOP, 1992)

Search in the space of
Task networks (reduction
of nonprimitive tasks)
(NOAH, 1975; NONLIN, 1977;
SIPE, 1985-; O-Plan, 1986)

Planning as (constraint) Satisfaction
(Graphplan, 1995; SATPLAN, 1996)

*Figure 5. A Chronology of Ideas in Classical Planning.*

large variety of planning algorithms have been developed (figure 5). Initial approaches to the planning problem have attempted to cast planning as a theorem-proving activity (Green 1969). The inefficiency of first-order theorem provers in existence at that time, coupled with the difficulty of handling the "frame problem"[1] in first-order logic, has led to search-based approaches in which the STRIPS assumption—namely, the assumption that any condition not mentioned in the effects list of an action remains unchanged after the action—is hard wired. Perhaps the first of these search-based planners was STRIPS (Fikes and Nilsson 1971), which searched in the space of world states using means-ends analysis (see Making State-Space Refinements Goal Directed). Searching in the space of states was found to be inflexible in some cases, and a new breed of approaches formulated planning as a search in the space of partially constructed plans (Penberthy and Weld 1992; McAllester and Rosenblitt 1991; Chapman 1987; Tate 1975). A closely related formulation called *hierarchical planning* (Wilkins 1984; Tate 1977; Sacerdoti 1972) allowed a partial plan to contain abstract actions that can incrementally be reduced to concrete actions. More recently, encouraged by the availability of high-performance constraint-satisfaction algorithms, formulations of planning as a constraint-satisfaction problem have become popular (Joslin and Pollack 1996; Kautz and Selman 1996; Blum and Furst 1995).

One of my aims in this article is to put all these approaches in a logically coherent framework so that we can see the essential connections among them. I use the refinement planning framework to effect such a unification.

## Refinement Planning: Overview

Because a solution for a planning problem is ultimately a sequence of actions, plan synthesis in a general sense involves sorting our way through the set of all action sequences until we end up with a sequence that is a solution. Thus, we have the essential idea behind refinement planning, that is, the process of starting with the set of all action sequences and gradually narrowing it down to reach the set of all solutions. The sets of action sequences are represented and manipulated in terms of partial plans that can be seen as a collection of constraints. The action sequences denoted by a partial plan, that is, those that are consistent with its constraints, are called its *candidates.* For technical reasons that become clear later, we find it convenient to think in terms of sets of partial plans (instead of single partial plans). A set of par-
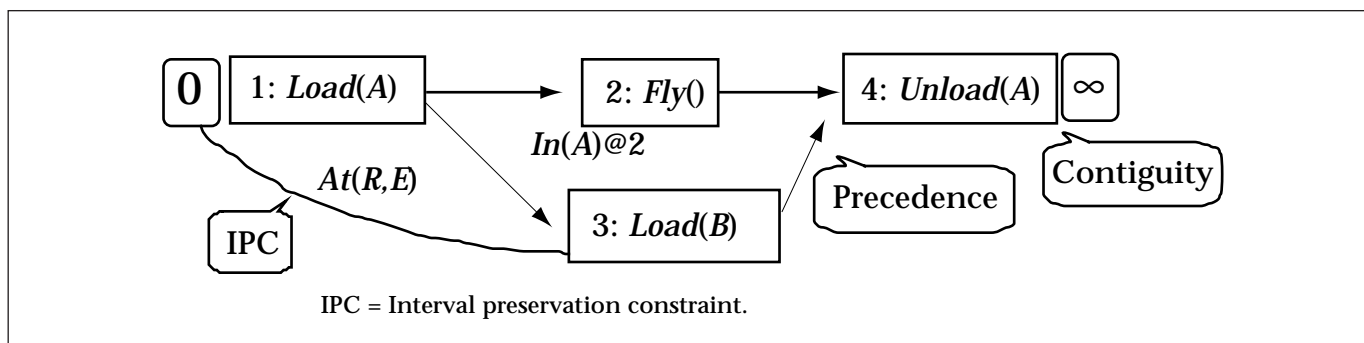
*Figure 6. An Example (Partial) Plan in the Rocket Domain.*

tial plans is called a *plan set,* and its constituent partial plans are referred to as the *components.* The *candidate set* of a plan set is defined as the union of the candidate sets of its components.

Plan synthesis in refinement planning involves a "split and prune" search (Pearl 1980). The pruning is carried out by applying refinement operations to a given plan set. The aim is to incrementally get rid of nonsolutions from the candidate set. The splitting part involves pushing the component partial plans of a plan set into different branches of the search tree. The motivation is to reduce the cost of applying refinements and termination check to the plan set and to support more focused pruning. Termination test involves checking if the set of minimal-length candidates of the current plan set contains a solution.

To make these ideas precise, we now look at the syntax and semantics of partial plans and refinement operations.

## Partial Plan Representation: Syntax

A *partial plan* can be seen as any set of constraints that together delineate which action sequences belong to the plan's candidate set and which do not. One representation[2] that is sufficient for our purpose models partial plans as a set of steps, ordering constraints between the steps, and auxiliary constraints.[3]

Each plan step is identified with a unique step number and corresponds to an action (allowing two different steps to correspond to the same action, thus facilitating plans containing more than one instance of a given action). There can be two types of ordering constraint between a pair of steps: (1) precedence and (2) contiguity. A *precedence constraint* requires one step to precede the second step (without precluding other steps from coming between the two), and a *conti-*

*guity constraint* requires that the two steps come immediately next to each other.

Auxiliary constraints involve statements about the truth of certain conditions over certain time intervals. We are interested in two types of auxiliary constraint: (1) *interval-preservation constraints* (IPCs), which require nonviolation of a condition over an interval (no action having an effect ¬$p$ is allowed in an interval where the condition $p$ is to be preserved), and (2) *point-truth constraints,* which require the truth of a condition at a particular time point.

A *linearization* of a partial plan is a permutation of its steps that is consistent with all its ordering constraints (in other words, a topological sort). A *safe linearization* is a linearization of the plan that is also consistent with the auxiliary constraints.

Figure 6 illustrates a plan from our rocket domain: Step 0 corresponds to the beginning of the plan. Step ∞ corresponds to the end of the plan. By convention, the effects of step 0 correspond to the conditions that hold in the initial state of the plan, and the preconditions of step ∞ correspond to the conditions that must hold in the goal state. There are four steps other than 0 and ∞. Steps 1, 2, 3, and 4 correspond respectively to the actions *Load*($A$), *Fly*(), *Load*($B$), and *Unload*($A$). The steps 0 and 1 are contiguous, as are the steps 4 and ∞ (illustrated in the figure by putting them next to each other); step 2 precedes 4; and the condition *At*($R$,$E$) must be preserved between 0 and 3 (illustrated in figure 6 by a labeled arc between the steps). Finally, the condition *In*($A$) must hold in the state preceding the execution of step 2. The sequences 0-1-2-3-4-∞ and 0-1-3-2-4-∞ are linearizations of this plan. Of these, the second one is a safe linearization, but the first one is not (because step 2 will violate the interval-preservation constraint on *At*($R$,$E$) between 0 and 3).

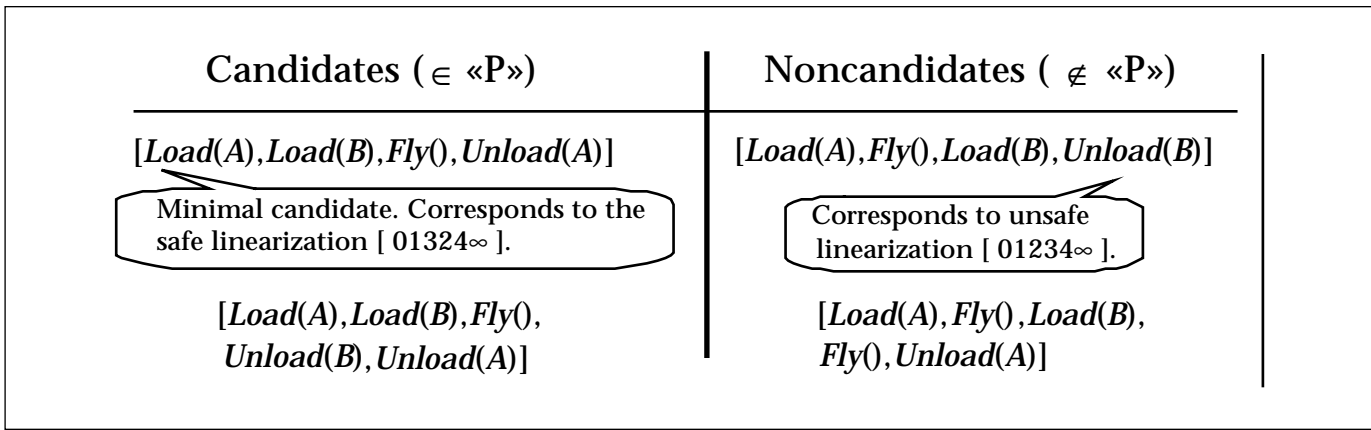| Candidates ($\in$ «P») | Noncandidates ($\notin$ «P») |
|---|---|
| [*Load*(*A*), *Load*(*B*), *Fly*(), *Unload*(*A*)] | [*Load*(*A*), *Fly*(), *Load*(*B*), *Unload*(*B*)] |
| Minimal candidate. Corresponds to the safe linearization [ 01324∞ ]. | Corresponds to unsafe linearization [ 01234∞ ]. |
| [*Load*(*A*), *Load*(*B*), *Fly*(), *Unload*(*B*), *Unload*(*A*)] | [*Load*(*A*), *Fly*(), *Load*(*B*), *Fly*(), *Unload*(*A*)] |

*Figure 7. Candidate Set of a Plan.*

## Partial Plan Representation: Semantics

The semantics of the partial plans are given in terms of candidate sets. A candidate can be seen as a model of the partial plan. An action sequence belongs to the candidate set of a partial plan if it contains the actions corresponding to all the steps of the partial plan in an order consistent with the ordering constraints on the plan, and it also satisfies all auxiliary constraints. For the plan shown in figure 6, the action sequences shown on the left in figure 7 are candidates, but those on the right are noncandidates.

Notice that the candidates might contain more actions than are present in the partial plan; thus, a plan's candidate set can potentially be infinite. We define the notion of minimal candidates to let us restrict our attention to a finite subset of the possibly infinite candidate set. Specifically, *minimal candidates* are candidates that only contain the actions listed in the partial plan (thus, their length is equal to the number of steps in the plan other than 0 and ∞). The top candidate on the left of figure 9 is a minimal candidate, but the bottom one is not. There is a one-to-one correspondence between the minimal candidates and the safe linearizations of a plan. For example, the minimal candidate on the top left of figure 7 corresponds to the safe linearization 0-1-3-2-4-∞ (as can be verified by translating the step names in the latter to corresponding actions).

The sequences on the right of figure 7 are noncandidates because both of them fail to satisfy the auxiliary constraints. Specifically, the first one corresponds to the unsafe linearization 1-2-3-4-∞. The second noncandidate starts with the minimal candidate [*Load*(*A*), *Load*(*B*), *Fly*(), *Unload*(*A*)] and adds another instance of the action *Fly*() that does not respect the IPC on *At*(*R*,*E*).

## Connecting Syntax and Semantics of Partial Plans

Figure 8 summarizes the connection between the syntax and the semantics of a partial plan. Each partial plan has, at most, an exponential number of linearizations, some of which are safe with respect to the auxiliary constraints. Each safe linearization corresponds to a minimal candidate of the plan. Thus, there are, at most, an exponential number of minimal candidates. A potentially infinite number of additional candidates can be derived from each minimal candidate by padding it with new actions without violating auxiliary constraints. Minimal candidates can thus be seen as the generators of the candidate set of the plan. The one-to-one correspondence between safe linearizations and minimal candidates implies that a plan with no safe linearizations will have an empty candidate set.

**Minimal candidates and solution extraction:** Minimal candidates have another important role from the point of view of refinement planning. We see in the following discussion that some refinement strategies add new step constraints to a partial plan. Thus, they simultaneously shrink the candidate set of the plan and increase the length of its minimal candidates. This property provides an incremental way of exploring the (potentially infinite) candidate set of a partial plan for solutions: Examine the minimal candidates of the plan after each refinement to see if any of them correspond to solutions. Checking if a minimal candidate is a solution can be done in linear time by simulating the execution of the minimal candidate and
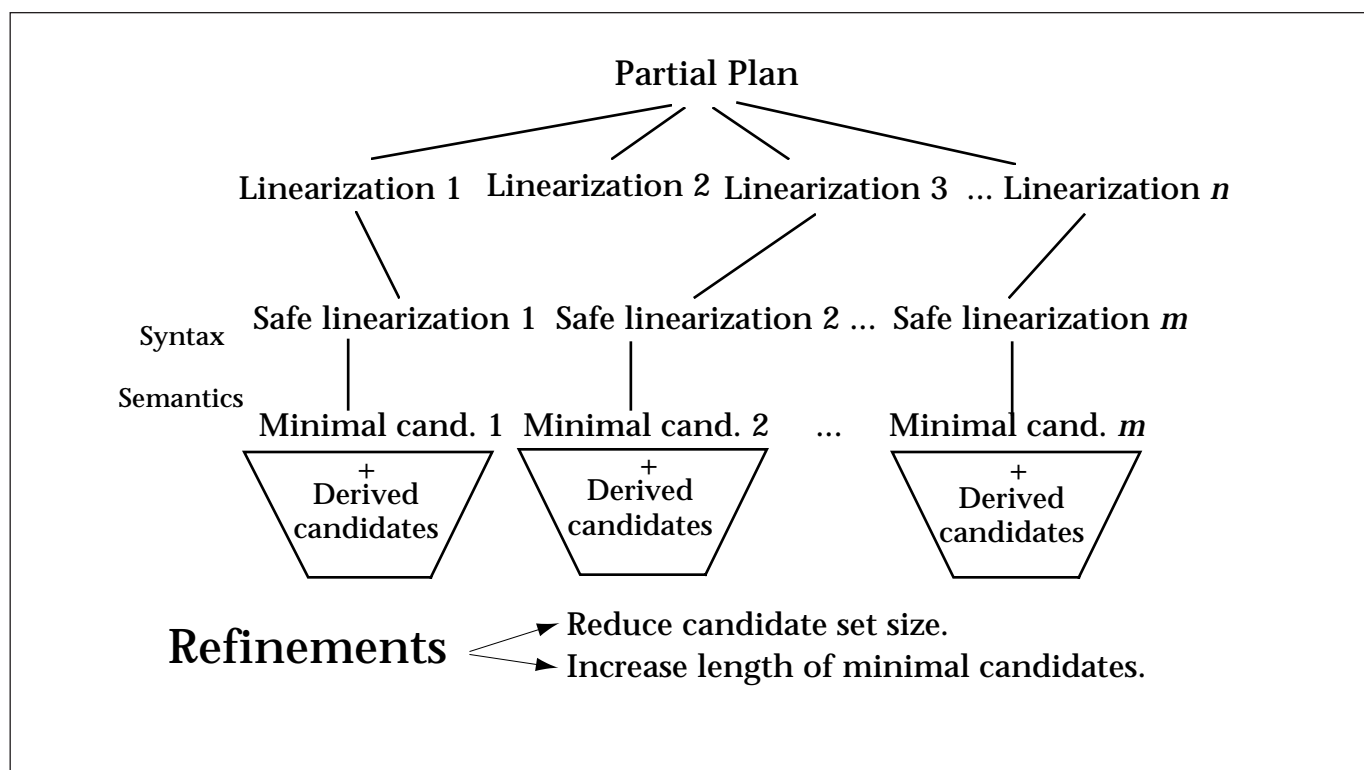
*Figure 8. Relating the Syntax and the Semantics of a Partial Plan.*

checking to see if the final state corresponds to a goal state (see Verifying a Solution).

## Refinement Strategies

Refinements are best seen as canned procedures that compute the consequences of the metatheory of planning, and the domain theory (in the form of actions), in the specific context of the current partial plan. A refinement strategy $R$ maps a plan set $P$ to another plan set $P'$ such that the candidate set of $P'$ is a subset of the candidate set of $P$. $R$ is said to be complete if $P'$ contains all the solutions of $P$. It is said to be *progressive* if the candidate set of $P'$ is a strict subset of the candidate set of $P$. It is said to be strongly progressive if the length of the minimal candidates increases after the refinement. It is said to be *systematic if* no action sequence falls in the candidate set of more than one component of $P'$. We can also define the *progress factor* of a refinement strategy as the ratio between the size of the candidate set of the refined plan set and the size of the original plan set.

Completeness ensures that we don't lose solutions with the application of refinements. Progressiveness ensures that refinement has pruning power. Systematicity ensures that we never consider the same candidate more than once if we explore the components of the

plan set separately (see Introducing Splitting into Refinement Planning).

Let us illustrate these notions with an example. Figure 9 shows an example refinement for our rocket problem. It takes the null plan set, corresponding to all action sequences, and maps it to a plan set containing three components. In this case, the refinement is complete because no solution to the rocket problem can start with any other action for the given initial state; progressive because it eliminated action sequences not beginning with *Load*(A), *Load*(B), or *Fly*() from consideration; and systematic because no action sequence will belong to the candidate set of more than one component (the candidates of the three components will differ in the first action).

Refinement strategies are best seen as canned inference procedures that compute the consequences of the metatheory of planning, and the domain theory (in the form of actions), in the specific context of the commitments in the current partial plan. In this case, given the theory of planning, which says that solutions must have actions that are executable in the states preceding them, and the current plan set constraint that the state of the world before the first step is the initial state where the rocket and the packages are
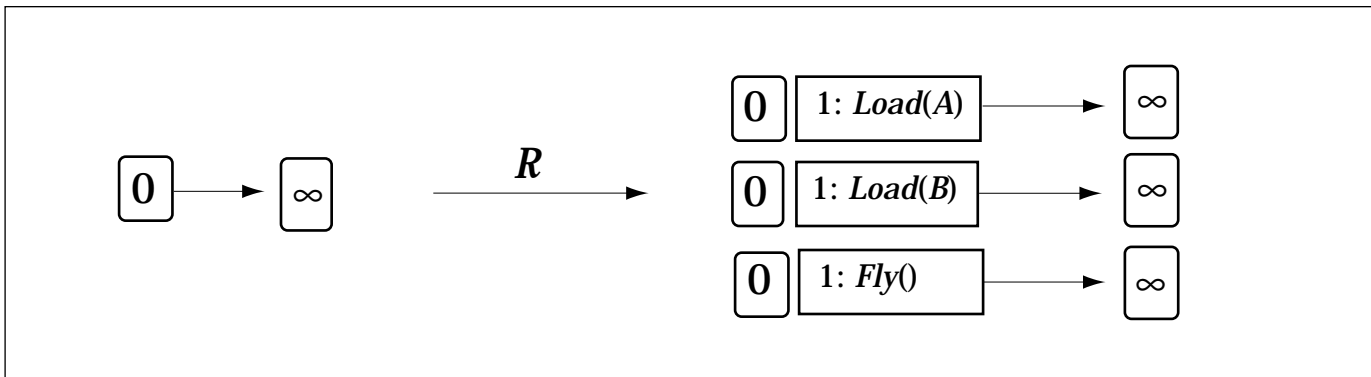
*Figure 9. An Example Refinement Strategy (Forward State Space).*

on earth, the forward state-space refinement infers that the only actions that can come as the second step in the solution are *Load*(*A*), *Load*(*B*), and *Fly*().

## Planning Using Refinement Strategies and Solution Extraction

Now, I present the general refinement planning template: It has three main steps. If the current plan set has an extractable solution—which is checked by inspecting its minimal candidates to see if any of them is a solution—we terminate. If not, we select a refinement strategy *R* and apply it to the current plan set to get a new plan set.

Refine (*P*: Plan set)

0*. If <<*P*>> is empty, Fail.

1. If a minimal candidate of *P* is a solution, return it. End.

2. Select a refinement strategy *R*.
   Apply *R* to *P* to get a new plan set *P'*.

3. Call *Refine*(*P'*).

As long as the selected refinement strategy is complete, we never lose a solution. As long as the refinements are progressive, for solvable problems, we eventually reach a plan set, one of whose minimal candidates is a solution.

The *solution-extraction process* involves checking the minimal candidates (corresponding to safe linearizations) of the plan to see if any one of them are solutions. This process can be cast as a model finding or satisfaction process (Kautz and Selman 1996). Recall that a candidate is a solution if each of the actions in the sequence has its preconditions satisfied in the state preceding the action.

As we see in Scale Up through Disjunctive Representations and Constraint-Satisfaction Techniques, some recent planners such as GRAPHPLAN (Blum and Furst 1995) and SATPLAN (Kautz and Selman 1996) can be seen as instantiations of this general refinement planning template. However, most earlier planners use a specialization of this template that I discuss next.

## Introducing Splitting into Refinement Planning

Seen as an instantiation of split and prune search, the previous algorithm does not do any splitting. It is possible to add splitting to the refinement process in a straightforward way—to separate the individual components of a plan set and handle them in different search branches. The primary motivation for this approach is to reduce the solution-extraction cost. After all, checking for solution in a single partial plan is cheaper than searching for a solution in a plan set. Another possible advantage of handling individual components of a plan set separately is that this technique, coupled with a depth-first search, might make the process of plan generation easier for humans to understand.

Notice that this search process will have backtracking even for complete refinements. Specifically, even if we know that the candidate set of a plan set *P* contains all the solutions to the problem, we do not know how they are distributed among the components of *P*. We see later (see Trade-Off in Refinement Planning) that the likelihood of the backtracking depends on the number and size of the individual components of *P*, which can be related to the nature of constraints added by the refinement strategies.

The following algorithm template introduces splitting into refinement planning:

Refine (*P*: Plan)

0*. If <<*P*>> is empty, Fail.

1. If *SOL*(*P*) returns a solution, terminate with success.
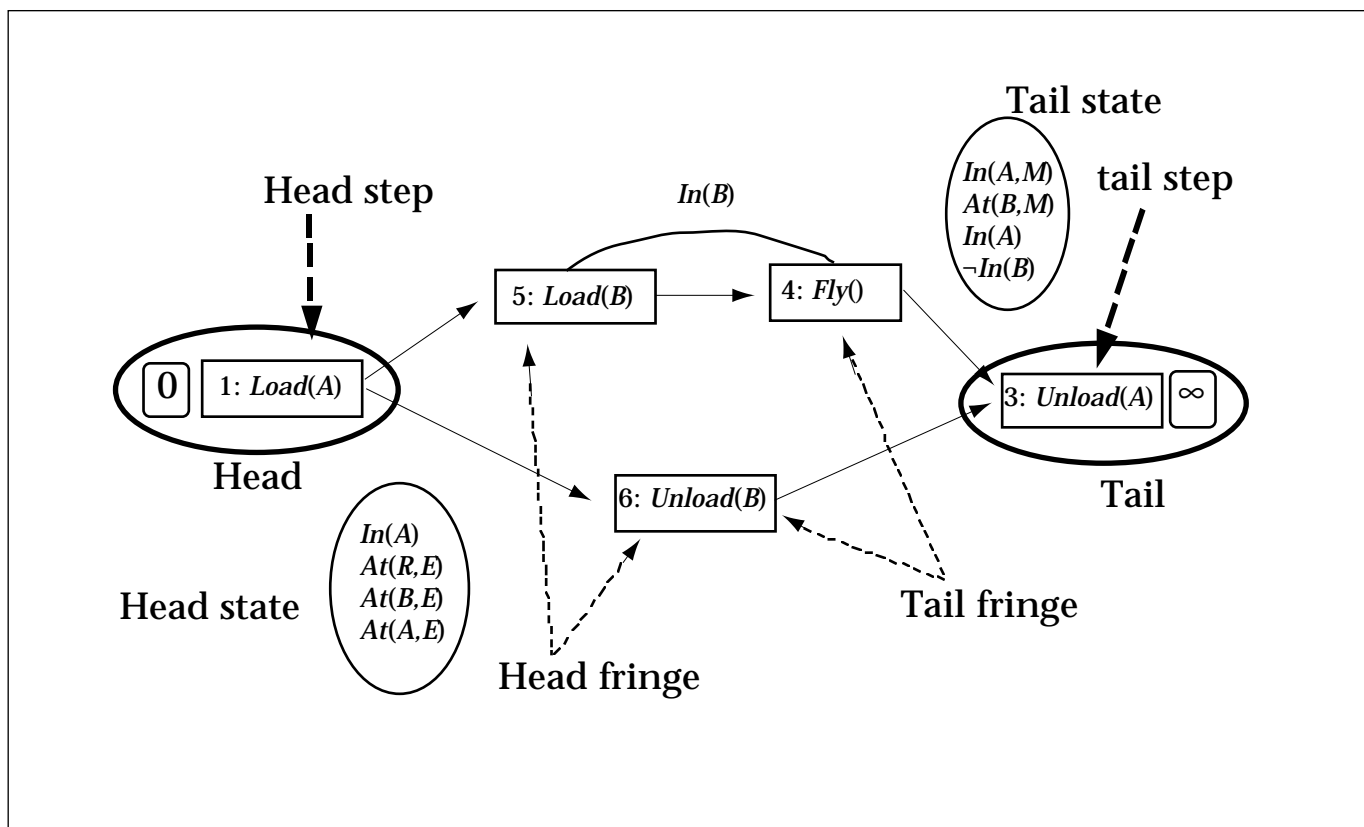
2. Select a refinement strategy *R*.

*Figure 10. Nomenclature for Partial Plans.*

Apply *R* to *P* to get a new plan set *P′*.

3. Nondeterministically select a component *P′ᵢ* of P′.

Call *Refine*(*P′ᵢ*).

It is worth noting the two new steps that made their way. First, the components of the plan set resulting after refinement are pushed into the search space and are handled separately (step 3). Thus, we confine the application of refinement strategies to single plans. Second, once we work on individual plans, we can consider solution-extraction functions that are cheaper than looking at all the minimal candidates (as we see in the next section).

This simple algorithm template forms the main idea behind all existing refinement planners. Various existing planners differ in terms of the specific refinement strategies they use in step 2. These refinement strategies fall broadly into four types: (1) state space, (2) plan space, (3) task reduction, and (4) tractability. We now look at these four refinement families in turn.

## Existing Refinement Strategies

In this section, we look at the details of the four different families of refinement strategies. Before we start, it is useful to introduce some additional terminology to describe the structure of partial plans. Figure 10 shows an example plan with its important structural aspects marked. The prefix of the plan, that is, the maximal set of steps constrained to be contiguous to step 0 (more specifically, steps $s_1, s_2, \ldots s_n$ such that $0*s_1, s_1*s_2, \ldots s_{n-1}*s_n$) is called the *head* of the plan. The last step of the head is called the *head step.* Because we know how to compute the state of the world that results when an action is executed in a given state, we can easily compute the state of the world after all the steps in the head are executed. We call this state the *head state.* Similarly, the suffix of the plan is called the *tail* of the plan, and the first step of the suffix is called the *tail step.* The set of conditions obtained by successively regressing the goal state over the actions of the tail is called the *tail state.* The tail state provides the weakest conditions under which the actions in the tail of the plan can be executed to result in a

*Compared to the forward state-space refinement, the backward state-space refinement generates plan sets with a fewer number of components because it concentrates only on those actions that are relevant to current goals.*

*This focus on relevant actions, in turn, leads to a lower branching factor for planners that consider the plan set components in different search branches.*

goal state. The steps in the plan that neither belong to its head nor belong to its tail are called the *middle steps.* Among the middle steps, the set of steps that can come immediately next to the head step in some linearization of the plan is called its *head fringe.* Similarly, the *tail fringe* consists of the set of middle steps that can come immediately next to the tail step in some linearization. With this terminology, we are ready to describe individual refinement strategies.

### Forward State-Space Refinement

Forward state-space refinement involves growing the prefix of a partial plan by introducing actions in the head fringe or the action library into the plan head. Actions are introduced only if their preconditions hold in the current head state:

Refine-forward-state-space (*P*)

1. Operator selection: Nondeterministically select a step *t* either from the operator library or from the head fringe such that the preconditions of *t* are applicable in the head state.
2. Operator application: Add a contiguity constraint between the current head step and the new step *t,* which makes *t* the new head step and updates the head state.

Figure 11 shows an example of this refinement. On the top is a partial plan whose head contains the single step 0, and the head state is the same as the initial state. The head fringe contains the single action *Unload*(*A*), which is not applicable in the head state. The action library contains three actions that are applicable in the head state. Accordingly, forward state-space refinement produces a plan set with three components.

Forward state-space refinement is progressive because it eliminates all action sequences with nonexecutable prefixes. It is also complete because any solution must have an executable prefix. It is systematic because each of its components differs in the sequence of steps in the plan head, and thus, their candidates will have different prefixes. Finally, if we are using only forward state-space refinements, we can simplify the solution-extraction function considerably—we can terminate as soon as the head state of a plan set component contains its tail state. At that point, the only minimal candidate of the plan corresponds to a solution.

The version of the refinement we considered previously extends the head by one action at a time, possibly leading to larger-than-required number of components in the resulting plan set. Consider the actions *Load*(*A*) and *Load*(*B*), each of which is applicable in the initial state, and both of which can be done simultaneously because they do not interact in any way. From the point of view of search-space size, it would be cheaper in such cases to add both actions to the plan prefix, thereby reducing the number of components in the resulting plan set. Simultaneous addition of multiple actions to the head can be accommodated by generalizing the contiguity constraints so that they apply to sets of steps. In particular, we could combine the top two components of the plan set in figure 11 into a single plan component, with both *Load*(*A*) and *Load*(*B*) made contiguous to step 0. More broadly, the generalized forward state-space refinement strategy should consider maximal sets of noninteracting actions that are all applicable in the current initial state together (Drummond 1989). Here, we consider two actions to interact if the preconditions of one are deleted by the effects of the other.

## Making State-Space Refinements Goal Directed

Forward state-space refinement as stated considers all actions executable in the state after the prefix. In real domains, there might be a large number of applicable actions, very few of which are relevant to the top-level goals of the problem. We can make the state-space refinements goal directed in one of two ways: (1) using means-ends analysis to focus forward state-space refinement on only those actions that are likely to be relevant to the top-level goals or (2) using backward state-space refinement that operates by growing the tail of the partial plan. I elaborate on these ideas in the next two subsections.

**Means-Ends Analysis**    First, we can force forward state-space refinement to consider only those actions that are going to be relevant to the top-level goals. The relevant actions can be recognized by examining a subgoaling tree of the problem, as shown in figure 12. Here the top-level goal can potentially be  achieved by the effects of the actions *Unload*(*A*) and *Fly*(). The preconditions of these actions are, in turn, achieved by the action *Load*(*A*). Because *Load*(*A*) is both relevant (indirectly) to the top goals and is applicable in the head state, the forward state-space refinement can consider this action. In contrast, an action such as *Load*(*C*) will never be considered despite its applicability because it does not directly or indirectly support any top-level goals.

This way of identifying relevant actions is known as *means-ends analysis* and was used by one of the first planners, STRIPS (Fikes and Nilsson 1971). One issue in using means-ends analysis is whether the relevant actions are identified afresh at each refinement cycle or whether the refinement and means-ends analysis are interleaved. STRIPS interleaved the computation of relevant actions with forward state-space refinement, suspending the means-ends analysis as soon as an applicable action has been identified. The action is then made contiguous to the current head step, thus changing the head state. The means-ends analysis is resumed with respect to the new state. In the context of the example shown in figure 12, having decided that the *Fly*() and *Unload*(*A*) actions are relevant for the top-level goals, STRIPS introduces the *Fly*() action into the plan head before continuing to consider actions such as *Load*(*A*) that are recursively relevant. Such interleaving can sometimes lead to premature operator application, which would have to be backtracked over. In fact, many of the famous incomplete-
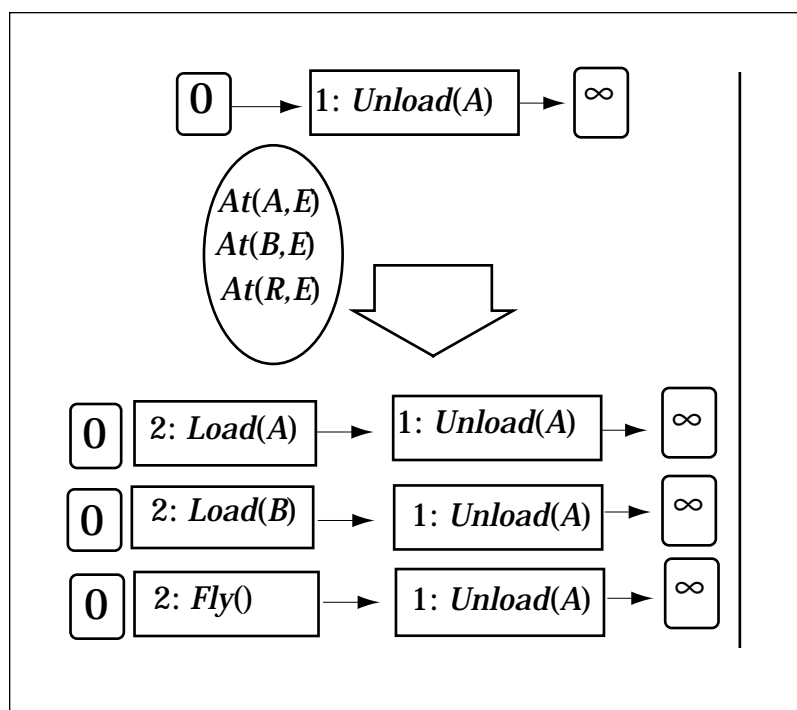
ness results related to STRIPS (Nilsson 1980) can be traced to this particular interleaving. More recently, McDermott (1996) showed that the efficiency of means-ends–analysis planning can be improved considerably by (1) deferring operator application until all relevant actions are computed and (2) repeating the computa-
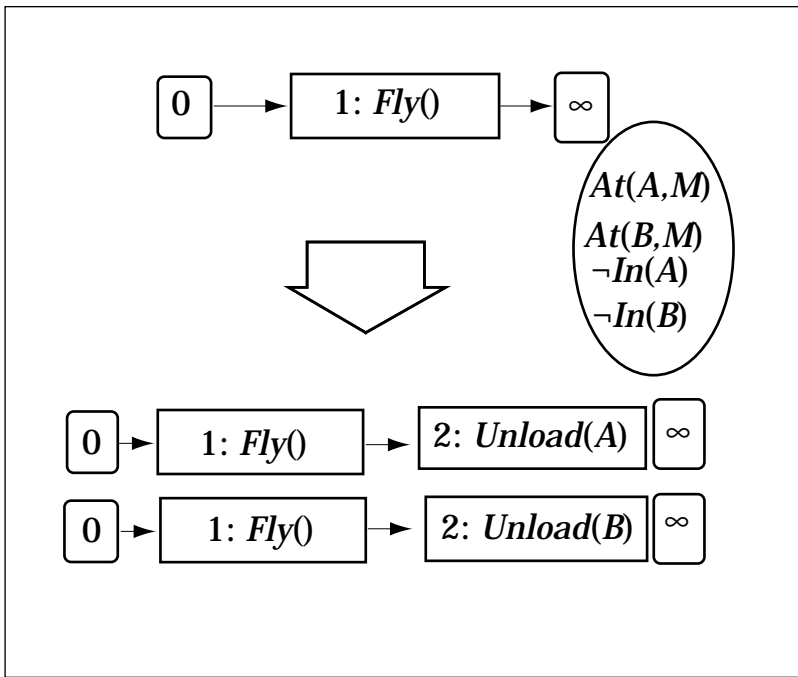


*Figure 11. Example of Forward State-Space Refinement.*
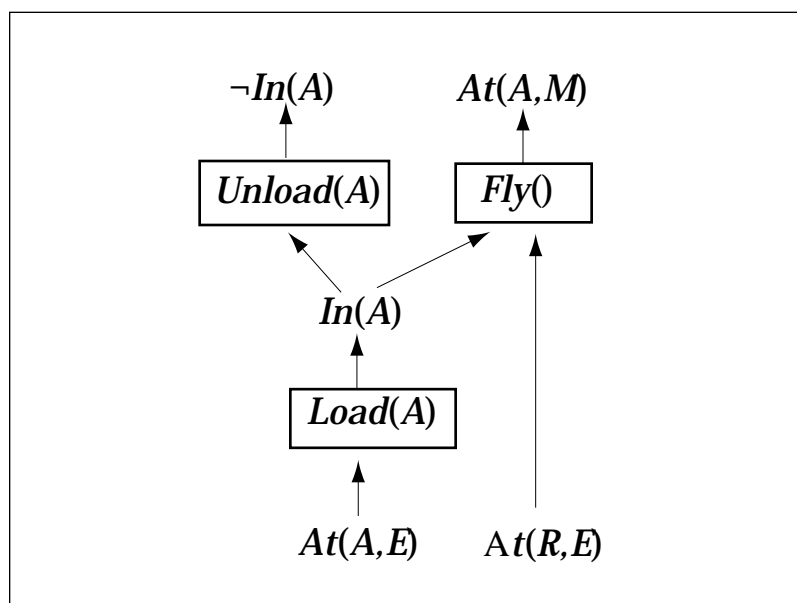


*Figure 12. Using Means-Ends Analysis to Focus Forward State-Space Refinement.*

*Figure 13. Backward State-Space Refinement.*

tion of relevant actions afresh after each refinement.

**Backward State-Space Refinement**
The second way of making state-space refinements goal directed is to consider growing the tail of the partial plan by applying actions in the backward direction to the tail state. All actions in the tail fringe or actions from the plan library are considered for application. An action is applicable to the tail state if it does not delete any conditions in the tail state (if it does, the regressed state will be inconsistent) and adds at least one condition in the tail state:

Refine-backward-state-space (*P*)

1. Operator selection: Nondeterministically select a step *t* either from the operator library or from the tail fringe such that at least one of its effects is relevant to the tail state, and none of its effects negates any conditions in the tail state.

2. Operator application: Add a contiguity constraint between the new step *t* and the current tail step, which makes *t* the new tail step and updates the tail state.

Figure 13 shows an example of backward state-space refinement. Here, the tail contains only the last step of the plan, and the tail state is the same as the goal state (shown in an oval on the right). Two library actions, *Unload*(*A*) and *Unload*(*B*), are useful in that

they can give some of the conditions of the tail state without violating any others. The *Fly*() action in the tail fringe is not applicable because it can violate the ¬*In*(*x*) condition in the tail state:

Compared to the forward state-space refinement, the backward state-space refinement generates plan sets with a fewer number of components because it concentrates only on those actions that are relevant to current goals. This focus on relevant actions, in turn, leads to a lower branching factor for planners that consider the plan set components in different search branches. However, because the initial state of a planning problem is completely specified, and the goal state is only partially specified, the head state computed by the forward state-space refinement is a complete state, but the tail state computed by the backward state-space refinement is only a partial state description. Bacchus and Kabanza (1995) argue that effective search-control strategies might require the ability to evaluate the truth of complex formulas about the state of the plan and view this approach as an advantage in favor of forward state-space refinements because truth evaluation can be done in terms of model checking rather than theorem proving.

## Position, Relevance, and Plan-Space Refinements

The state-space refinements have to decide both the position and the relevance of a new action to the overall goals. Often times, we might know that a particular action is relevant but not know its exact position in the eventual solution. For example, we know that a fly action is likely to be present in the solution for the rocket problem but do not know exactly where in the plan it will occur. Plan-space refinement is motivated by the idea that in such cases, it helps to introduce an action into the plan without constraining its absolute position.[4] Of course, the disadvantage of not fixing the position is that we will not have state information, which makes it harder to predict the states of the world during the execution based on the current partial plan.

The difference between state-space and plan-space refinements has traditionally been understood in terms of least commitment, which, in turn, is related to candidate-set size. Plans with precedence relations have larger candidate sets than those with contiguity constraints. For example, it is easy to see that although all three plans shown in figure 14 contain the single *Fly* action, the solution sequence
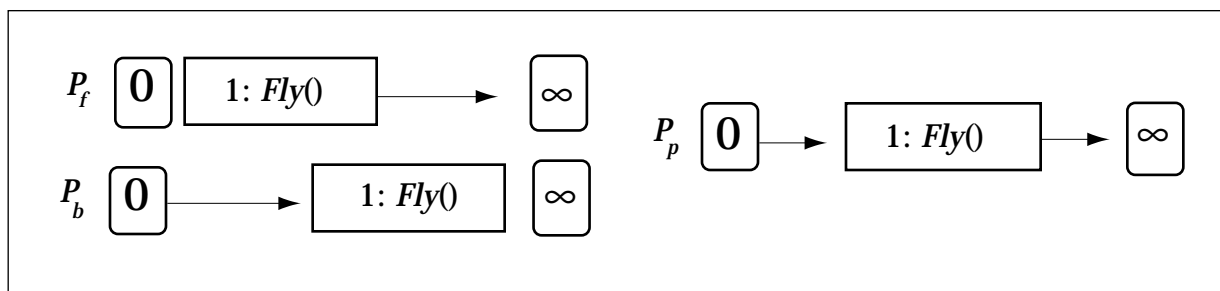
*Figure 14. State-Space Refinements Attempt to Guess
Both the Position and the Relevance of an Action to a Given Planning Problem.*
Plan-space refinements can consider relevance without committing to position.

[*Load*(*A*), *Load*(*B*), *Fly*, *Unload*(*A*), *Unload*(*B*)]

belongs only to the candidate set of the plan with precedence constraints.

Because each search branch corresponds to a component of the plan set produced by the refinement, planners using state-space refinements are thus more likely to backtrack from a search branch. (It is, of, course worth noting that the backtracking itself is an artifact of splitting plan-set components into the search space. Because all refinements are complete, backtracking would never be required had we worked with plan sets without splitting.)

This brings us to the specifics of plan-space refinement. As summarized here, the plan-space refinement starts by picking any precondition of any step in the plan and introducing constraints to ensure that the precondition is provided by some step (establishment) and is preserved by the intervening steps (declobbering):

Refine-Plan-Space(*P*)

1. Goal selection: Select a precondition <*C,s*> of *P*.

2. Goal establishment: Nondeterministically select a new or existing step *t*.

   Establishment: Force *t* to come before *s* and give *C*.

   Arbitration: Force every step between *t* and *s* to preserve *C*.

3. Bookkeeping: (*optional*)

   Add IPC <*t,C,s*> to protect the establishment.

   Add IPC <*t,¬C,s*> to protect the contributor.

An optional bookkeeping step (also called the *protection step*) imposes interval-preservation constraints to ensure that the established precondition is protected during future refinements. Plan-space refinement can have several instantiations depending on whether bookkeeping strategies are used and how the preconditions are selected for establishment in step 1.

Figure 15 shows an example of plan-space refinement. In this example, we pick the precondition *At*(*A,M*) of the last step. We add the new step *Fly*() to support this condition. To force *Fly* to give *At*(*A,M*), we add the condition *In*(*A*) as a precondition to it. This latter condition is called a *causation precondition*. At this point, we need to make sure that any step possibly intervening between the step 2: *Fly* and the step ∞ preserves *At*(*A,M*). In this example, only *Unload*(*A*) intervenes, and it does preserve the condition; so, we are done. The optional bookkeeping step involves adding interval preservation constraints to preserve this establishment during subsequent refinement operations (when new actions might come between *Fly* and the last step). In the example, the bookkeeping step involves adding either one or both of the interval-preservation constraints <2, *At*(*A,M*), ∞> and <2, ¬*At*(*A,M*), ∞>. Informally, the first one ensures that no action deleting *At*(*A,M*) will be allowed between 2 and ∞. The second one says that no action adding *At*(*A,M*) will be allowed between 2 and ∞. If we add both these constraints, we can show that the refinement is systematic (McAllester and Rosenblitt 1991). As an aside, the fact that we are able to ensure systematicity of the refinement without fixing the positions of any of the steps involved is technically quite interesting.

Figure 16 is another example where we need to do both establishment and declobbering to support a precondition. Specifically, we consider the precondition *At*(*A,E*) of step ∞ and establish it using the effects of the existing step 0. No causation preconditions are required because 0 gives *At*(*A,E*) directly. However, the step 1: *Fly*() can delete the condition *At*(*A,E*), and it is coming in between 0
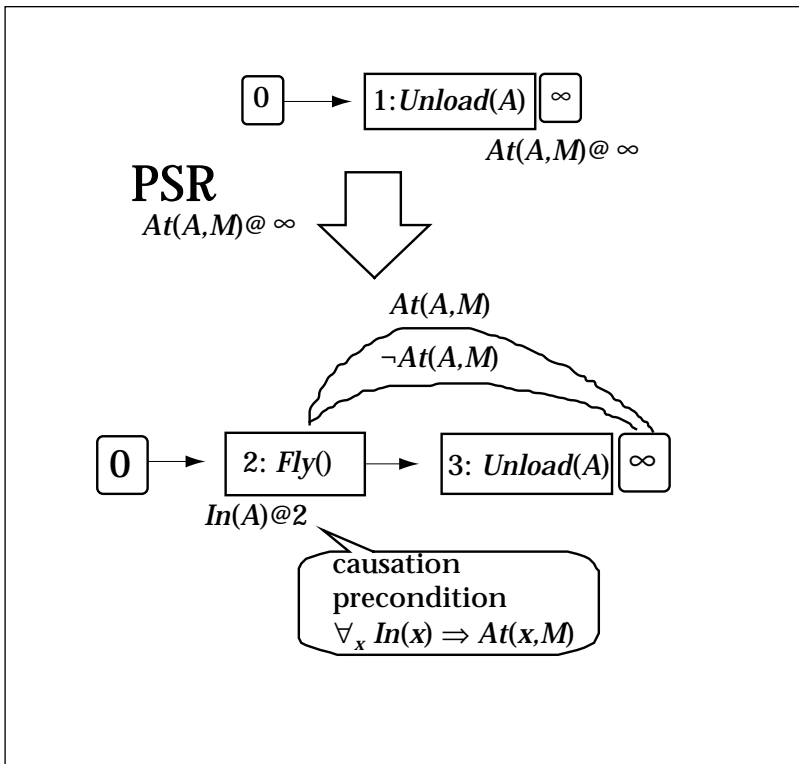
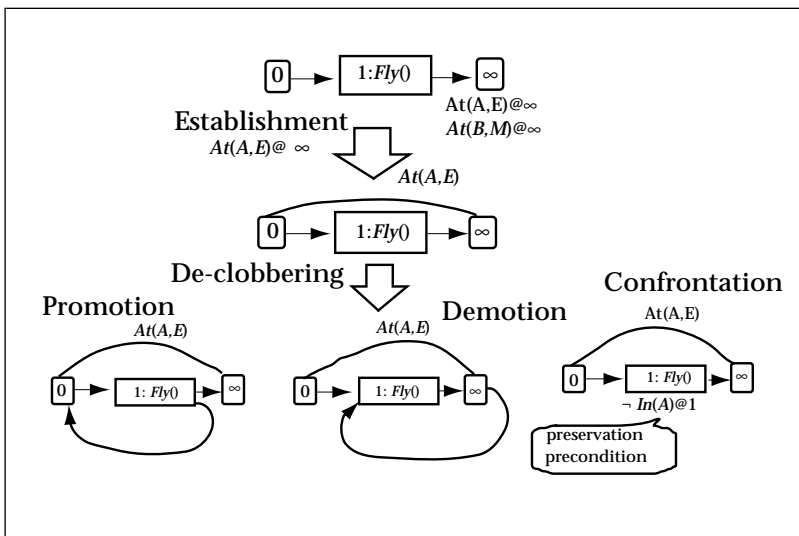*Figure 15. Example of Plan-Space Refinement.*



*Figure 16. Plan-Space Refinement Example Showing Both Establishment and Declobbering Steps.*

and $\infty$. To shore up the establishment, we must either order step 1 to be outside the interval $[0\ \infty]$ (which is impossible in this case) or force step 1: *Fly*() to preserve *At(A,E)*. The latter can be done by adding the preservation precondition $\neg In(A)$ to step 1 (because

if *A* is not in the rocket, then *A*'s position will not change when the rocket is flown). These three ways of shoring up the establishment are called (1) promotion, (2) demotion, and (3) confrontation, respectively.

## Hierarchical (HTN) Refinement

The refinements that we have seen till now treat all action sequences that reach the goal state as equivalent. In many domains, the users might have significant preferences among the solutions. For example, when I use my planner to make travel plans to go from Phoenix to Portland, I might not want a plan that involves bus rides. The question is how do I communicate such biases to the planner such that it will not waste any time progressing toward unwanted solutions? Although removing bus-ride action from the planner's library of actions is a possible solution, it might be too drastic. I might want to allow bus travel for shorter itineraries, for example.

One natural way is to introduce nonprimitive actions and restrict their reduction to primitive actions through user-supplied reduction schemas. Consider the example in figure 17. Here the nonprimitive action $Ship(o_1)$ has a reduction schema that translates it to a plan fragment containing three actions. Typically, there can be multiple possible legal reductions for a nonprimitive action. The reduction schemas restrict the planner's access to the primitive actions and, thus, stop progress toward undesirable solutions (Kambhampati and Srivastava 1996). A solution is considered desirable only if it can be parsed by the reduction schemas. Completeness is ensured only with respect to these desirable solutions (rather than all legal solutions).

For this method to work, we need the domain writer to provide us reduction schemas over and above domain actions. This requirement can be steep because the reduction schemas typically contain valuable control information that is not easily reconstructed from limited planning experience. However, one hope is that in domains where humans routinely build plans, such reduction knowledge can easily be elicited. Of course, acquiring the knowledge and verifying its correctness can still be nontrivial (Chien 1996).

## Tractability Refinements

All the refinements we have looked at until now are progressive in that they narrow the candidate set of a plan to which they are

applied. Many planners also use a variety of refinements that are not progressive (that is, have no pruning power). The motivation for their use is to reduce the plan-handling costs further; thus, we call them *tractability refinements.* Many of them can be understood as deriving the consequences of plan set constraints alone (without recourse to domain and planning theory) and splitting any disjunction among the plan constraints into the search space.

We can classify the tractability refinements into three categories: The first attempts to reduce the number of linearizations of the plan. In this category, we have *preordering refinements,* which order two unordered steps, and *prepositioning refinements,* which constrain the relative position of two steps.

Preordering refinements are illustrated in figure 18. The single partially ordered plan at the top of the figure is converted into two totally ordered plans below. Planners using preordering refinements include TOCL (Barrett and Weld 1994) and TO (Minton, Bresina, and Drummond 1994).

Prepositioning refinement is illustrated in figure 19, where one refinement considers the possibility of step 1 being contiguous to step 0, but the other considers the possibility of step 1 being noncontiguous to step 0. Planners such as STRIPS and PRODIGY use prepositioning refinements (to transfer the steps from the means-ends–analysis tree to the plan head; see Means-Ends Analysis).

The second category of tractability refinements attempts to make all linearizations safe with respect to auxiliary constraints. Here, we have *presatisfaction refinements,* which split a plan in such a way that a given auxiliary constraint is satisfied by all linearizations of the resulting components. Figure 20 illustrates a presatisfaction refinement with respect to the interval preservation constraint <0, *At(A,E)*, ∞>. To ensure that this constraint is satisfied in every linearization, the plan shown at the top is converted into the plan set with three components shown at the bottom. The first two components attempt to keep the step *Fly*() from intervening between 0 and ∞. The last component ensures that *Fly*() will be forced to preserve the condition *At(A,E)*. Readers might note a strong similarity between the presatisfaction refinements and the declobbering phase of plan-space refinement (see Position, Relevance, and Plan-Space Refinements). The important difference is that declobbering is done with respect to the condition that is established in the current refinement to ensure that the established
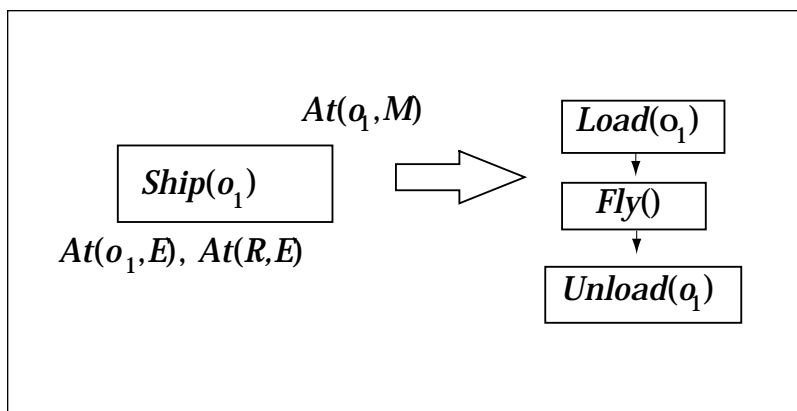


*Figure 17. Using Nonprimitive Tasks, Which Are Defined in Terms of Reductions to Primitive Tasks.*
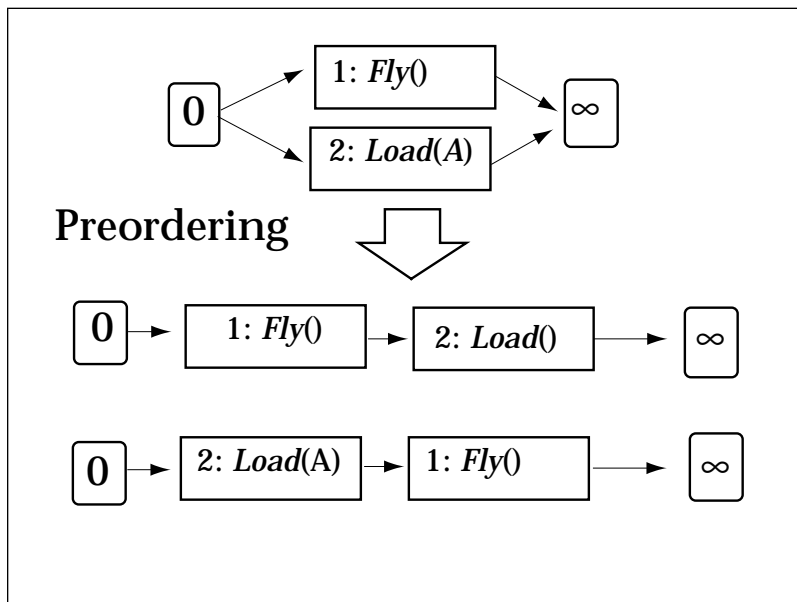


*Figure 18. Preordering Refinements.*

condition is not deleted by any step that is currently present in the plan. There is no guarantee that the steps that will be introduced by future refinements will continue to respect this establishment. In contrast, presatisfaction refinements are done to satisfy the interval-preservation constraints (presumably added by the bookkeeping phase of the plan-space refinement to protect an established condition). As long as the appropriate interval-preservation constraints are present, they will be enforced with respect to both existing steps and any steps that might be introduced

by future refinements. Many plan-space planners, including SNLP (McAllester and Rosenblitt 1991), UCPOP (Penberthy and Weld 1992), and NONLIN (Tate 1977) use presatisfaction refinements.

The third category of tractability refinements attempts to reduce uncertainty in the action identity. An example is *prereduction*

*refinement,* which converts a plan containing a nonprimitive action into a set of plans, each containing a different reduction of the nonprimitive action. Figure 21 illustrates the prereduction refinements. The plan at the top contains a nonprimitive action, *Ship*(*A*), which can, in principle, be reduced in a variety of ways to plan fragments containing only primitive actions. To reduce this uncertainty, prereduction refinements convert this plan to a plan set each of whose components corresponds to different ways of reducing *Ship*(*A*) action (in the context of figure 20, it is assumed that only one way of reducing *Ship*(*A*), viz., that shown in figure 17, is available).

Although tractability refinements as a whole seem to have weaker theoretical motivations than progressive refinements, it is worth noting that most of the prominent differences between existing algorithms boil down to differences in the use of tractability refinements. This point is illustrated in table 1, which characterizes several plan-space planners in terms of the specifics of the plan-space refinements they use (protection strategies, goal-selection strategies) and the type of tractability refinements they use.

### Interleaving Different Refinements

One of the advantages of the treatment of refinement planning that I presented is that it naturally allows for interleaving of a variety of refinement strategies in solving a single prob-
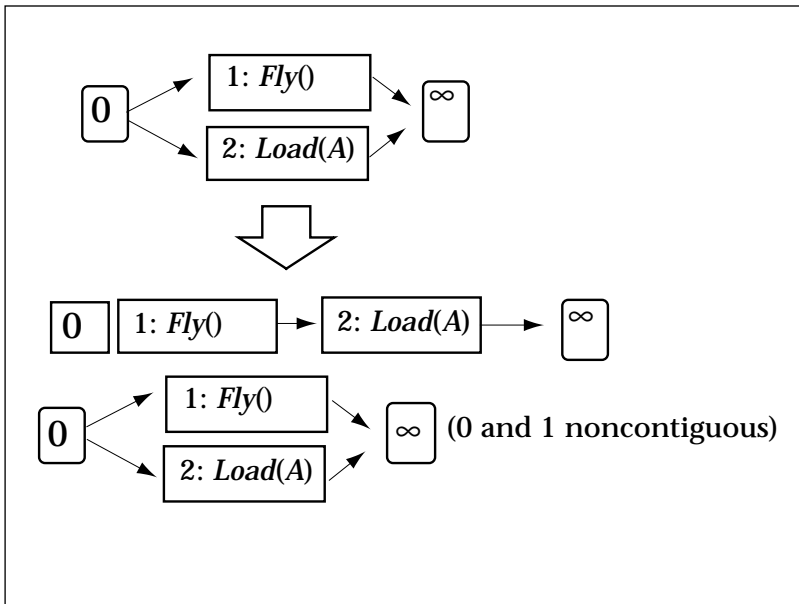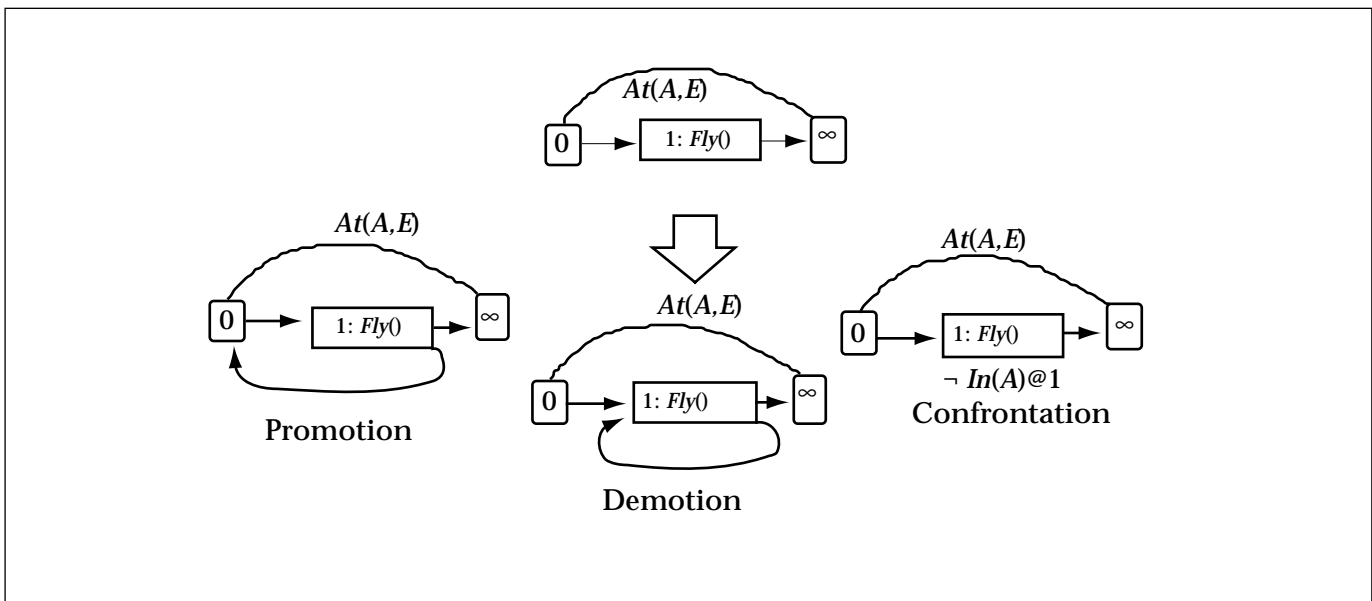


*Figure 19. Prepositioning Refinements.*



*Figure 20. Presatisfaction Refinements.*

# How Important Is Least Commitment?

One of the more hotly debated issues related to refinement planning algorithms is the role and importance of least commitment in planning. Informally, *least commitment* refers to the idea of constraining the partial plans as little as possible during individual refinements, with the intuition that overcommitting can eventually make the partial plan inconsistent, necessitating backtracking. To illustrate, in figure 16 we saw that individual components of state-space refinements tend to commit regarding both the absolute position and the relevance of actions inserted into the plan, but plan-space refinements commit to the relevance, leaving the position open by using precedence constraints.

Perhaps the first thing to understand about least commitment is that it has no special exclusive connection to ordering constraints (as is implied in some textbooks, where the phrase *least commitment planning* is used synonymously with partial order or plan-space planning). For example, a plan-space refinement that does (the optional) bookkeeping by imposing interval-preservation constraints is more constrained than a plan-space refinement that does not. Similarly, a hierarchical refinement that introduces an abstract action into the partial plan is less committed than a normal plan-space refinement that introduces only primitive actions (because a single abstract action can be seen as a stand-in for all the primitive actions that it can eventually be reduced to).

The second thing to understand about least commitment is that its utility depends on the nature of the domain. In general, commitment makes it easier to check if a partial plan contains a solution but increases the chance of backtracking. Thus, least commitment can be a winner in domains of low solution density and a loser in domains of high solution density.

The final, and perhaps the most important, thing to note about least commitment is that it makes a difference only when the planner splits the components of the plan sets into the search space. Although most traditional refinement planners do split plan set components, many recent planners such as GRAPHPLAN (Blum and Furst 1995; see Scale Up through Disjunctive Representations and Constraint-Satisfaction Techniques) handle plan sets without splitting, pushing most of the computation into the solution-extraction phase. In such planners, backtracking during refinement is not an issue (assuming that all refinements are complete), and thus, the level of commitment used by a refinement strategy does not directly affect the performance. What matters instead is the ease of extracting the solutions from the plan sets produced by the different refinements (which, in turn, might depend on factors such as the pruning power of the refinement, that is, how many candidates of the parent plan it is capable of eliminating from consideration) and the ease of propagating constraints on the plan set (see Scale Up through Disjunctive Representations and Constraint-Satisfaction Techniques).

lem. From a semantic viewpoint, because different refinement strategies correspond to different ways of narrowing the candidate sets, it is perfectly legal to interleave them. We can formally guarantee completeness of planning if each of the individual refinement strategies is complete. Figure 22 illustrates the solving of our rocket problem with the use of several refinements: We start with backward state space, then plan space, then forward state space, and then a preposition refinement.

Based on the specific interleaving strategy used, we can devise a whole spectrum of refinement planners, which differ from the existing single refinement planners. Our empirical studies (Kambhampati and Srivastava 1996, 1995) show that interleaving refinements this way can sometimes lead to superior performance over single-refinement planners.
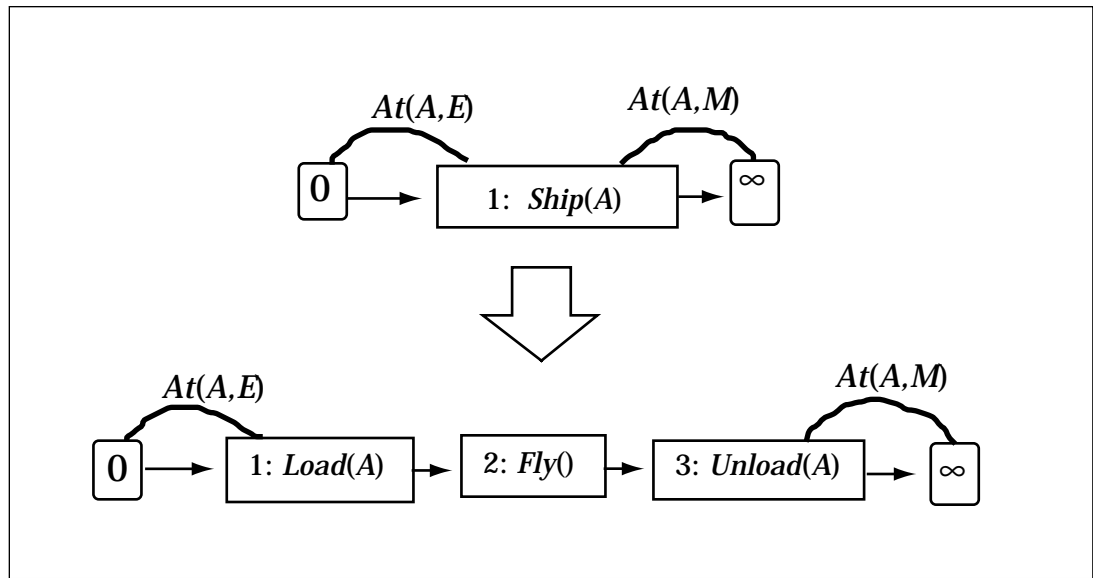
*Figure 21. Prereduction Refinements.*
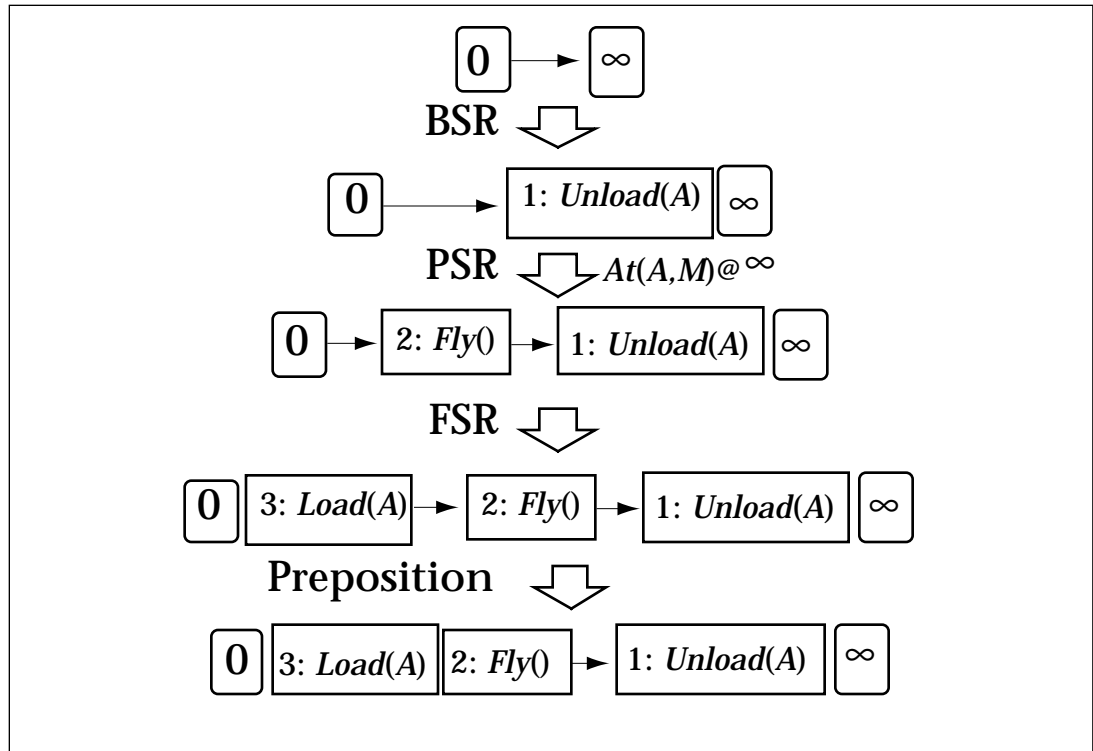


*Figure 22. Interleaving Refinements.*

It must be noted, however, that the issue of how one selects a refinement is largely open. We have tried refinement selection based on the number of components produced by each refinement, or the amount of narrowing of a candidate set each refinement affords, with some success.

## Trade-Offs in Refinement Planning

I described a parameterized refinement planning template that allows for a variety of specific algorithms depending on which refinement strategies are selected and how

| Planner | Goal Selection | Protection | Tractability refinements |
|---|---|---|---|
| TWEAK (Chapman 1987) | Based on modal truth criterion | none | None |
| SNLP (McAllester and Rosenblitt 1991), UCPOP (Penberthy and Weld 1992) | Arbitrary | IPCs to protect the established condition as well as its negation | presatisfaction |
| TOCL (Barrett and Weld, 1994) | Arbitrary | IPCs to protect the established condition as well as its negation | preordering |
| UA, TO (Minton, Bresina, and Drummond 1994) | Based on modal truth criterion | none | preordering (UA orders only interacting steps, but TO orders all pairs of steps) |

IPC = interval-preservation constraint.

*Table 1. A Spectrum of Plan-Space Planners.*

they are instantiated. We now attempt to understand the trade-offs governing some of these choices and see how one can go about choosing a planner, given a specific population of problems to solve. I concentrate here on the trade-offs in refinement planners that split plan set components into search space (see Introducing Splitting into Refinement Planning).

## Asymptotic Trade-Offs

Let us start with an understanding of the asymptotic trade-offs in refinement planning. I use an estimate of the search-space size in terms of properties of the plans at the fringe of the search tree (figure 23). Suppose $K$ is the total number of action sequences (to make $K$ finite, we can consider all sequences of or below a certain length). Let $F$ be the number of nodes on the fringe of the search tree generated by the refinement planner and $k$ be the average number of candidates in each of the plans on the fringe. Let $\rho$ be the number of times a given action sequence enters the candidate sets of fringe plans and $p$ be the *progress factor,* the fraction by which the candidate set narrows each time a refinement is done. We then have

$$F = p^d \times K \times \rho/k .$$

Because $F$ is approximately the size of the search space, it can also be equated to the search-space size derived from the effective branching factor $b$ and the effective depth $d$ of the generated search tree. Specifically,

$$F = p^d \times K \times \rho/k = b^d .$$

The time complexity of search can be written out as $C \times F$, where $C$ is the average cost of handling plan sets. $C$ itself can be broken down into two components: (1) $C_R$, the cost of applying refinements, and (2) $C_S$, the cost of extracting solutions from the plan. Thus,

$$T = (C_S + C_R) F .$$

These formulas can be used to understand the asymptotic trade-offs in refinement planning, as shown in figure 23.

For example, using refinement strategies with lower commitment (such as plan-space refinements as opposed to state-space refinements or plan-space refinements without bookkeeping strategies as opposed to plan-space refinements with bookkeeping strategies) leads to plans with higher candidate-set sizes and, thus, reduces $F$, but it can increase $C$. Using tractability refinements increases $b$ and, thus, increases $F$ but might reduce $C$ by reducing $C_S$ (because the tractability refinements reduce the variation among the linearizations of the plan, thereby facilitating cheaper solution extractors). The protection (bookkeeping) strategies reduce the redundancy factor $\rho$ and, thus, reduce $F$. However, they might increase $C$ because protection is done by adding more constraints, whose consistency needs to be verified.

Although instructive, the analysis of this section does not make conclusive predictions on practical performance because performance depends on the relative magnitudes of change in $F$ and $C$. To predict performance, we look at empirical evaluation.

## Empirical Study of Trade-Offs in Refinement Planning

The parameterized and unified understanding of refinement planning provided in this article allows us to ask specific questions about
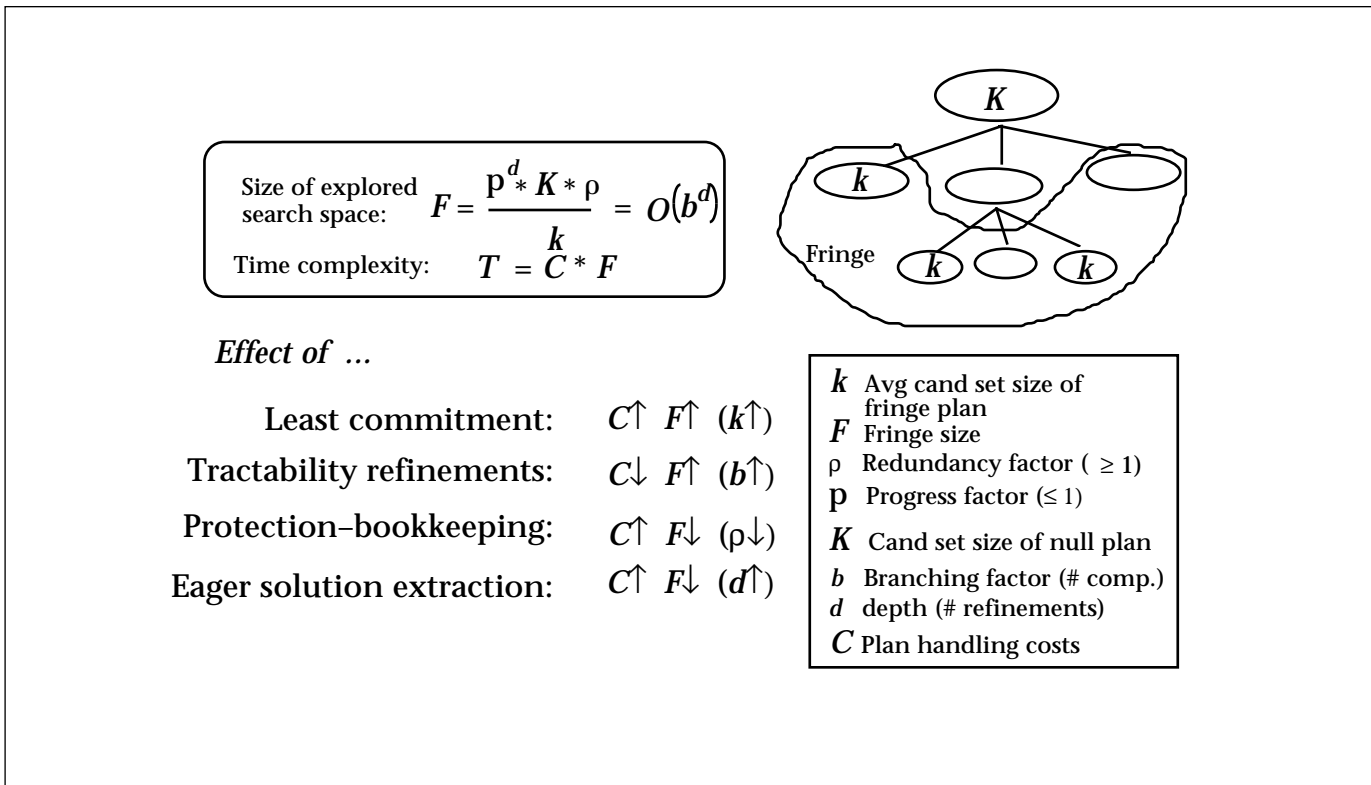
$$F = \frac{p^d * K * \rho}{k} = O(b^d)$$

Size of explored search space:

Time complexity: $T = C * F$

**Effect of ...**

| | |
|---|---|
| Least commitment: | $C{\uparrow}\ F{\uparrow}\ (k{\uparrow})$ |
| Tractability refinements: | $C{\downarrow}\ F{\uparrow}\ (b{\uparrow})$ |
| Protection–bookkeeping: | $C{\uparrow}\ F{\downarrow}\ (\rho{\downarrow})$ |
| Eager solution extraction: | $C{\uparrow}\ F{\downarrow}\ (d{\uparrow})$ |

$k$ Avg cand set size of fringe plan
$F$ Fringe size
$\rho$ Redundancy factor ($\geq 1$)
$p$ Progress factor ($\leq 1$)
$K$ Cand set size of null plan
$b$ Branching factor (# comp.)
$d$ depth (# refinements)
$C$ Plan handling costs

*Figure 23. Asymptotic Trade-Offs in Refinement Planning.*

the utility of specific design choices and answer them through normalized empirical studies. Here, we look at two choices—(1) tractability refinements and (2) bookkeeping (protection) strategies—because many existing planners differ along these dimensions.

Empirical results (Kambhampati, Knoblock, and Yang 1995) show that tractability refinements lead to reductions in search time only when the additional linearization they cause as a side effect also reduces the number of establishment possibilities. This reduction happens in domains where there are conditions that are asserted and negated by many actions. Results also show that protection strategies affect performance only in the cases where solution density is so low that the planner looks at the full search space.

In summary, for problems with normal solution density, performance differentials between planners are often attributable to differences in tractability refinements.

## Selecting among Refinement Planners Using Subgoal Interaction Analysis

Let us now turn to the general issue of selecting among refinement planners given a population of problems (constraining our attention once again to those planners that split the plan set components completely into the search space). We are, of course, interested in selecting a planner for which the given population of problems is easy. Thus, we need to relate the problem and planner characteristics to the ease of solving the problem by the planner. If we make the reasonable assumption that planners will solve a conjunctive goal problem by solving the individual subgoals serially (that is, develop a complete plan for the first subgoal and then extend it to also cover the second subgoal), we can answer this question in terms of the interactions between subgoals. Intuitively, two subgoals are said to interact if the planner might have to backtrack over a plan that it made for one subgoal to achieve both goals.

A *subplan* for a subgoal is a partial plan all of whose linearizations will execute and achieve the goal. Figure 24 shows two subplans for the *At(A,M)* goal in the rocket problem. Every refinement planner *R* can be associated with a class $P_R$ of subplans it is capable of producing for a subgoal. For example, for the goal *At(A,M)*, a planner using purely state-space refinements will produce prefix plans of the sort shown at the top of figure 24, which have steps only in the head, but a pure plan-space planner will produce elastic
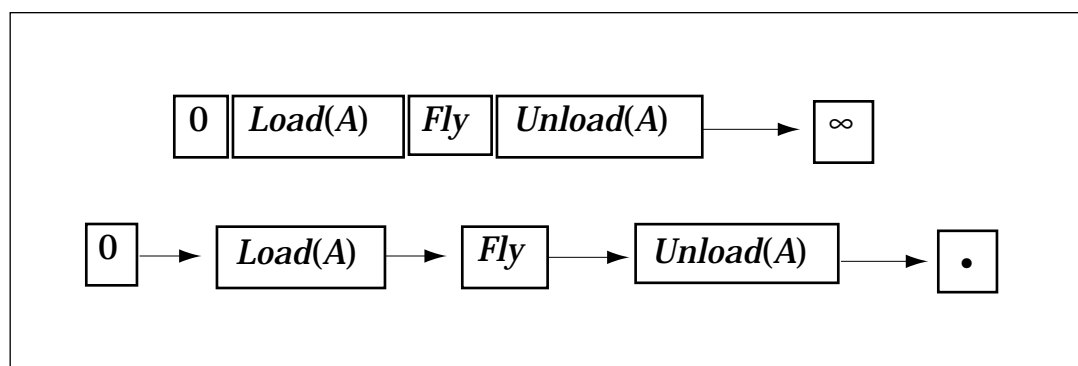
*Figure 24. Different Partial Plans for Solving the Same Subgoal.*

plans of the sort shown on the bottom, which only have steps in the middle, and new steps can be introduced in between existing ones.

The key question in solving two subgoals $G_1$ and $G_2$ serially is whether a subplan for $G_1$ in the given plan class is likely to be extended to be a subplan for the conjunctive goal $G_1$ and $G_2$. Two goals $G_1$ and $G_2$ are said to be trivially serializable (Kambhampati, Ihrig, and Srivastava 1996; Barrett and Weld 1994) with respect to a class of plans $P_R$ if every subplan of one goal belonging to $P_R$ can eventually be refined into a subplan for solving both goals. If all goals in a domain are pairwise trivially serializable with respect to the class of plans produced by a planner, then clearly, plan synthesis in this domain is easy for the planner (because the complexity will be linear in the number of goals).

It turns out that the level of commitment inherent in a plan class is an important factor in deciding serializability. Clearly, the lower the commitment of plans in a given class is, the higher the chance for trivial serializability. For example, the plan at the top of figure 24 cannot be extended to handle the subgoal $At(B,M)$, but the plan at the bottom can. Lower commitment explains why many domains with subgoal interactions are easier for plan-space planners than for state-space planners (Barrett and Weld 1994).

The preceding discussion does not imply a dominance of state-space planners by plan-space planners, however. In particular, the lower the commitment is, the higher the cost of handling plans in general is. Thus, the best guideline is to select the refinement planner with the highest commitment, and with respect to which class of (sub)plans, most goals in the domain are trivially serializable. Empirical studies show this strategy to be effective (Kambhampati, Ihrig, and Srivastava 1996).
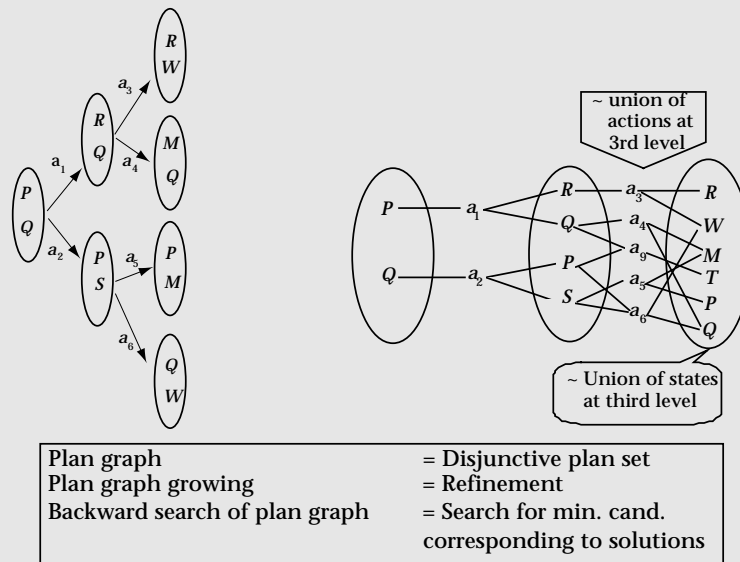
## Scaling Up Refinement Planners

Although refinement planning techniques have been applied to some complex real-world problems, such as beer brewing (Wilkins 1988), space-observation planning (Fuchs et. al. 1990), and spacecraft assembly (Aarup et. al., 1994), their widespread use has been inhibited to some extent by the fact that most existing planners scale up poorly when presented with large problems. Thus, there has been a significant emphasis on techniques for improving the efficiency of plan synthesis. One of these techniques involves improving performance by customizing the planner's behavior toward the problem population, and the second involves using disjunctive representations. Let me now survey the work in these directions.

### Scale Up through Customization

Customization can be done in a variety of ways. The first way is to bias the search of the planner with the help of control knowledge acquired from the user. As we discussed earlier, nonprimitive actions and reduction schemas are used, for the most part, to support such customization in the existing planners. There is now more research on the protocols for acquiring and analyzing reduction schemas (Chien 1996).

There is evidence that not all expert control knowledge is available in terms of reduction schemas. In such cases, incorporating the control knowledge into the planner can be tricky. One intriguing idea is to fold the control knowledge into the planner by automatically synthesizing planners—from domain specification and the declarative theory of refinement planning—using interactive software synthesis tools. I have started a project on implementing this approach using the KESTREL interactive software synthesis system, and the preliminary results have been

| Plan graph | = Disjunctive plan set |
| Plan graph growing | = Refinement |
| Backward search of plan graph | = Search for min. cand. |
| | corresponding to solutions |

*Understanding the GRAPHPLAN Algorithm as a Refinement Planner*
*Using State-Space Refinements over Disjunctive Partial Plans.*

# Understanding Graphplan

The GRAPHPLAN algorithm developed by Blum and Furst (1995) has generated a lot of excitement in the planning community on two counts: (1) it was by far the most efficient domain-independent planner on several benchmark problems and (2) its design seemed to have little in common with the traditional state-space and plan-space planners.

Within our general refinement planning framework, GRAPHPLAN can be seen as using forward state-space refinements without splitting the plan set components. We can elaborate this relation in light of our discussion of disjunctive representations. On the left of the figure is a state tree generated by a forward state-space planner that uses full splitting. On the right is the plan-graph structure generated by GRAPHPLAN for the same problem. Note that the plan graph can roughly be seen as a disjunction of the branches of the tree on the left. Specifically, the ovals representing the plan-graph proposition lists at a particular level can be seen as approximately the union of the states in the state-space tree at this level. Similarly, the actions at a given level in the plan graph can be seen as the union of actions on the various transitions at the level in the state tree. It is important to note that the relation is only approximate; for example, the action $a_9$ in the second action level and the proposition $T$ in the third level do not have any correspondence with the search tree. As explained in Refining Disjunctive Parts, this approximation is part of the price we pay for refining disjunctive plans directly. However, the propagation of mutual exclusion constraints allows GRAPHPLAN to keep a reasonably close cor-

respondence with the state tree without incurring the exponential space requirements of the state tree.

Viewing GRAPHPLAN in terms of our generalized refinement planning framework clarifies its sources of strength. For example, although Blum and Furst seem to suggest that an important source of GRAPHPLAN's strength is its ability to consider multiple actions at each time step, thereby generating parallel plans, this ability in itself is not new. As described in Forward State-Space Refinement, it is possible to generalize forward state-space refinement to consider sets of noninterfering actions simultaneously. Indeed, GRAPHPLAN's big win over traditional state-space planners (that do full splitting of plan set components into the search space) comes from its handling of plan set components without splitting, which, in turn, is supported by its use and refinement of disjunctive plans. Experiments with GRAPHPLAN confirm this hypothesis (Kambhampati and Lambrecht 1997).

The strong connection between forward state-space refinement and GRAPHPLAN also suggests that techniques such as means-ends analysis that have been used to focus forward state-space refinement can also be used to focus GRAPHPLAN. Indeed, we found that despite its efficiency, GRAPHPLAN can easily fail in domains where many actions are available, and only a few are relevant to the top-level goals of the problem. Thus, it would be interesting to deploy the best methods of means-ends analysis (such as the greedy regression graph proposed by McDermott [1996]) to isolate potentially relevant actions and only use them to grow the plan graph.
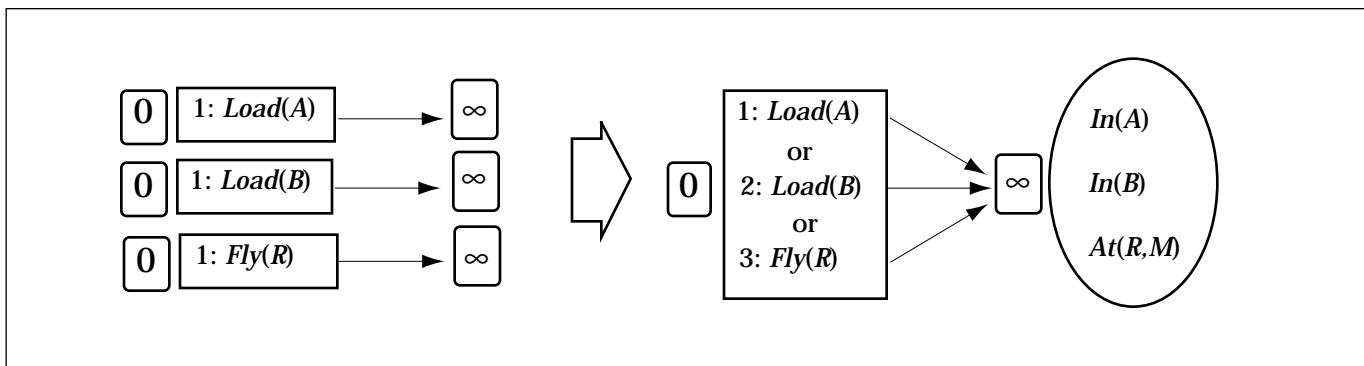
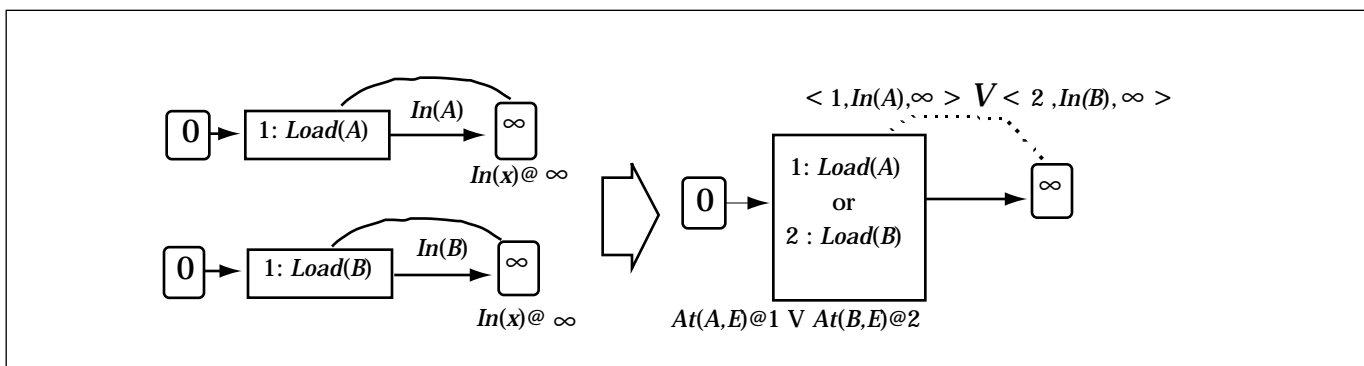*Figure 25. Disjunction over State-Space Refinements.*



*Figure 26. Disjunction over Plan-Space Refinements.*

promising (Srivastava and Kambhampati 1996).

Another way to customize is to use learning techniques and make the planner learn from its failures and successes. The object of learning might be the acquisition of search-control rules that advise the planner what search branch to pursue (Kambhampati, Katukam, and Qu 1996; Minton et. al. 1989) or the acquisition of typical planning cases that can then be instantiated and extended to solve new problems (Ihrig and Kambhampati 1996; Veloso and Carbonell 1993; Kambhampati and Hendler 1992). This area of research is active, and a sampling of papers can be found in the machine-learning sessions from the American Association for Artificial Intelligence conference and the International Joint Conference on Artificial Intelligence.

## Scale Up through Disjunctive Representations and Constraint-Satisfaction Techniques

Another way to scale up refinement planners is to directly address the question of search-space explosion. Much of this explosion is because all existing planners reflexively split the plan set components into the search space. We saw earlier with the basic refinement planning template that splitting is not required for completeness of refinement planning.

Let us examine the consequences of not splitting plan sets. Of course, we reduce the search-space size and avoid the premature commitment to specific plans. We also separate the action-selection and action-sequencing phases, so that we can apply scheduling techniques for the latter.

There can be two potential problems, however. First, keeping plan sets together can lead to unwieldy data structures. The way to get around this problem is to internalize the disjunction in the plan sets so that we can represent them more compactly (see later discussion). The second potential problem is that we might just be transferring the complexity from one place to another—from search-space size to solution extraction. However, solution extraction can be cast as a model-finding activity, and there have been a slew of efficient search strategies for propositional model finding (Crawford and Auton 1996; Selman, Mitchell, and Levesque 1992). I now elaborate on these ideas.

**Disjunctive Representations**  The general idea of disjunctive representations is to allow disjunctive step, ordering, and auxiliary constraints into a plan. Figures 25 and 26 show two examples that illustrate the compaction we can get through them. The three plans on the left in figure 25 can be combined into a single disjunctive step with disjunctive contiguity constraints. Similarly, the two plans in figure 26 can be compacted using a single disjunctive step constraint, a disjunctive precedence constraint, a disjunctive interval-preservation constraint, and a disjunctive point-truth constraint.

Candidate-set semantics for disjunctive plans follow naturally: The presence of the disjunctive constraint $c_1 \vee c_2$ in a partial plan constrains its candidates to be consistent with either the constraint $c_1$ or the constraint $c_2$.

Disjunctive representations clearly lead to a significant increase in the cost of plan handling. For example, in the disjunctive plan in figure 25, we don't know which of the steps will be coming next to 0, and thus, we don't know what the state of the world will be after the disjunctive step. Similarly, in the disjunctive plan in figure 26, we don't know whether steps 1 or 2 or both will be present in the eventual plan. Thus, we don't know whether we should work on the *At(A,E)* precondition or the *At(B,E)* precondition.

This uncertainty leads to two problems: First, how are we to refine disjunctive partial plans? Second, how are we to extract solutions from the disjunctive representation? We discuss the first problem in the following section. The second problem can be answered to some extent by posing the solution-extraction phase as a constraint-satisfaction problem (such as propositional satisfiability) and using efficient constraint-satisfaction engines. Recall that solution extraction merely involves finding a minimal candidate of the plan that is a solution (in the sense that the preconditions of all the steps, including the final goal step, are satisfied in the states preceding them).

**Refining Disjunctive Plans**  To handle disjunctive plans directly, we need to generalize the particulars of the refinement strategies because the standard refinement strategies are clearly developed only for partial plans without disjunction (see Existing Refinement Strategies). For example, for the disjunctive plan on the right in figure 26, we don't know which of the steps will be coming next to 0, and thus, we don't quite know what the state of the world will be after the disjunctive step. Thus, the forward state-space refinement will

not know which actions should next be applied to the plan prefix. Similarly, for the disjunctive plan in figure 26, we don't know whether steps 1 or 2 or both will be present in the eventual plan. Thus, a plan-space refinement won't know whether it should work on the *At(A,E)* precondition or the *At(B,E)* precondition or both.[6]

One way of refining such plans is to handle the uncertainty in a conservative fashion. For example, for the plan in figure 25, although we do not know the exact state after the first (disjunctive) step, we know that it can only be a subset of the union of conditions in the effects of the three steps. Knowing that only 1, 2, or 3 can be the first steps in the plan tells us that the state after the first step can only contain the conditions *In(A)*, *In(B)*, and *At(R,M)*. Thus, we can consider a generalization of forward state-space refinement that adds only those actions whose preconditions are subsumed by the union of the effects of the three steps.

This variation is still complete but will be less progressive than the standard version that operates on nondisjunctive plan sets. It is possible that even though the preconditions of an action are in the union of effects, there is no real way for the action to take place. For example, although the preconditions of the "unload at moon" action might seem satisfied, it is actually never going to occur as the second step in any solution because *Load*() and *Fly*() cannot be done at the same time. This example brings up an important issue: Disjunctive plans can be refined at the expense of some of the progressivity of the refinement.

Although the loss of progressivity cannot be avoided, it can be reduced to a significant extent by generalizing that refinements infer more than action constraints. In the example, the refinement can infer that steps 1 and 3 cannot both occur in the first step (because their preconditions and effects are interacting). This interference tells us that the second state might have either *In(A)* or *At(R,M)* but not both. Here, the interaction between steps 1 and 3 propagates to make the conditions *In(A)* and *At(R,M)* mutually exclusive in the next disjunctive state. Thus, any action that needs both *In(A)* and *At(R,M)* can be ignored in refining this plan. This example uses constraint propagation to reduce the number of refinements generated. The particular strategy shown here is used by Blum and Furst's (1995) GRAPHPLAN algorithm (see the sidebar).

Similar techniques can be used to refine the disjunctive plan in figure 26. For exam-

| Planner | Refinement | Splitting ($k$) |
|---|---|---|
| UCPOP (Penberthy and Weld 1992), SNLP (McAllester and Rosenblitt 1991) | Plan space | $k$ = #Comp |
| TOPI (Barrett and Weld 1996) | Backward state space | $k$ = #Comp |
| GRAPHPLAN (Blum and Furst, 1995) | Forward state space | $k$ = 1 |
| UCPOP-D (Kambhampati and Yang 1996), DESCARTES (Joslin and Pollack 1996) | Forward state space | $1 < k <$ #Comp |

*Table 2. A Spectrum of Refinement Planners.*

ple, knowing that either 1 or 2 must precede the last step and give the condition *In(x)* tells us that if 1 doesn't, then 2 must. This inference reduces the number of establishment possibilities that plan-space refinement has to consider at the next iteration.

**Open Issues in Planning with Disjunctive Representations**     In the last year or so, several efficient planners have been developed that can be understood in terms of disjunctive representations. These planners include GRAPHPLAN (Blum and Furst 1995), SATPLAN (Kautz and Selman 1996), DESCARTES (Joslin and Pollack 1996), and UCPOP-D (Kambhampati and Yang 1996).

However, many issues need careful attention (Kambhampati 1997). For example, a study of the constraint-satisfaction literature shows that propagation and refinement can have synergistic interactions. A case in point is the eight-queens problem, where constraint propagation becomes possible only after we commit to the placement of at least one queen. Thus, the best planners might be doing controlled splitting of plan sets (rather than no splitting at all) to facilitate further constraint propagation. The following generalized refinement planning template supports controlled splitting of plan sets:

Refine (*P:* [a disjunctive] plan set)

0*. If <<*P*>> is empty, Fail.

1. If a minimal candidate of *P* is a solution, terminate.

2. Select a refinement strategy *R.*
   Apply *R* to *P* to get a new plan set *P′*.

3. Split *P′* into $k$ plan sets.

4. Simplify plan sets by doing constraint propagation.

5. Nondeterministically select one of the plan sets $P′_i$.

Call Refine ($P′_i$).

Step 3 in the algorithm splits a plan set into $k$ components. Depending on the value of $k$, as well as the type of refinement strategy selected in step 2, we can get a spectrum of refinement planners, as shown in table 2. In particular, the traditional refinement planners that do complete splitting can be modeled by choosing $k$ to be equal to the number of components of the plan set. Planners such as GRAPHPLAN that do not split can be modeled by choosing $k$ to be equal to 1.

In between are the planners such as DESCARTES and UCPOP-D that split a plan set into some number of branches that is less than the number of plan set components. One immediate question is exactly how many branches should a plan set be split into? Because the extent of propagation depends on the amount of shared substructure between the disjoined plan set components, one way of controlling splitting would be to keep plans with shared substructure together.

The relative support provided by various types of refinement for planning with disjunctive representations needs to be understood. The trade-off analyses described in Trade-Offs in Refinement Planning need to be generalized to handle disjunctive representations. The analyses based on commitment and ensuing backtracking are mostly inadequate when we do not split plan set components. Nevertheless, specific refinement strategies can have a significant effect on the performance of disjunctive planners. As an example, consider the fact that although GRAPHPLAN, which does forward state-space refinement on disjunctive plans, is efficient, no comparable planner does backward state-space refinements.

Finally, the interaction between the refinements and the solution-extraction process needs to be investigated more carefully.
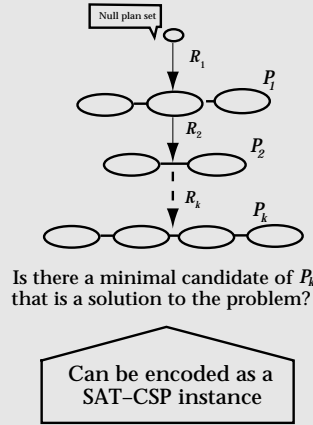
If $R_1, R_2, ..., R_k$ are all complete
( and progressive)
Then,
Minimal candidates of $P_k$ will
contain all $k$-length solutions

Is there a minimal candidate of $P_k$
that is a solution to the problem?

Can be encoded as a
SAT–CSP instance

Shape of the encoding depends on
– Refinements $R_i$
– Representations for plan sets
– Disjunctive–nondisjunctive

*Relating Refined Plan at the* k*th Level to* SATPLAN *Encodings.*

# Refinement Planning versus Encoding Planning as Satisfiability

Recently, Kautz et. al. (Kautz and Selman 1996; Kautz, McAllester, and Selman 1996) have advocated solving planning problems by encoding them first as SAT problems and then using efficient SAT solvers such as GSAT to solve them. Their approach involves generating a SAT encoding, all models of which will correspond to $k$-length solutimons to the problem (for some fixed integer $k$). Model finding is done by efficient SAT solvers such as GSAT (Selman, Levesque, and Mitchell 1992). Kautz et. al. propose to start with some arbitrary value of $k$ and increase it if they do not find solutions of this length. They have considered a variety of ways of generating the encodings.

In the context of the general refinement planning framework, I can offer a rational basis for the generation of the various encodings. Specifically, the natural place where SAT solvers can be used in refinement planning is in the solution-extraction phase. As illustrated in the first figure, after doing $k$ complete and progressive refinements on a null plan, we get a plan set whose minimal candidates contain all $k$-length solutions to the problem. Thus, picking a solution boils down to searching through the minimal candidates—which can be cast as a SAT problem. This account naturally relates the character of the encodings to what refinements are used in coming with the $k$-length plan set and how the plan sets themselves are represented (recall that disjunctive representations can reduce the progressivity of refinements).

Kautz and his colleagues concentrate primarily on direct translation of planning problems to SAT encodings, sidelining refinement planning. I believe that such an approach confounds two orthogonal issues: (1) reaching a compact plan set whose minimal candidates contain all the solutions and (2) posing the solution extraction as a SAT problem. The issues guiding the first are by and large specific to planning and are best addressed in terms of refining disjunctive plans. The critical issue in posing the solution extraction as a SAT problem is to achieve compact SAT encodings. The techniques used to address this issue—such as converting $n$-ary predicates into binary predicates or compiling out dependent variables—are generic and are only loosely tied to planning.

It is thus best to separate plan set construction and its compilation into a SAT instance. Such a separation allows SATPLAN techniques to exploit, rather than reinvent, (disjunctive) refinement planning. Their emphasis can then shift toward understanding the trade-offs offered by SAT encodings based on disjunctive plans derived by different types of refinement. In addition, basing encodings on $k$th-level plan sets can also lead to SAT instances that are smaller on the whole. Specifically, the pruning done by disjunctive refinements can also lead to a plan set with a fewer number of minimal candidates and, consequently, a tighter encoding than can be achieved by direct translation. This conjecture is supported to some extent by the results reported in Kautz and Selman (1996) that show that SAT encodings based on $k$-length planning graphs generated by GRAPHPLAN can be solved more efficiently than the linear encodings generated by direct translation.

As I indicated in the preceding discussion, if we have access to efficient solution-extraction procedures, we can, in theory, reduce the role of refinements significantly. For example, the disjunctive plan in figure 25 can be refined by simply assuming that every step is a disjunction of all the actions in the library. The solution-extraction function will then have to sort through the large disjunctions and find the solutions. To a first approximation, this is the way the state-space and plan-space encodings of SATPLAN (Kautz and Selman 1996) work. The approach we sketched involves using the refinement strategies to reduce the amount of disjunction, for example, to realize that the second step can only be drawn from a small subset of all library actions, and then using SAT methods to extract solutions from the resulting plan. I speculate that this approach will scale up better by combining the forces of both refinement and efficient solution extraction.

## Conclusion and Future Directions

Let me conclude by reiterating that refinement planning continues to provide the theoretical backbone for most of the AI planning techniques. The general framework I presented in this article allows a coherent unification of all classical planning techniques, provides insights into design trade-offs, and outlines avenues for the development of more efficient planning algorithms.

Perhaps equally important, a clearer understanding of refinement planning under classical assumptions will provide us valuable insights into planning under nonclassical assumptions. Indeed, the operation of several nonclassical planners described in the literature can be understood from a refinement planning point of view. These planners include probabilistic least commitment planners such as BURIDAN (Kushmerick, Hanks, and Weld 1995) and planners dealing with partially accessible environments such as XII (Golden, Etzioni, and Weld 1996).

A variety of exciting avenues are open for further research in refinement planning (Kambhampati 1997). First, we have seen that despite the large number of planning algorithms, there are really only two fundamentally different refinement strategies: (1) plan space and (2) state space. It would be interesting to see if there are novel refinement strategies with better properties. To some extent, this formulation of new refinements can depend on the type of representations we use

for partial plans. In a recent paper, Ginsberg (1996) describes a partial plan representation and refinement strategy that differs significantly from the state-space and plan-space strategies, although its overall operation conforms to the framework of refinement followed by the solution extraction described here. It will be interesting to see how it relates to the classical refinements.

We also do not understand enough about what factors govern the selection of specific refinement strategies. We need to take a fresh look at trade-offs in plan synthesis, given the availability of planners using disjunctive representations. Concepts such as subgoal interaction do not make too much sense in these scenarios.

Finally, much can be gained by porting the refinement planning techniques to planning under nonclassical scenarios. For example, how can we use the analogs of hierarchical refinements, and disjunctive representations, to make planning under uncertainty more efficient?

## Further Reading

For the ease of exposition, I have simplified the technical details of the refinement planning framework in some places. For a more formal development of the syntax and semantics of refinement planning, see Kambhampati, Knoblock, and Yang (1995). For the details of unifying and interleaving state-space, plan-space, and hierarchical refinements, see Kambhampati and Srivastava (1996). For a more technical discussion of the issues in disjunctive planning, see Kambhampati and Yang (1996) and Kambhampati (1997). Most of these papers can be found at rakaposhi.eas.asu.edu/yochan.html. A more global overview of the areas of planning and scheduling can be found in Dean and Kambhampati (1996).

Pednault (1994) is the best formal introduction to the syntax and semantics of ADL. Barrett and Weld (1994) report on a comprehensive analysis of the trade-offs between state-space and plan-space planning algorithms. Minton, Bresina, and Drummond (1994) compare partial-order and total-order plan-space planning algorithms. This study focuses on the effect of preordering tractability refinements. Kambhampati, Knoblock, and Yang (1995) provide some follow-up results on the effect of tractability refinements on planner performance.

There are also several good sources for more detailed accounts of individual

*Perhaps equally important, a clearer understanding of refinement planning under classical assumptions will provide us valuable insights into planning under nonclassical assumptions.*

refinement planning algorithms. Weld (1994) is an excellent tutorial introduction to partial order planning. Erol (1995) provides theoretically clean formalization of the hierarchical planning algorithms. Penberthy and Weld (1994) describe a refinement planner that can handle deadlines and continuous change. Wellman (1987) proposes a general template for planning under uncertainty based on dominance proving that is similar to the refinement planning template discussed here.

Although we concentrated on plan synthesis in classical planning, the theoretical models developed here are also helpful in explicating the issues in replanning and plan reuse as well as the interleaving of planning and execution. For a more global account of the area of automated planning that meshes well with the unifying view described in this article, you might refer to the online notes from the graduate-level course on planning that I teach at Arizona State University. The notes can be found at rakaposhi.eas.asu.edu/planning-class.html.

In addition to the national and international AI conferences, planning-related papers also appear in the biannual International Conference on AI Planning Systems and the European Conference (formerly European Workshop) on Planning Systems. There is a mailing list for planning-related discussions and announcements. For a subscription, e-mail a message to planning@asu.edu.

## Acknowledgments

## Notes

1. In the context of planning, the frame problem refers to the idea that a first-order logic–based description of actions must not only state what conditions are changed by an action but also what conditions remain unchanged after the action. In any sufficiently rich domain, many conditions are left unchanged by an action, thus causing two separate problems: First, we might have to write the so-called frame axioms for each of the action-unchanged condition pairs. Second, the theorem prover has to use these axioms to infer that the unchanged conditions in fact remained the same. Although the first problem can be alleviated using domain-specific frame axioms (Haas 1987) that state for each condition the circumstances under which it changes, the second problem cannot be resolved so easily. Any general-purpose theorem prover would have to explicitly use the frame axioms to prove the persistence of conditions.

2. It is perhaps worth repeating that this representation for partial plans is sufficient but not necessary. I use it chiefly because it subsumes the partial plan representations used by most existing planners. For a significantly different representation of partial plans, which can also be given candidate set–based semantics, the reader is referred to some recent work by Ginsberg (1996).

3. As readers familiar with partial-order–planning literature will note, I am simplifying the representation by assuming that all actions are fully instantiated, thus ignoring codesignation and noncodesignation constraints between variables. Introduction of variables does not significantly change the nature of refinement planning.

4. David Chapman's influential 1987 paper on the foundations of nonlinear planning has unfortunately caused some misunderstandings about the nature of plan-space refinement. Specifically, Chapman's account suggests that the use of a modal truth criterion is de rigeur for doing plan-space planning. A *modal truth criterion* is a formal specification of the necessary and sufficient conditions for ensuring that a state variable will have a particular value in the state preceding (or following) a given action in a *partially ordered plan* (that is, a partial plan containing actions ordered by precedence constraints). Chapman's idea is to make the truth criterion the basis of plan-space refinement. This idea involves first checking to see if every precondition of every action in the plan is true according to the truth criterion. For each precondition that is not true, the planner then considers adding all possible combinations of additional constraints (steps, orderings) to the plan to make them true. Because interpreting the truth criterion

turns out to be NP-hard when the actions in the plan can have conditional effects, Chapman's work has led to the belief that the cost of an individual plan-space refinement can be exponential.

The fallacy in this line of reasoning becomes apparent when we note that checking the truth of a proposition in a partially ordered plan is never necessary for solving the classical planning problem (whether by plan-space or some other refinement) because the solutions to a classical planning problem are action sequences! The partial ordering among steps in a partial plan constrains the candidate set of the partial plan and is not to be confused with action parallelism in the solutions. Our account of plan-space refinement avoids this pitfall by not requiring the use of a modal truth criterion in the refinement. For a more elaborate clarification of this and other formal problems about the nature and role of modal truth criteria, the reader is referred to Kambhampati and Nau (1995).

5. A common misrepresentation of the state-space and plan-space refinements in the early planning literature involved identifying plan-space refinements with plans where the actions are partially ordered and identifying state-space refinements with plans where the actions are totally ordered. As the description here shows, the difference between state-space and plan-space refinements is better understood in terms of precedence and contiguity constraints. These constraints differ primarily in whether new actions are allowed to intervene between a pair of ordered actions: Precedence relations allow an arbitrary number of additional actions to intervene, but contiguity relations do not.

A planner employing plan-space refinement—and, thus, using precedence relations—can produce totally ordered partial plans if it uses preordering based tractability refinements (see Tractability Refinements). Examples of such planners include TOCL (Barrett and Weld 1994) and TO (Minton, Bresina, and Drummond 1994). Similarly, a planner using state-space refinements can produce partial plans with some actions unordered with respect to each other if it uses the generalized state-space refinement that considers sets of noninterfering actions together in one plan set component rather than in separate ones (see Forward State-Space Refinement).

6. As I noted earlier, hierarchical refinement introduces nonprimitive actions into a partial plan. The presence of a nonprimitive action can be interpreted as a disjunctive constraint on the partial plan—effectively stating that the partial plan must contain all the primitive actions corresponding to at least one of the eventual reductions of the nonprimitive task. Despite this apparent similarity, traditional HTN planners differ from the disjunctive planners discussed here in that they eventually split disjunction into the search space with the help of prereduction refinements, considering each way of reducing the nonprimitive task in a different search branch. Solution extraction is done only on nondisjunctive plans. The utility of the disjunc-

tion in this case is primarily to postpone the branching to lower levels of the search tree. In contrast, disjunctive planners can (and perhaps should) do solution extraction directly from disjunctive plans.

## References

Aarup, M.; Arentoft, M. M.; Parrod, Y.; and Stokes, I. 1994. OPTIMUM-AIV: A Knowledge-Based Planning and Scheduling System for Spacecraft AIV. In *Intelligent Scheduling,* eds. M. Fox and M. Zweben, 451–470. San Francisco, Calif.: Morgan Kaufmann.

Bacchus, F., and Kabanza, F. 1995. Using Temporal Logic to Control Search in a Forward-Chaining Planner. In *Proceedings of the Third European Workshop on Planning.* Amsterdam: IOS.

Barrett, A., and Weld, D. 1994. Partial-Order Planing: Evaluating Possible Efficiency Gains. *Artificial Intelligence* 67(1): 71–112.

Blum, A., and Furst, M. 1995. Fast Planning through Plan-Graph Analysis. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Chapman, D. 1987. Planning or Conjunctive Goals. *Artificial Intelligence* 32(3): 333–377.

Chien, S. 1996. Static and Completion Analysis for Planning Knowledge Base Development and Verification. In *Proceedings of the Third International Conference on AI Planning Systems,* 53–61. Menlo Park, Calif.: AAAI Press.

Crawford, J., and Auton, L. 1996. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence* 81.

Dean, T., and Kambhampati, S. 1996. Planning and Scheduling. In *CRC Handbook or Computer Science and Engineering.* Boca Raton, Fla.: CRC Press.

Drummond, M. 1989. Situated Control Rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning,* 103–113. San Francisco, Calif.: Morgan Kaufmann.

Erol, K. 1995. Hierarchical Task Network Planning Systems: Formalization, Analysis, and Implementation. Ph.D. diss., Department of Computer Science, University of Maryland.

Erol, K.; Nau, D.; and Subrahmanian, V. 1995. Complexity, Decidability, and Undecidability Results for Domain-Independent Planning. *Artificial Intelligence* 76(1–2): 75–88.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3–4): 189–208.

Fuchs, J. J.; Gasquet, A.; Olalainty, B. M.; and Currie, K. W. 1990. PLANERS-1: An Expert Planning System for Generating Spacecraft Mission Plans. In Proceedings of the First International Conference on Expert Planning Systems, 70–75. Brighton, U.K.: Institute of Electrical Engineers.

Ginsberg, M. 1996. A New Algorithm for Generative Planning. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representa-*

*tion and Reasoning,* 186–197. San Francisco, Calif.: Morgan Kaufmann.

Golden, K.; Etzioni, O.; and Weld, D. 1996. XII*:* Planning with Universal Quantification and Incomplete Information, Technical Report, Department of Computer Science and Engineering, University of Washington.

Green, C. 1969. Application of Theorem Proving to Problem Solving. In Proceedings of the First International Joint Conference on Artificial Intelligence, 219–239. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Haas, A. 1987. The Case for Domain-Specific Frame Axioms. In *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop,* ed. F. M. Brown. San Francisco, Calif.: Morgan Kaufmann.

Ihrig, L., and Kambhampati, S. 1996. Design and Implementation of a Replay Framework Based on a Partial-Order Planner. In Proceedings of the Thirteenth National Conference on Artificial Intelligence, 849–854. Menlo Park, Calif.: American Association for Artificial Intelligence.

Joslin, D., and Pollack, M. 1996. Is "Early Commitment" in Plan Generation Ever a Good Idea? In Proceedings of the Thirteenth National Conference on Artificial Intelligence, 1188–1193. Menlo Park, Calif.: American Association for Artificial Intelligence.

Kambhampati, S. 1997. Challenges in Briding Plan Synthesis Paradigms. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence. Forthcoming.

Kambhampati, S., and Hendler, J. 1992. A Validation Structure–Based Theory of Plan Modification and Reuse. *Artificial Intelligence* 55(2–3): 193–258.

Kambhampati, S., and Lambrecht, E. 1997. Why Does GRAPHPLAN Work? Technical Report, ASU CSE TR 97-005, Department of Computer Science, Arizona State University.

Kambhampati, S., and Nau, D. 1995. The Nature and Role of Modal Truth Criteria in Planning. *Artificial Intelligence* 82(1–2): 129–156.

Kambhampati, S., and Srivastava, B. 1996. Unifying Classical Planning Approaches, Technical Report, 96-006, Department of Computer Science and Engineering, Arizona State University.

Kambhampati, S., and Srivastava, B. 1995. Universal Classical Planner: An Algorithm for Unifying State-Space and Plan-Space Planning. In *Proceedings of the Third European Workshop on Planning.* Amsterdam: IOS.

Kambhampati, S., and Yang, X. 1996. On the Role of Disjunctive Representations and Constraint Propagation in Refinement Planning. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning,* 35–147. San Francisco, Calif.: Morgan Kaufmann.

Kambhampati, S.; Ihrig, L.; and Srivastava, B. 1996. A Candidate Set–Based Analysis of Subgoal Interaction in Conjunctive Goal Planning. In *Proceedings of the Third International Conference on AI Planning*

*Systems,* 125–133. Menlo Park, Calif.: AAAI Press.

Kambhampati, S.; Katukam, S.; and Qu, Y. 1996. Failure-Driven Dynamic Search Control for Partial-Order Planners: An Explanation-Based Approach. *Artificial Intelligence* 88(1–2): 253–315.

Kambhampati, S.; Knoblock, C.; and Yang, Q. 1995. Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning. *Artificial Intelligence* 76(1–2): 167–238.

Kautz, H., and Selman, B. 1996. Pushing the Envelope: Planning Propositional Logic and Stochastic Search. In Proceedings of the Thirteenth National Conference on Artificial Intelligence, 1194–1201. Menlo Park, Calif.: American Association for Artificial Intelligence.

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding Plans in Propositional Logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning,* 374–385. San Francisco, Calif.: Morgan Kaufmann.

Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An Algorithm for Probabilistic Least Commitment Planning. *Artificial Intelligence* 76(1–2).

McAllester, D., and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In Proceedings of the Ninth National Conference on Artificial Intelligence, 634–639. Menlo Park, Calif.: American Association for Artificial Intelligence.

McDermott, D. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In *Proceedings of the Third International Conference on AI Planning Systems,* 142–149. Menlo Park, Calif.: AAAI Press.

Minton, S.; Bresina, J.; and Drummond, M. 1994. Total-Order and Partial-Order Planning: A Comparative Analysis. *Journal of Artificial Intelligence Research* 2:227–262.

Minton, S.; Carbonell, J. G.; Knoblock, C.; Kuokka, D. R.; Etzioni, O.; and Gil, Y. 1989. Explanation-Based Learning: A Problem-Solving Perspective. *Artificial Intelligence* 40:363–391.

Nilsson, N. 1980. *Principles of Artificial Intelligence.* San Francisco, Calif.: Morgan Kaufmann.

Pearl, J. 1984. *Heuristics.* Reading, Mass.: Addison-Wesley.

Pednault, E. 1994. ADL and the State-Transition Model of Action. *Journal of Logic and Computation* 4(5): 467–512.

Pednault, E. 1988. Synthesizing Plans That Contain Actions with Context-Dependent Effects. *Computational Intelligence* 4(4): 356–372.

Penberthy, S., and Weld, D. 1994. Temporal Planning with Continuous Change. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1010–1015. Menlo Park, Calif.: American Association for Artificial Intelligence.

Penberthy, S., and Weld, D. 1992. UCPOP: A Sound, Complete, Partial-Order Planner for ADL. In Proceedings of the Third International Conference on the Principles of Knowledge Representation, 103–114. San Francisco, Calif.: Morgan Kaufmann.

Sacerdoti, E. 1975. The Non-Linear Nature of Plans.

In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 206–214. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Selman, D.; Levesque, H. J.; and Mitchell, D. 1992. GSAT: A New Method for Solving Hard Satisfiability Problems. In Proceedings of the Tenth National Conference on Artificial Intelligence, 440–446. Menlo Park, Calif.: American Association for Artificial Intelligence.

Srivastava, B., and Kambhampati, S. 1996. Synthesizing Customized Planners from Specifications, Technical Report, 96-014, Department of Computer Science and Engineering, Arizona State University.

Tate, A. 1977. Generating Project Networks. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 888–893. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Tate, A. 1975. Interacting Goals and Their Use. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 215–218. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Veloso, M., and Carbonell, J. 1993. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Machine Learning* 10:249–278.

Weld, D. 1994. An Introduction to Least Commitment Planning. *AI Magazine* 15(4): 27–61.

Wellman, M. 1987. Dominance and Subsumption in Constraint-Posting Planning. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence, 884–890. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Wilkins, D. 1988. *Practical Planning.* San Francisco, Calif.: Morgan Kaufmann.

Wilkins, D. E. 1984. Domain-Independent Planning: Representation and Plan Generation. *Artificial Intelligence* 22(3).

**Subbarao Kambhampati** is an associate professor at Arizona State University, where he directs the Yochan Research Group, investigating planning, learning, and case-based reasoning. He is a 1994 National Science Foundation Young Investigator awardee. He received his B.Tech from the Indian University of Technology, Madras, and his M.S. and Ph.D. from the University of Maryland at College Park. His e-mail address is rao@asu.edu.

# Proceedings of the Eighth Midwest AI and Cognitive Science Conference

Dayton, Ohio, May 30 - June 1, 1997

*Edited by Eugene Santos, Jr.*

The Midwest Artificial Intelligence and Cognitive Science Conference (MAICS) is a general forum for research in both of the above-mentioned fields and provides an opportunity for interdisciplinary cooperation and discussion between these two fields. Like the previous seven annual meetings, the Eighth MAICS has encouraged submissions and participation from those new to the research community.

The papers presented in 1997 can be roughly categorized into the following eight subfields in AI and cognitive science: knowledge representation, intelligent tutoring systems, uncertainty, knowledge and data mining, virtual environments and intelligent actors, vision, language and interface, and self-organizing maps.