Refinement Planning as a unifying framework for plan synthesis

Subbarao Kambhampati
Department of Computer Science
Arizona State University
Tempe, AZ 85287-5406
http://rakaposhi.eas.asu.edu/yochan.html

Table of Contents

1. INTRODUCTION 2

2. PLANNING AND CLASSICAL PLANNING 3

2.1 Modeling Actions and states 4

2.2 Action Application 5

2.3 Verifying a solution 6

2.4 Chronology of classical planning approaches 6

3. REFINEMENT PLANNING: OVERVIEW 7

3.1 Partial Plan Representation: Syntax 9

3.2 Partial Plan Representation: Semantics 10

3.3 Connecting Syntax and Semantics of Partial Plans 11

3.4 Refinement Strategies 12

3.5 Planning using Refinement Strategies and solution extraction 13

3.6 Introducing Search into Refinement Planning 14

4. EXISTING REFINEMENT STRATEGIES 15

4.1 Forward State Space Refinement 16
4.1.1 Generalizing Forward State Space Refinement to allow parallel actions 18

4.2 Making state-space refinements goal-directed 18
4.2.1 Means-ends analysis 19
4.2.2 Backward State Space Refinement 20

4.3 Position, Relevance and Plan-space Refinements 21

4.4 Hierarchical (HTN) refinement 26

4.5 Tractability Refinements 27

4.6 Interleaving different Refinements 31

5. TRADEOFFS IN REFINEMENT PLANNING 32

5.1 Asymptotic Tradeoffs 32

5.2 Empirical Study of Tradeoffs in Refinement Planning 34

5.3 Selecting among refinement planners using subgoal interaction analysis 35

6. SCALING-UP REFINEMENT PLANNERS 37

6.1 Scale-up through customization 37

6.2 Scale-up through Disjunctive representations and constraint satisfaction techniques 38
6.2.1 Disjunctive Representations 38
6.2.2 Refining Disjunctive Plans 39
6.2.3 Open issues in planning with disjunctive representations 41

7. CONCLUSION AND FUTURE DIRECTIONS 48

List of Figures

Figure 1. Role of planning in intelligent agency 3
Figure 2. Classical planning problem 3
Figure 3. Rocket domain 4
Figure 4. Actions in rocket domain (upper case letters denote constants, while lower cse ones stand for variables) 5
Figure 5. Progression and Regression of world states through actions. 5
Figure 6. A chronology of ideas in classical planning 7

Figure 7. Overview of refinement planning. On the right is the semantic (candidate set) view and on the left is the syntactic (plan set) view. ____ *8*
Figure 8. An example (partial) plan in rocket domain ____ *9*
Figure 9. Candidate set of a plan ____ *10*
Figure 10. Relating the syntax and semntics of a partial plan. ____ *11*
Figure 11. An example refinement strategy (Forward State Space) ____ *12*
Figure 12. Complete and progressive refinements narrow the candidate set without losing solutions ____ *13*
Figure 13. Basic refinement planning template. ____ *13*
Figure 14. Refinement planning with search. ____ *14*
Figure 15. Introducing search into refinement planning by splitting and handling individual components of plan sets separately. ____ *15*
Figure 16. Nomenclature for partial plans. ____ *16*
Figure 17. Forward State Space Refinement. ____ *17*
Figure 18. Example of Forward State Space Refinement ____ *17*
Figure 19. Using means-ends analysis to focus forward state space refinement. ____ *19*
Figure 20. Backward State Space Refinement. ____ *20*
Figure 21. Backward state-space refinement. ____ *20*
Figure 22. 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. ____ *21*
Figure 23. Steps in plan-space refinement ____ *23*
Figure 24. Example of Plan space refinement. ____ *24*
Figure 25. Plan space refinement example showing both establishment and de-clobbering steps. ____ *26*
Figure 26. Using non-primitive tasks, which are defined in terms of reductions to primitive tasks. ____ *26*
Figure 27. Pre-ordering refinements ____ *27*
Figure 28. Pre-positioning refinements. ____ *28*
Figure 29. Pre-satisfaction refinements ____ *29*
Figure 30. Pre-reduction refinements. ____ *30*
Figure 31. Interleaving refinements ____ *32*
Figure 32. Asymptotic tradeoffs in refinement planning. ____ *33*
Figure 33. Different partial plans for solving the same subgoal ____ *36*
Figure 34. Disjunction over state-space refinements. ____ *39*
Figure 35. Disjunction over plan-space refinements ____ *39*
Figure 36. Refinement planning with controlled splitting. ____ *42*
Figure 37. Understanding Graphplan algorithm as a refinement planner using state space refinements over disjunctive partial plans. ____ *44*
Figure 38. Relating refined plan at k-th level to SATPLAN encodings. ____ *46*
Figure 39. Relating encodings based on minimal candidates of refined plans to direct encodings generated by SATPLAN. ____ *47*

Abstract:

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 Artificial Intelligence for over 30 years. Work on efficient planning algorithms still continues to be a very hot topic of research in AI, and has lead to several exciting developments in the past several years. This article aims to provide a tutorial introduction to all the algorithms and approaches to the planning problem in Artificial Intelligence. In order to fulfill this ambitious objective, I will 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 will provide the reader with a brand-name free appreciation of essential issues in planning.

1. Introduction

Planning is the problem of synthesizing a course of action that when executed, will take an agent from a given initial state to a desired goal state. Automating plan synthesis has been an important goal of the research in Artificial Intelligence for over 30 years. A large variety of algorithms, with differing empirical tradeoffs, 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 aim to remedy this situation by providing a unified overview of the approaches for plan synthesis. I will do this by describing a general and powerful plan synthesis paradigm called “refinement planning,” and showing 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 will separate the essential tradeoffs from the peripheral ones (e.g. brand-name affiliations), and provide the reader a firm understanding of the existing work as well as a feel for the important open research questions.

Here is an overview of the article. I will briefly discuss the (classical) planning problem in Artificial Intelligence, and provide the chronology of the many existing approaches. I will then propose refinement planning as a way of unifying all these approaches, and present the formal framework for refinement planning. Next, I will describe the refinement strategies that correspond to existing planners. I will then discuss the tradeoffs among

different refinement planning algorithms. Finally, I will describe some promising new directions for scaling up refinement planning algorithms.

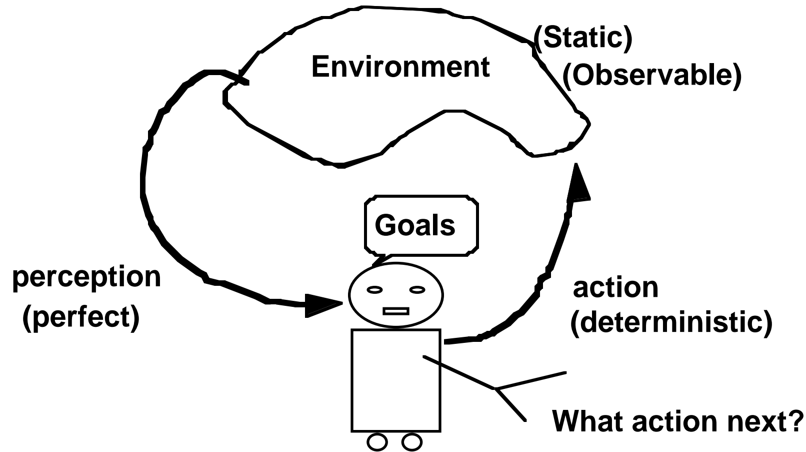

Figure 1. Role of planning in intelligent agency

2. 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 upon based on its overall goals, information about the current state of the environment, and the dynamics of the evolution of its environment (see Figure 1).

The complexity of plan synthesis depends on a variety of properties of the environment and the agent, inclduing (i) whether the environment evolves only in response to the agent's actions or also independently (ii) whether the state of the environment is observable or partially hidden (iii) whether the sensors of the agent are powerful enough to perceive the state of the environment and finally (iv) whether the agent's actions have deterministic or stochastic effects on the state of the environments. Perhaps the simplest case of planning occurs when the environment is static, (*in that it changes only in response to the agent's actions*), 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**.

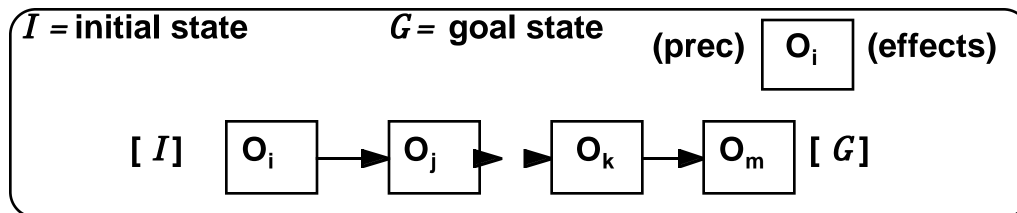

Figure 2. Classical planning problem

The classical planning problem is thus specified (see 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 limitations, the classical planning problem is still very important in understanding the structure of intellgent agency. Work on classical planning has historically also helped our understanding of planning under non-classical assumptions. The problem itself is computationally hard -- P-Space hard or worse [Erol et. al., 1995], and a significant amount of research has gone into efficient search-based formulations.

2.1 Modeling Actions and states

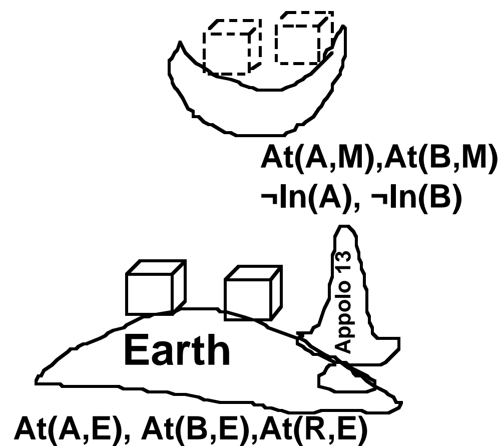

Figure 3. Rocket domain

We shall now look at the way the classical planning problem is modeled. Let us use a very 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 completely specified, so negated conditions (i.e., state-variables with false values) need not be seen. 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 creatively named Action Description Language [Pednault, 1988], where preconditions and effects are first order quantified formulas (with no disjunction in the effects formula, since the actions are deterministic).

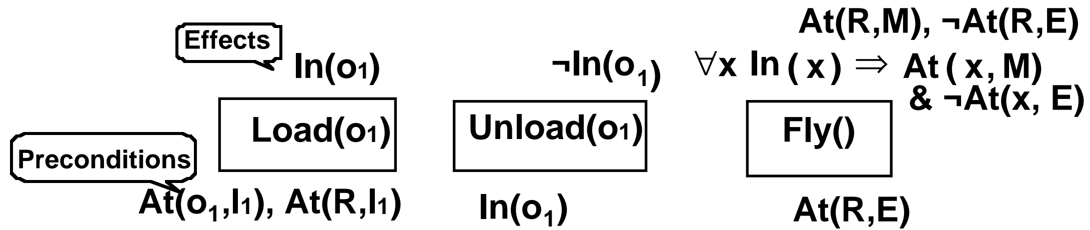

Figure 4. Actions in rocket domain (upper case letters denote constants, while lower cse ones stand for variables)

We have three actions in our rocket domain (see Figure 4)— **Load** which causes a package to be *in* the rocket, **Unload**, which gets it out, and **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 may be worth noting that the implication in the second effect of the Fly action is not a strict logical implication, but rather a shorthand notation for saying that when In(x) holds for any x in the state in hich Fly action is executed, At(x,M) and ¬At(x,E) will be true in the state resulting after the execution.

2.2 Action Application

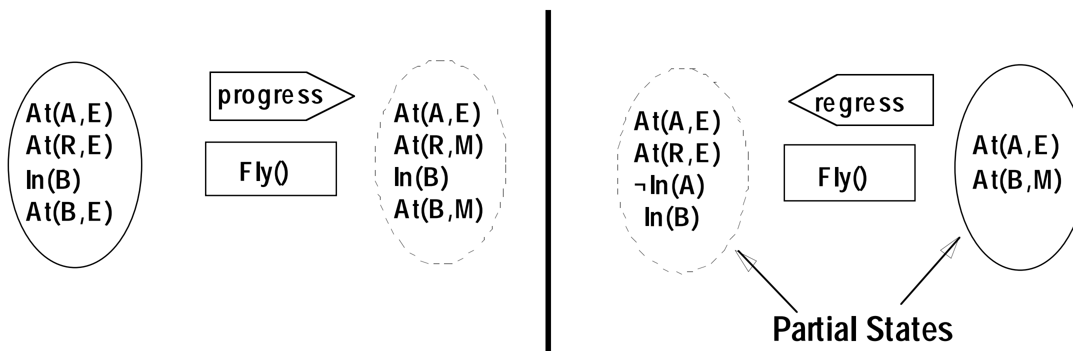

Figure 5. Progression and Regression of world states through actions.

As mentioned earlier, actions are seen as state transformation functions. In particular, an action can be executed in any state where its preconditions hold and upon execution the state is modified such that state-variables named in the effects have the specified values, while the rest 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 5 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** (if this happens, the state will be inconsistent, 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 5 illustrates the process of regressing the final (goal) state of the rocket problem through the action Fly().

2.3 Verifying a solution

Having described how actions transform states, we can 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 (i.e., whether the state variables named in the goal specification of the problem occur in the that 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, and regressing the resulting state over the second to 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).

2.4 Chronology of classical planning approaches

Plan generation under classical assumptions had received wide-spread attention and a large variety of planning algorithms have been developed. Initial approaches to the planning problem have attempted to cast planning as a theorem proving activity [Green, 1970]. The inefficiency of first-order theorem-provers in existence at that time, coupled with the difficulty of handling the “frame problem”[2] in first order logic, have lead to search-based

[2] The frame problem, in the context of planning, refers to the idea that a first-order logic based description of actions must not only state what conditions are changed by an ation, but also what conditions remain unchanged after the action. Since in any sufficiently rich domain, there are very many conditions that are left unchanged by an action, this causes two separate problems. First, we may 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 infact remained the same . Although the first problem can be alleviated by using “domain specific frame axioms” [Haas, 1987] which state, for each condition, the circumstances under which it changes, the second problem

approaches in which the "STRIPS assumption" --viz., 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 the STRIPS planner [Fikes et. al., 1972], which searched in the space of world states using means-ends-analysis (see Section 4.2). 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 [Tate, 1975; Chapman, 1987; McAllester and Rosenblitt, 1991; Penberthy and Weld, 1992]. A closely related formulation called "hierarchical planning" [Sacerdoti, 1972; Tate, 1977; Wilkins, 1984] allowed a partial plan to contain "abstract" actions which can be incrementally 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 [Blum and Furst, 1995; Joslin and Pollack, 1996; Kautz and Selman, 1996].

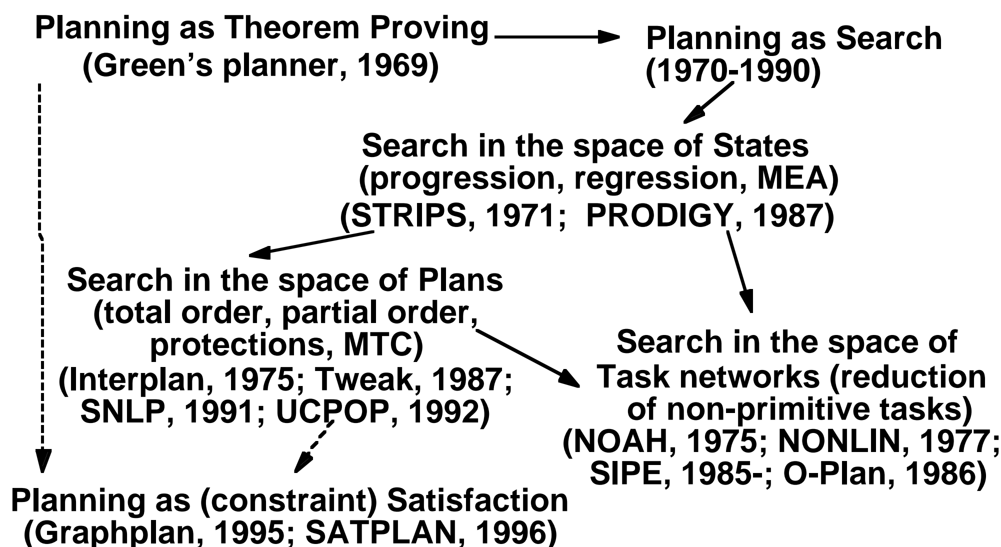

Figure 6. A chronology of ideas in classical planning

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

3. Refinement Planning: Overview

Since 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. This is the essential idea behind refinement

---

cannot be so easily resolved. Any general purpose theorem prover would have to use the frame axioms explicitly to prove persistence of conditions.

planning, which is the process of starting with the set of all action sequences and narrowing it down gradually to reach the set of all solutions. The sets of action sequences are represented and manipulated in terms of **partial plans** which can be seen as a collection of constraints**.** The action sequences denoted by a partial plan, i.e., those that are consistent with its constraints, are called its **candidates**. For technical reasons that will become clear later, we find it convenient to think in terms of sets of partial plans (instead of single partial plans). A set of partial plans is called a **planset** with its constituent partial plans referred to as the **components.** The candidate set of a planset is efined as the union of the candidate sets of its components.

Figure 7 provides a schematic illustration of the ideas underlying refinement planning. On the left is a venn diagram relation between the set of action sequences under consideration in refinement planning. The set of all action seuqences is narrowed down to a subset *P* of action sequences, which in turn is narrowed down to another subset *P'* of action sequences. Notice that all these sets are supersets of the set of all solutions for the problem. The manipulation of plansets corresponding to this narrowing process is shown on the right of Figure 7. The null plan corresponds to the set of all action sequences. A refinement operation attempts to narrow a plan's candidate set without eliminating any solutions from it. This is done by augmenting the constraints comprising the partial plan. For example, in the figure the null plan is refined into the plan set *P* which is further refined into a plan set *P'* and so on. This process can be seen as progressing towards a planset all of whose candidates are solutions.

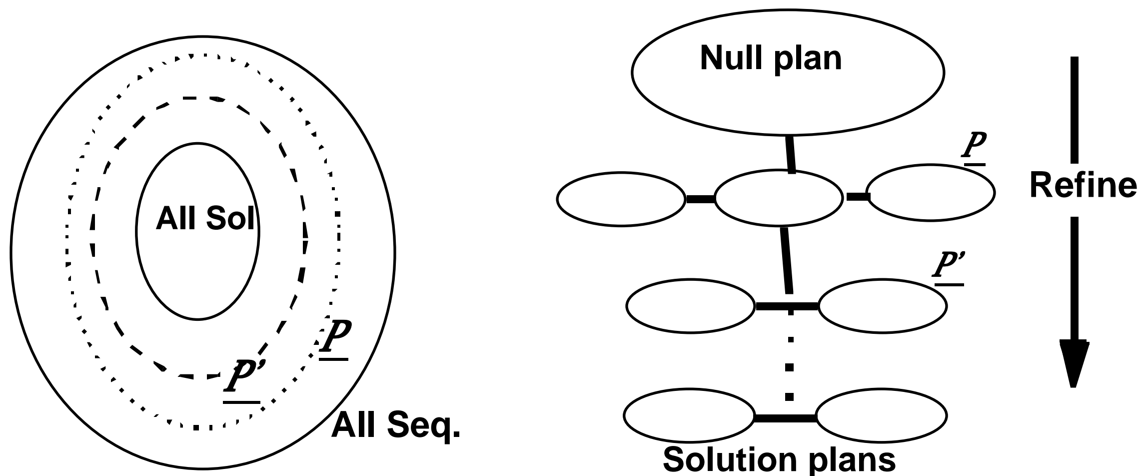

Figure 7. Overview of refinement planning. On the right is the semantic (candidate set) view and on the left is the syntactic (plan set) view.

A refinement operation narrows the candidate set of a planset by adding constraints to its component plans. If no solutions are eliminated in this process, we will eventually

progress towards set of all solutions. Termination can occur as soon as we can pick up a solution using some bounded time operation -- called the *solution extraction function.*

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

3.1 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[3] that is sufficient for our purposes models partial plans as a set of steps, ordering constraints between the steps, and auxiliary constraints.[4]

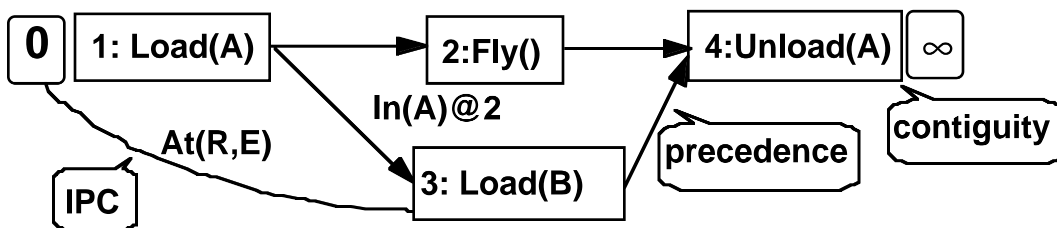

Figure 8. An example (partial) plan in rocket domain. IPC stands for interval preservation constraint.

Each plan step is identified with a unique step number, and corresponds to an action (this allows 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 constraints between a pair of steps—viz., **precedence** and **contiguity**. A precedence constraint requires one step to precede the second step (without precluding other steps from coming between the two), while a contiguity 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 will be interested in two types of auxiliary constraints—*interval preservation constraints* which require *non-violation* of a condition over an interval (no action having an effect ¬p will be allowed in an interval where the condition p is to be

[3] It is perhaps worth repeating that this representation for partial plans is sufficient but not necessary. We 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 [Ginsberg, 1996].

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

preserved), and *point truth constraints* that 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 otherwords, a topological sort). A **safe linearization** is a linearization of the plan that is also consistent with the auxiliary constraints.

Figure 8 shows an example plan from our rocket domain in this representation. 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), 2 precedes 4, and the condition At(R,E) must be preserved between 0 and 3 (illustrated in the figure 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 while the first one is not (since step 2 will violate the interval preservation constraint on At(R,E) between 0 and 3).

3.2 Partial Plan Representation: Semantics

The semantics of the partial plans are given in terms of candidate sets. 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 example plan shown in

Figure 8, the action sequences shown on the left in Figure 9 are candidates, while those on the right are non-candidates.

Candidates (∈ «P»)	Non-Candidates (∉ «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 9. Candidate set of a plan

Notice that the candidates may contain more actions than are present in the partial plan. Because of this, a plan's candidate set can be potentially 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 while 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 9 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 9 are non-candidates 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 non-candidate can be seen as starting with the minimal candidate [Load(A),Load(B),Fly(),Unload(A)] and adding another instance of the action Fly() which does not respect the IPC on At(R,E).

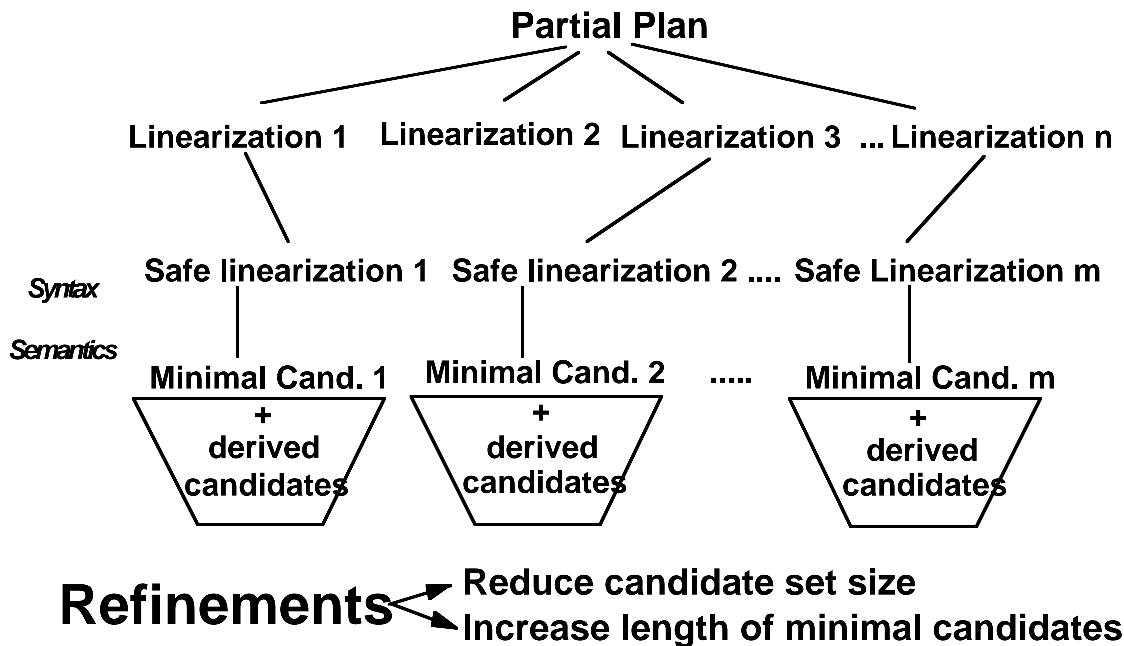

Figure 10. Relating the syntax and semntics of a partial plan.

3.3 Connecting Syntax and Semantics of Partial Plans

Figure 10 summarizes the connection between the syntax and semantics of a partial plan. Each partial plan has at most 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 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 linearlizations 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 shall see in the following, that refinement strategies add new constraints to a partial plan. They thus simultaneously shrink the candidate set of the plan, and increase the length of its minimal candidates. This 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 checking to see if the final state corresponds to a goal state (see Section 2.3).

3.4 Refinement Strategies

We will now formally define a refinement strategy. Refinement strategies are best seen as operating on plansets. A refinement strategy ***R*** maps a planset **P** to another planset **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 **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 planset and the size of the original planset.

Completeness ensures that we don't lose solutions by the application of refinements. Progressiveness ensures that refinement narrows the candidate set. Systematicity ensures that we never consider the same candidate more than once, if we were to explore the components of the planset separately (see Section 3.6) .

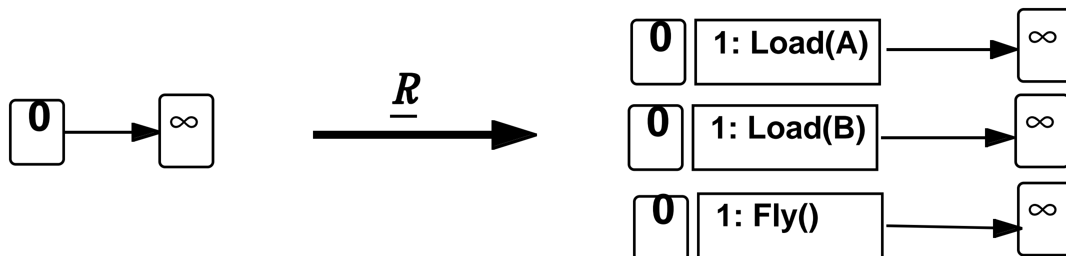

Figure 11. An example refinement strategy (Forward State Space)

Let us illustrate these notions with an example. Figure 11 shows an example refinement, for our rocket problem. It takes the null planset, corresponding to all action sequences and maps it to a plan set containing 3 components. In this case, the refinement is complete

since no solution to the rocket problem can start with any other action for the given initial state, progressive since it eliminated action sequences not beginning with Load(A), Load(B) or Fly() from consideration, and systematic since 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).

Figure 12 illustrates the process of refinement of a planset **P** using a complete and progressive refinement strategy, from a candidate-set perspective. The planset **P** is refined into the planset **P'**, where **P'** is a proper subset of **P** (thus some candidates have been eliminated from consideration). However, both **P** and **P'** have the same intersection with the set of all solutions.

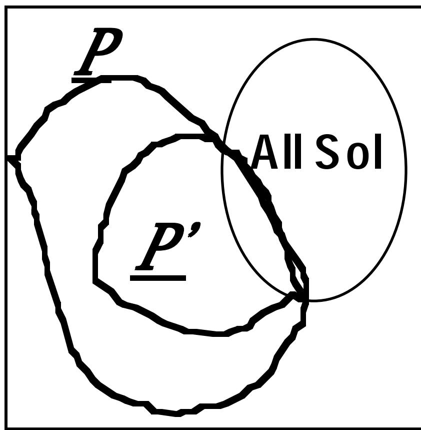

Figure 12. Complete and progressive refinements narrow the candidate set without losing solutions

3.5 Planning using Refinement Strategies and solution extraction

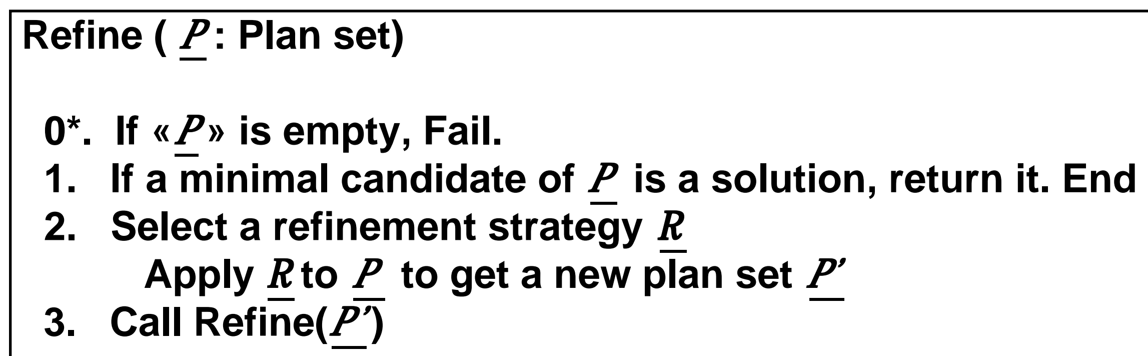

Figure 13. Basic refinement planning template.

We are now in a position to present the general refinement planning template, which we do in Figure 13. It has three main steps. If the current planset 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 planset.

As long as the selected refinement strategy is complete, we will never lose a solution. As long as the refinements are progressive, for solvable problems, we will eventually reach a planset one of whose minimal candidates will be 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 have their preconditions satisfied in the state preceding the action.

As we shall see in Section 6.2, some recent planners like 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 we discuss next.

3.6 Introducing Search into Refinement Planning

The algorithm template in Figure 13 does not have any search in the foreground. Specifically, assuming that the cost of applying a refinement strategy to a plan set is polynomial (as is the case for the existing refinement strategies; see Section 4), the bulk of the computation is pushed into the solution extraction function, which has to sort through all minimal candidates of the planset, looking for solutions. It is possible to add “search” to the refinement process in a straightforward way -- to separate the individual components of a planset, and handle them in different search branches. The primary motivation for doing this is to reduce 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, coupled with a depth-first search may make the process of plan generation easier for humans to understand.

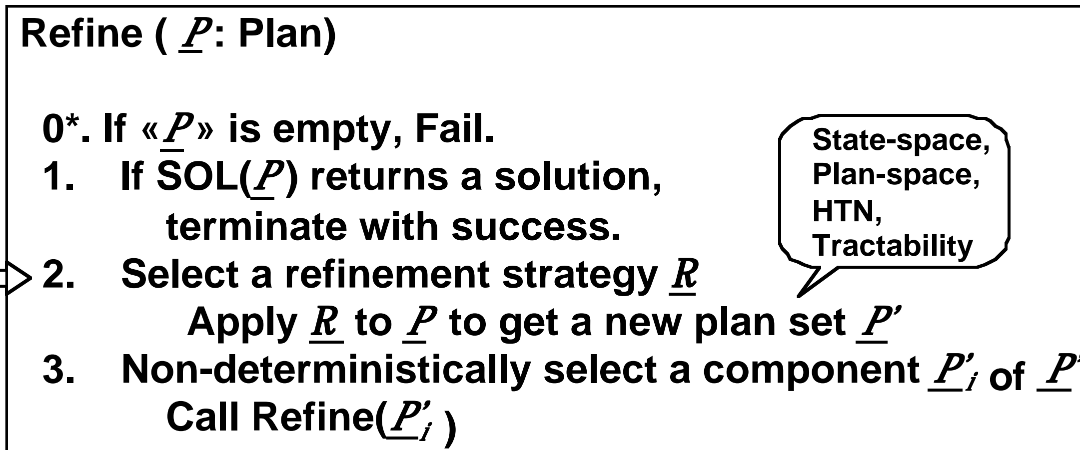

Figure 14. Refinement planning with search.

Figure 15 illustrates introduction of search into refinement planning graphically. On the left is a sequences of plan sets generated by refining the null plan. On the right is an equivalent refinement tree where each component of the plan set is kept in a separate search branch and is refined separately. Notice that this search process will have backtracking even for complete refinements. Specifically even if we know that the candidate set of a planset **P** contains all the solutions to the problem, we do not know how they are distributed among the components of **P.** We shall see later (Section 5) that the likelihood of the backtracking depends upon 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 algorithm template shown in Figure 14 introduces search into refinement planning. 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). We thus 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 shall see in the next section).

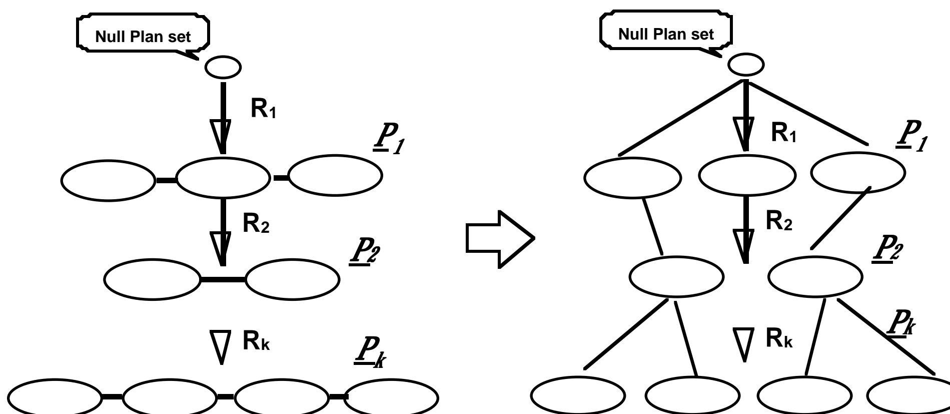

Figure 15. Introducing search into refinement planning by splitting and handling individual components of plan sets separately.

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 fall broadly into four types—state-space, plan space, task reduction and tractability refinements. We will now look at these four refinement families in turn.

4. Existing Refinement Strategies

In this section, we shall 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 16 shows an example plan with its important structual aspects marked. The prefix of the plan, i.e., 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**." Since 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**." Tail state can be seen as providing the weakest conditions under which the actions in the tail of the plan can be executed to result in a 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.

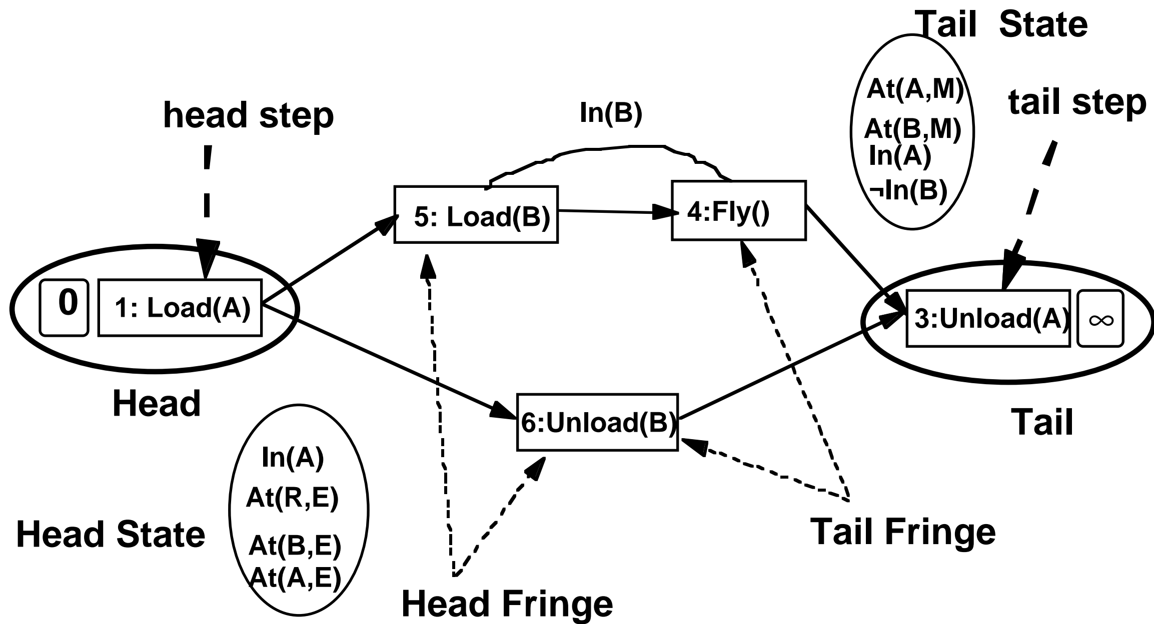

Figure 16. Nomenclature for partial plans.

4.1 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 head-state

2. Operator Application: Add a contiguity constraint between the current head step, and the new step t. (This makes t the new head step, and updates head-state)

Figure 17. Forward State Space Refinement.

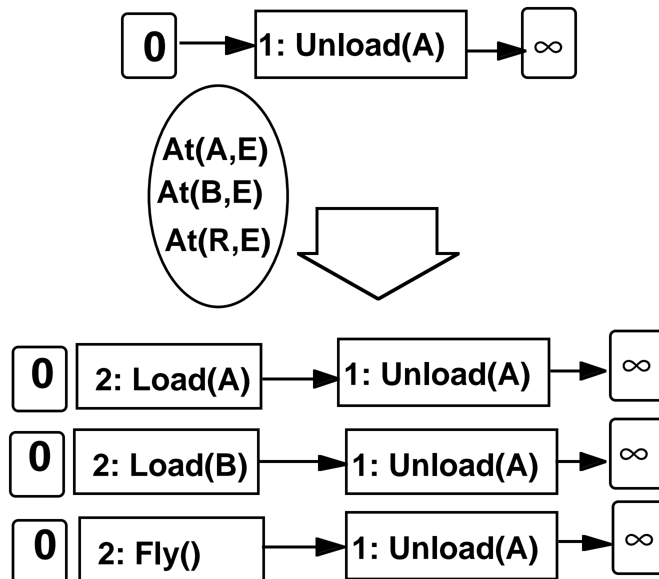

Figure 18. Example of Forward State Space Refinement

Figure 18 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 since it eliminates all action sequences with non-executable prefixes. It is also complete since any solution must have an executable prefix. It is systematic, since each of its components differ 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. Clearly, this ensures that the only minimal candidate of the plan corresponds to a solution.

4.1.1 Generalizing Forward State Space Refinement to allow parallel actions

The version of the refinement we considered above extends the head by one action at a time. This could lead to larger than required number of components in the resulting planset. To see this, consider the actions Load(A) and Load(B), each of which are applicable in the initial state, and both of which can be done simultaneously as 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 the actions to the plan prefix, thereby reducing the number of components of resulting planset. This 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 planset in Figure 18 into a single plan component with both Load(A) and Load(B) being made contiguous to step 0. More broadly, the generalized forward state space refinement strategy should consider maximal sets of non-interacting actions that are all applicable in the current initial state together [Drummond, 1989]. Here we consider two actions as interacting if the preconditions of one are deleted by the effects of the other.

4.2 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 may 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 -- 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 using backward state space refinement which operates by growing the tail of the partial plan. We elaborate these ideas in the next two subsections.

4.2.1 Means-ends analysis

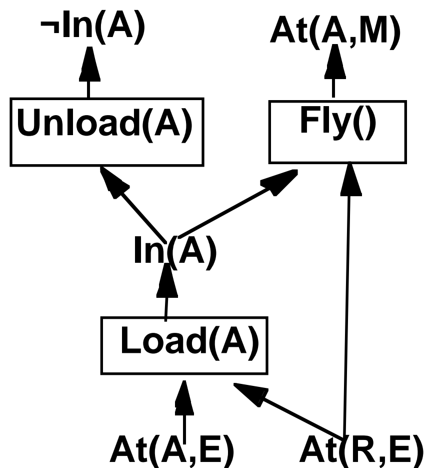

Figure 19. Using means-ends analysis to focus forward state space refinement.

First, we can force forward state space refinement to consider only those actions which 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 19. Here the top level goal can be potentially 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). Since 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, since it does not directly or indirectly support any top level goals.

This way of identifying relevant actions is known as **means-ends analysis** and has been used by one of the first planners called 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 19, having decided that Fly() and Unload(A) actions are relevant for the top level goals, STRIPS will introduce "Fly()" action into the plan head, before continuing to consider actions such as Load(A) that are recursively relevant. This can sometimes lead to premature operator application that would have to be backtracked over. In fact many of the famous "incompleteness" results related to STRIPS planner [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 (a) deferring operator application until all relevant actions are computed and (b) repeating the computation of relevant actions afresh after each refinement.

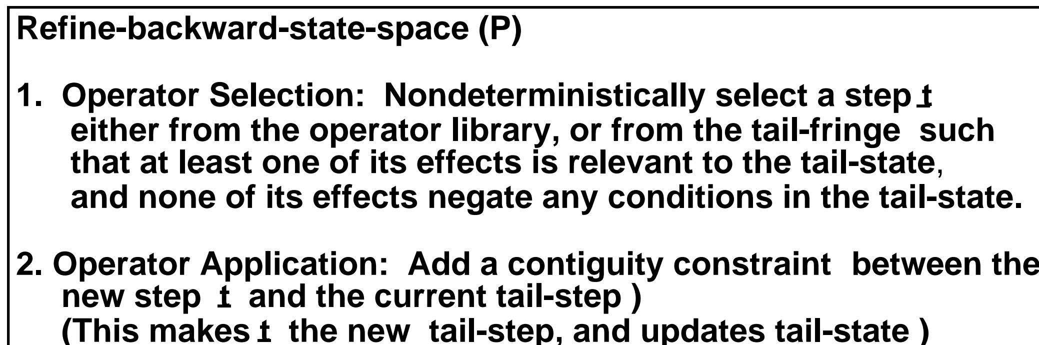

Figure 20. Backward State Space Refinement.

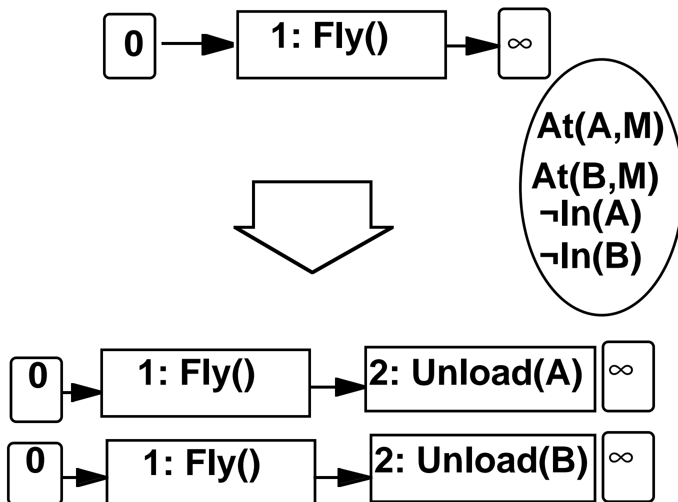

Figure 21. Backward state-space refinement.

4.2.2 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. Figure 21 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 since it can violate the ¬In(x) condition in the tail state.

Compared to the forward state space refinement, the backward state space refinement generates plansets with fewer number of components, as it concentrates only on those actions that are relevant to current goals. This in turn leads to a lower branching factor for planners that consider the planset components in different search branches. On the other hand, since initial state of a planning problem is completely specified and goal state is only partially specified, the head state computed by the forward state space refinement is a complete state while 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 require the ability to evaluate the truth of complex formulas about the state of the plan, and view this as an advantage in favor of forward state space refinements, since truth evaluation can be done in terms of model checking rather than theorem proving.

4.3 Position, Relevance and Plan-space Refinements

The state space refinements have to decide both the "position" and "relevance" of a new action to the overall goals. Often times, we may 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. In such cases, it helps to introduce an action into the plan, without constraining its absolute position. This is the main motivation behind plan-space refinement. 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.

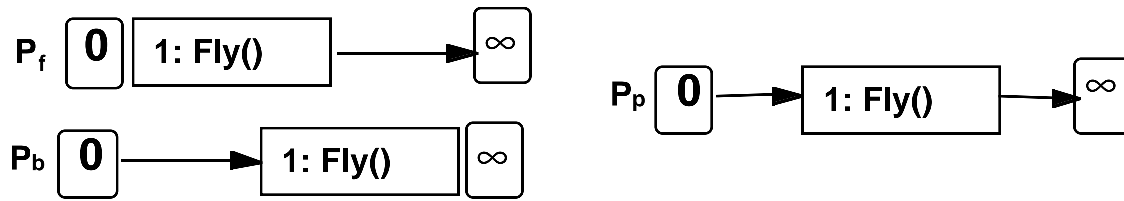

Figure 22. State-space refinements attempt to guess both the position and the relevance of an action to a given planning problem. Plan space refinements can consider relevance without committing to position.

"Modal Truth Criterion" and Plan-space refinement

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 (i.e., 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 involved 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 will then consider adding all possible combinations of additional constraints (steps, orderings) to the plan to make them true. Since interpreting the truth criterion turns out to be NP-hard when the actions in the plan can have conditional effects, this has lead to the belief that the cost of an individual plan-space refinement can be exponential.

The fallacy in this line of reasoning becomes apparent once 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) since 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 requring 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].

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 22 contain the single fly action, the solution sequence

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

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

Since 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. Since all refinements are complete, backtracking would never be required had we worked with plansets without splitting.)

This brings us to the specifics of plan-space refinement. As summarized in Figure 23, 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 (de-clobbering). An optional bookkeeping step (also called protection step) imposes interval preservation constraints to ensure that the established precondition is “protected” during future refinements. PSR can have several instantiations depending on whether or not bookkeeping strategies are used, and how the preconditions are selected for establishment in step 1.

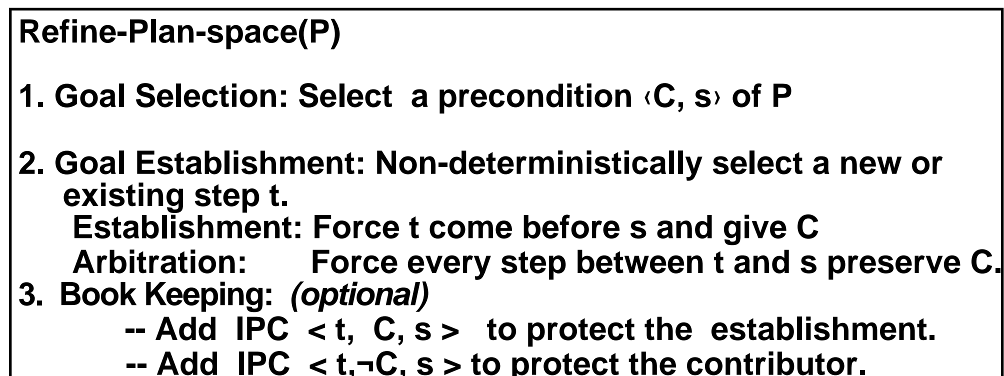

Figure 23. Steps in plan-space refinement

Figure 24 shows an example of plan-space refinement. In this example, we pick the precondition At(A,M) of the last step (which stands for the top level goal). We add the new step Fly() to support this condition. In order 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 book-keeping

step involves adding interval preservation constraints to preserve this establishment during subsequent refinement oprations (when new actions may come between Fly and the last step). This is done by adding either one or both of the interval preservation constraints <2, At(A,M), ∞> and 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.

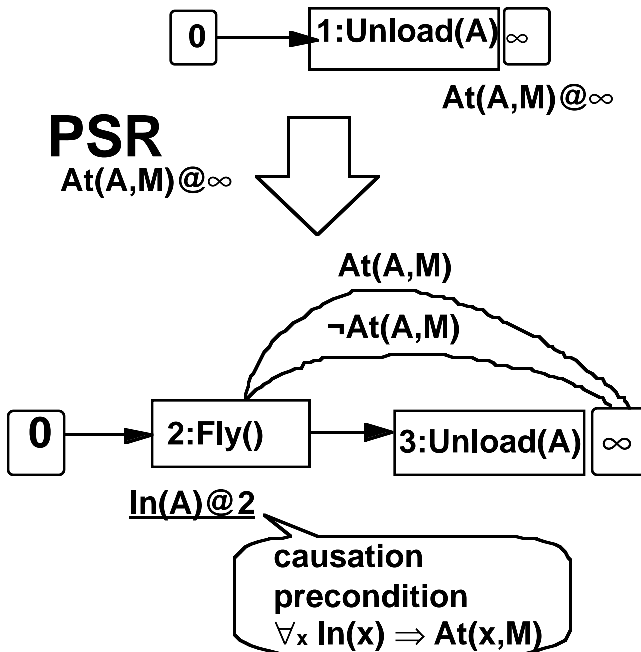

Figure 24. Example of Plan space refinement.

Partial-order, Total-order and Contiguous plans

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

A planner employing plan-space refinement, and thus using precedence relations, can produce totally ordered partial plans if it uses pre-ordering based tractability refinements (see Section 4.5). Examples of such planners include TOCL [Barrett and Weld, 1994] and TO [Minton et al., 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 non-interfering actions together in one planset component rather than in separate ones (see Section 4.1.1).

Figure 25 shows another example where we need to do both establishment and de-clobbering 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 since 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 and ∞. 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 ¬In(A) to step 1 (since if A is not in the rocket, then A's position will not change when rocket is flown). These three ways of shoring up the establishment are called "promotion", "demotion" and "confrontation" respectively.

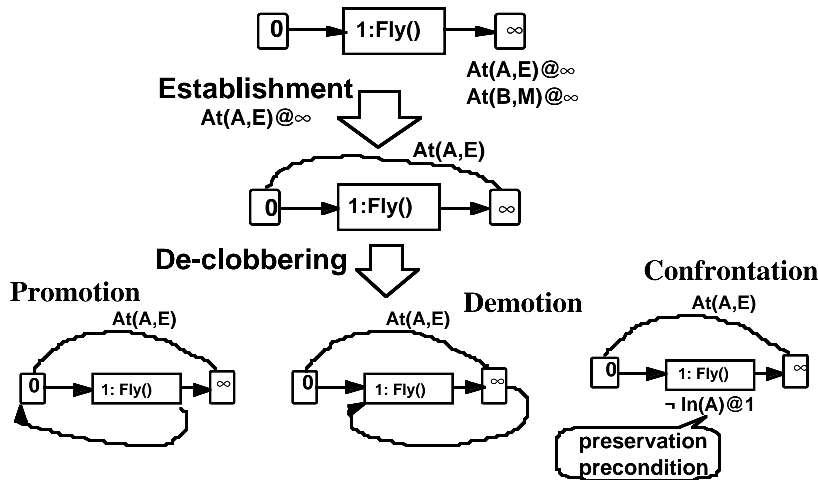

Figure 25. Plan space refinement example showing both establishment and de-clobbering steps.

4.4 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 may have significant preferences among the solutions. For example, when I use my planner to make travel plans to go from Phoenix to Portland, I may not want a plan that involves bus rides. The question is how do we communicate these biases to the planner such that it will not waste any time progressing towards unwanted solutions? While removing bus-ride action from the planner's library of actions is a possible solution, it may be too drastic. I may want to allow bus travel for shorter itineraries, for example.

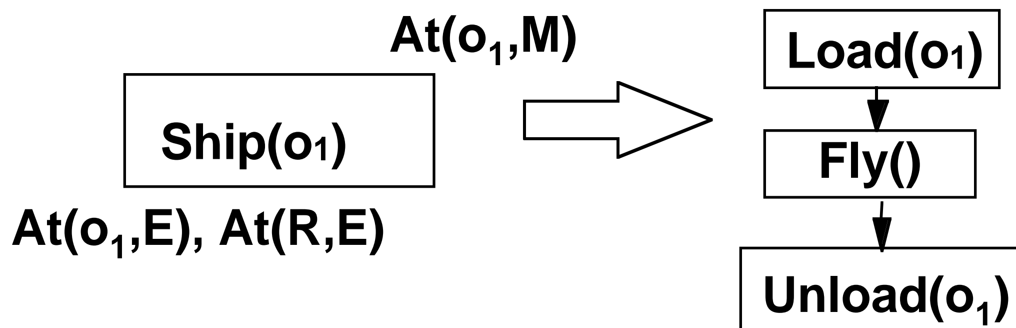

Figure 26. Using non-primitive tasks, which are defined in terms of reductions to primitive tasks.

One natural way turns out to be to introduce non-primitive actions, and restrict their reduction to primitive actions through user-supplied reduction schemas. Consider the example in Figure 26. Here the non-primitive action Ship(o_1) has a reduction schema that translates it to a plan-fragment containing three actions. Typically, there may be multiple possible legal reductions for a non-primitive action. The reduction schemas restrict the

planner's access to the primitive actions and thus stop progress towards undesirable solutions [Kambhampati, 1995].

For this method to work, we do need the domain-writer to provide us reduction schemas over and above domain dynamics. This can be a steep requirement since 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 be easily elicited. Of course, acquiring that knowledge and verifying its correctness can still be non-trivial [Chien et. al., 1996].

4.5 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. The motivation for their use is to reduce the plan handling costs further -- we thus call them tractability refinements. Many of them can be understood as splitting any implicit disjunction among the plan constraints into the search space.

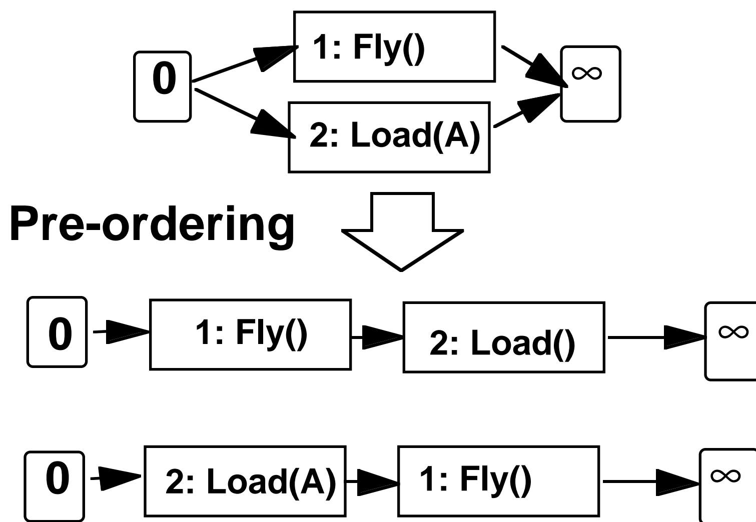

Figure 27. Pre-ordering refinements

We can classify the tractability refinements into three categories. The first attempt to reduce the number of linearizations of the plan. In this category, we have **pre-ordering refinements** which order two unordered steps, and **pre-positioning refinements** which constrain the relative position of two steps.

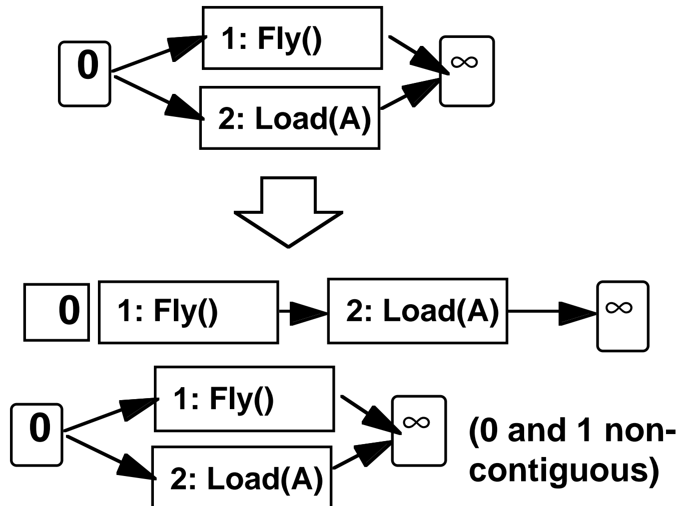

Figure 28. Pre-positioning refinements.

Pre-ordering refinements are illustrated in Figure 27—the single partially ordered plan at the top of the figure is converted into two totally ordered plans below. Planners employing pre-ordering refinements include TOCL [Barrett and Weld, 1994] and TO [Minton et. al, 1994].

Pre-positioning refinement is illustrated in Figure 28, where one refinement considers the possibility of step 1 being contiguous to step 0 while the other considers the possibility of step 1 being non-contiguous to step 0. Planners such as STRIPS and PRODIGY can be understood as using pre-positioning refinements (to transfer the steps from the means-ends analysis tree to the plan head; see Section 4.2.1).

The second category of tractability refinements attempts to make all linearizations safe with respect to auxiliary constraints. Here we have **pre-satisfaction** refinements, which split a plan in such a way that a given auxiliary constraint is satisfied by all linearizations of the resulting components. Figure 29 illustrates a pre-satisfaction 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 attempt to keep the step Fly() from intervening between 0 and ∞. The last one ensures that Fly() will be forced to preserve the condition At(A,E). Readers may note a strong similarity between the pre-satisfaction refinements and the de-clobbering phase of plan-space refinement (see Section 4.3). The important difference is that de-clobbering is done with respect to the condition that is established in the current refinement, to ensure that the established condition is not deleted by any step that is currently present in the plan. There is no gurantee that steps that will be introduced by future refinements will continue to respect this establishment. In contrast, pre-satisfaction 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 may 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 pre-satisfaction refinements.

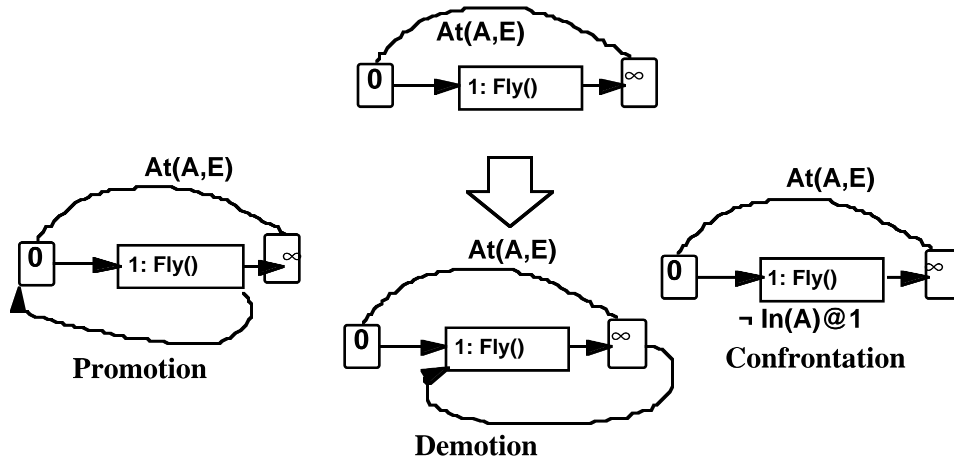

Figure 29. Pre-satisfaction refinements

The third category of tractability refinements attempt to reduce uncertainty in the action identity. An example is **pre-reduction** refinement, which converts a plan containing a non-primitive action into a set of plans each containing a different reduction of that non-primitive action. Figure 30 illustrates the pre-reduction refinements. The plan at the top contains a non-primitive 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, pre-reduction refinements convert this plan to a plan set each of whose components correspond to a different ways of reducing Ship(A) action (in the context of Figure 30, it is assumed that only one way of reducing Ship(A), viz., that shown in Figure 26, is available).

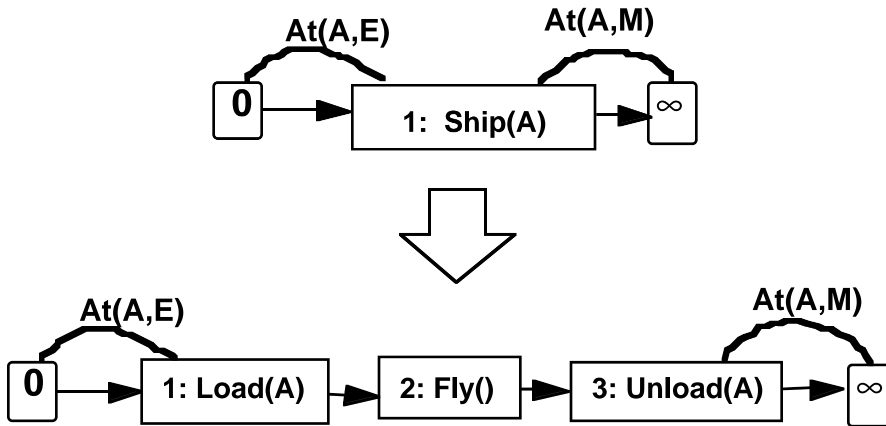

Figure 30. Pre-reduction refinements.

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 by Table 1 which characterizes several plan-space planners in terms of the specifics of the plan-space refinements they employ (protection strategies, goal selection strategies), and the type of tractability refinements they use.

Table 1. A spectrum of plan-space planners

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	pre-satisfaction
TOCL [Barrett and Weld, 1994]	Arbitrary	IPCs to protect the established condition as well as its negation	pre-ordering
UA, TO [Minton et. al., 1994]	Based on Modal Truth Criterion	none	pre-ordering (UA orders only interacting steps while TO orders all

			pairs of steps)

4.6 Interleaving different Refinements

One of the advantages of the treatment of refinement planning that I have presented is that it naturally allows for interleaving of a variety of refinement strategies in solving a single problem. From a semantic view point, since different refinement strategies correspond to different ways of splitting the candidate sets, it is perfectly legal to interleave them. We can formally guarantee completeness of planning if each of the individual refinement strategies are complete. Figure 31 shows an example of solving 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 pre-position 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, 1995, 1996] show that interleaving refinements this way can sometimes lead to superior peformance over single-refinement planners.

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 candidate set each refinement affords, with some success.

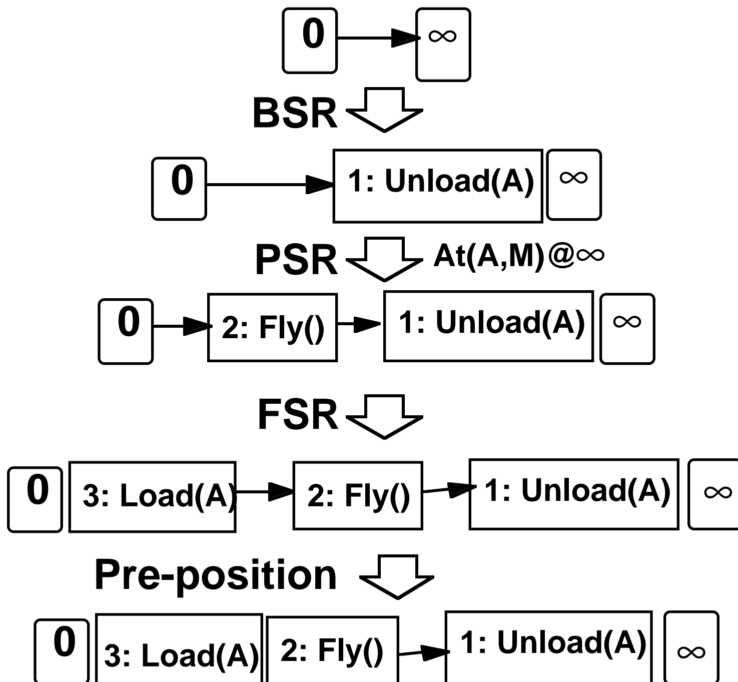

Figure 31. Interleaving refinements

5. Tradeoffs in refinement planning

We have described a parameterized refinement planning template that allows for a variety of specific algorithms depending on which refinement strategies are selected and how they are instantiated. We shall now attempt to understand the tradeoffs governing some of these choices, and see how one can go about choosing a planner, given a specific population of problems to solve. We concentrate here on the tradeoffs in refinement planners that split planset components into search space (see Section 3.6).

5.1 Asymptotic Tradeoffs

Let us start with an understanding of the asymptotic trade-offs in refinement planning. To do this, we shall use an estimate of the search space size in terms of properties of the plans at the fringe of the search tree (see Figure 32). Suppose **K** is the total number of action sequences (to make this 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** the average number of candidates in each of the plans on the fringe. Let ρ 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 candidate set narrows each time a refinement is done. We then have

$$\text{F} = \frac{p^d \times K \times \rho}{k}.$$

Since **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 effective depth **d** of the generated search tree. Specifically,

$$F = \frac{p^d \times K \times \rho}{k} = b^d.$$

The time complexity of search can be written out as **C**x**F**, where **C** is the average cost of handling plansets. **C** itself can be broken down into two components, $\mathbf{C_R}$, the cost of applying refinements and and $\mathbf{C_S}$, the cost of extracting solutions from the plan. Thus,

$$T = (\mathbf{C_S} + \mathbf{C_R})\ F$$

These formulas can be used to understand the asymptotic tradeoffs in refinement planning, as shown in Figure 32.

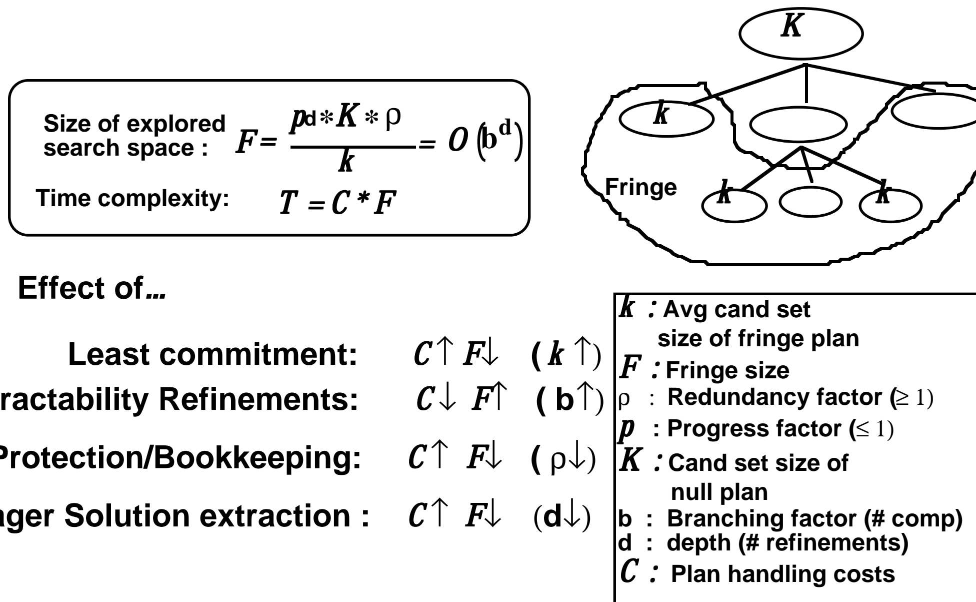

Figure 32. Asymptotic tradeoffs in refinement planning.

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 may reduce **C** by reducing $\mathbf{C_S}$ (since the tractability refinements reduce the variation amongst the linearizations of the plan, thereby facilitating cheaper solution extractors). The protection (bookkeeping) strategies reduce the redundancy factor ρ, and thus reduce **F**. But, they may increase **C** since protection is done by adding additional constraints, whose consistency needs to be verified.

While instructive, the analysis of this section does not make conclusive predictions on practical performance since the latter depends on the relative magnitudes of changes in **F** and **C**. To quantify this, we look at empirical evaluation.

How important is "least commitment"?

One of the more hotly debated issues about 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 over-committing may eventually make the partial plan inconsistent, necessitating backtracking. As an illustration, in Figure 22 we saw that individual components of state-space refinements tend to commit regarding both the absolute position and relevance of actions inserted into the plan, while plan-space refinements commit to the relevance, but leave 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 "least commitment" planning is used synonymously with partial order or plan-space planning). For example, a plan space refinement that does (the optional) book-keeping by imposing interval preservation constraints is "more constrained" than a plan-space refinement that does not. Specifically, consider the action sequences that contain an action intervening between the actions corresponding to the producer and consumer steps of an establishment, and deleting the condition being established. These cannot belong to the candidate set of the plans with bookkeeping (protection) constraints, but may belong to the candidate set of plans without protection constraints. 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 (since a single abstract action can be seen as a stand in for all of the primitive actions that it can be eventually 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.

5.2 Empirical Study of Tradeoffs in Refinement Planning

The parameterized and unified understanding of refinement planning provided in this article allows us to ask specific questions about the utility of specific design choices, and answer them through normalized empirical studies. Here, we will look at two choices—use of

tractability refinements and *bookkeeping (protection) strategies*—since many existing planners differ along these dimensions [Kambhampati, Knoblock and Yang, 1996].

How important is "least commitment"? (contd)

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 plansets into the search space*. Although most traditional refinement planners do split planset components, many recent planners such as Graphplan [Blum and Furst, 1995] (see Section 6.2), handle plansets 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 plansets produced by the different refinements (which in turn may depend on factors such as the "progressivity" of the refinement -- i.e., how many candidates of the parent plan it is capable of eliminating from consideration), and the ease of propagating constraints on the planset (see Section 6.2).

Empirical results [Kambhampati, Knoblock and Yang, 1996] show that tractability refinements lead to reductions in search time only when the additional linearization they cause has the side-effect of reducing the number of establishment possibilities significantly. This happens in domains where there are conditions that are asserted and negated by many actions. Results also show that protection strategies have an effect on performance only in the cases where solution density is so low as to make the planner look at the full search space.

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

5.3 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 planset components completely into the search space). We are of course interested in selecting a planner for which the given population of problems are "easy". In order to do this, we need to relate the problem and planner characteristics to the ease of solving that problem by that planner. If we make the reasonable assumption that planners will solve a conjunctive goal problem by solving the individual subgoals serially (i.e., 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 may have to backtrack over a plan that it made for one subgoal, in order to achieve the second subgoal.

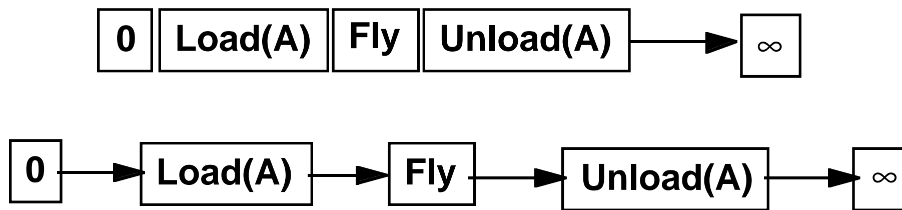

Figure 33. Different partial plans for solving the same subgoal

A subplan for a subgoal is a partial plan all of whose linearizations will execute and achieve the goal. Figure 33 shows two subplans for the AT(A,M) goal in the rocket problem. Every refinement planner **R** can be associated with a class $\mathbf{P_R}$ of subplans it is capable of producing for a subgoal. For example, for the goal At(A,M) in the rocket problem, a planner using purely state-space refinements will produce "prefix" plans of the sort shown at the top of Figure 33, which have steps only in the head, while a pure plan-space planner will produce "elastic" plans of the sort shown on the bottom, which only have steps in the middle.

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** [Barrett and Weld, 1994; Kambhampati et. al., 1996] with respect to a class of plans $\mathbf{P_R}$, if every subplan of one goal belonging to $\mathbf{P_R}$ can be eventually refined into a subplan for solving both goals. If all goals in a domain are pair-wise trivially serializable with respect to the class of plans produced by a planner, then clearly plan synthesis in that domain is easy for that planner (since the complexity will be linear in the number of goals).

It turns out that the level of commitment inherent in a plan class is a very important factor in deciding serializability. Clearly, the lower the commitment of plans in a given class, the higher the chance of trivial serializability. For example, the plan at the top of Figure 33 cannot be extended to handle the subgoal At(B,M) while the plan at the bottom can. This is why many domains with subgoal interactions are easier for plan-space planners than for state-space planners [Barrett and Weld, 1992].

The preceding does not imply a dominance of state-space planners by plan-space planners however. In particular, the lower the commitment, the higher also is the cost of handling plans, in general. Thus, the best guideline is to select the refinement planner with the highest commitment, and with respect to whose class of (sub)plans, most goals in the domain are trivially serializable. Empirical studies show this to be an effective strategy [Kambhampati et. al., 1996].

6. Scaling-up Refinement Planners

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

6.1 Scale-up through customization

Customization can be done in a variety of ways. The first is to bias the search of the planner with the help of control knowledge acquired from the user. As we discussed earlier, non-primitive 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 et. al., 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 very tricky. One intriguing idea is to "fold in 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. We have started a project on implementing this approach using Kestrel interactive software synthesis system, and the preliminary results have been promising [Srivastava and Kambhampati, 1996].

Another way of customization is to use learning techniques and make the planner learn from its failures and successes. The object of learning may be acquisition of search control rules that advise the planner what search branch to pursue [Minton et. al., 1989; Kambhampati, Katukam and Qu, 1996], or the acquisition of typical planning cases which can then be instantiated and extended to solve new problems [Kambhampati and Hendler, 1992; Veloso and Carbonell, 1993; Ihrig and Kambhampati, 1996]. This is a very active area of research and a sampling of papers can be found in the machine learning sessions at AAAI and IJCAI.

6.2 Scale-up through Disjunctive representations and constraint satisfaction techniques

Another way of scaling up refinement planners is to directly address the question of search space explosion. Much of this explosion is due to the fact that all existing planners

reflexively split the plan set components into the search space. We have seen earlier (Figure 13) that this is not required for completeness of refinement planning.

So, let us examine the consequences of not splitting plansets. 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 off, keeping plan sets together may lead to very unwieldy data structures. The way to get around this is to “internalize” the disjunction in the plansets so that we can represent them more compactly (see below). The second potential problem is that we may just be transferring the complexity from one place to another—from search space size to solution extraction. This may be true. However, there are two reasons to believe that we may still perform better. First, as we mentioned earlier, solution extraction can be cast as a model-finding activity, and there have been a slew of very efficient search strategies for propositional model finding [Selman and Kautz, 1992; Crawford and Auton, 1996]. Second, we may be able to do even better by directly refining the disjunctive plans. I will now elaborate these ideas.

6.2.1 Disjunctive Representations

The general idea of disjunctive representations is to allow disjunctive step, ordering, and auxiliary constraints into a plan. Figure 34 and Figure 35 show two examples that illustrate the compaction we can get through them. The three plans on the left in Figure 34 can be combined into a single disjunctive step, with disjunctive contiguity constraints. Similarly, the two plans in Figure 35 can be compacted by 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 from the fact that the presence of the disjunctive constraint $\mathbf{c}_1 \vee \mathbf{c}_2$ in a partial plan constrains its candidates to be consistent with either the constraint $\mathbf{c}_1$ or the constraint $\mathbf{c}_2$.

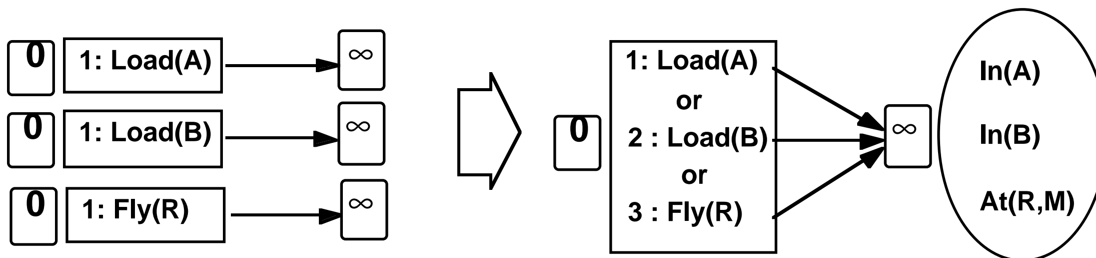

Figure 34. Disjunction over state-space refinements.

Disjunctive representations clearly lead to a significant increase in the cost of plan handling. For example, in the disjunctive plan in Figure 34 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. Similarly, in the disjunctive plan in Figure 35 , 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.

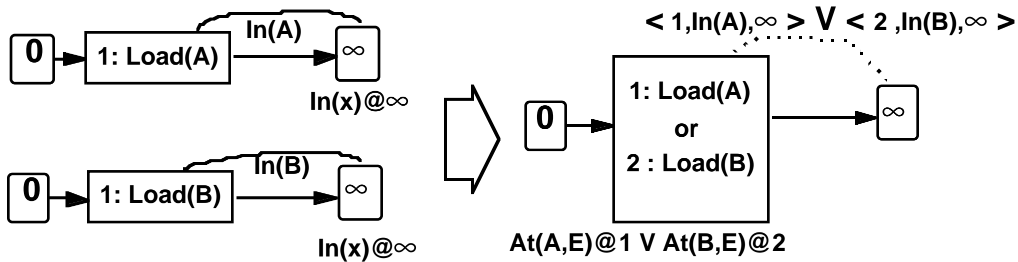

Figure 35. Disjunction over plan-space refinements

This uncertainty leads to two problems -- (a) how are we to "refine" disjunctive partial plans? and (b) 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 which is a solution (in the sense that the preconditions of all steps, including the final goal step are satisfied in the states preceding them).

6.2.2 Refining Disjunctive Plans

In order to handle disjunctive plans directly, we need to generalize the particulars of the refinement strategies, as these are clearly developed only for partial plans without disjunction (see Section 4). For example, for the disjunctive plan on the right in Figure 34, 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 be applied to the plan prefix next. Similarly, for the disjunctive plan in Figure 35, 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 At(A,E) precondition or the At(B,E) precondition or both.

Disjunctive Representations and HTN Refinement

As we noted in Section 4.4, HTN refinement introduces non-primitive actions into a partial plan. The presence of a non-primitive 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 non-primitive 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 pre-reduction refinements, considering each way of reducing the non-primitive task in a different search branch. The solution extraction is done only on non-disjunctive plans. The utility of the disjunction 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.

One way of refining such plans is to handle the uncertainty in a conservative fashion. For example, for the plan in Figure 34, 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). We can thus consider a variation 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 version that operates on non-disjunctive plansets. To see the latter, note that it is possible that even though the preconditions of an action are in the union of effects, there is no real way for that action to take place. For example, although the preconditions of "unload at moon" action may seem satisfied, it is actually never going occur as the second step in any solution because Load() and Fly() cannot be done at the same time. This brings up an important issue-- *disjunctive plans can be refined at the expese 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 doing constraint propagation along with refinements. For example, steps 1 and 3 cannot both occur in the first step (since their preconditions and effects are interacting) . This tells us that the second state may either have In(A) or At(R,M), but not both. Here the interaction between the 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 which needs both In(A) and At(R,M) can be ignored in refining this plan. This is an example of using constraint

propagation to reduce the number of refinements generated. The particular strategy shown here is employed by Blum and Furst's [1995] Graphplan algorithm (see the sidebar).

Similar techniques can be used to refine the disjunctive plan in Figure 35. For example, 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 is an instance of constraint propagation on orderings and reduces the number of establishment possibilities that plan space refinement has to consider at the next iteration.

6.2.3 Open issues in planning with disjunctive representations

In the last year or so several efficient planners have been developed which can be understood in terms of disjunctive representations. These include Graphplan [Blum and Furst, 1995], SATPLAN [Kautz and Selman, 1996], DESCARTES [Joslin and Pollack, 1996] and UCPOP-D [Kambhampati and Yang, 1996].

There are however many issues that need careful attention. For example, research in constraint satisfaction literature shows that propagation and refinement can have synergistic interactions. A case in point is 8-queens problem, where constraint propgation becomes possible only after we commit to the placement of at least one queen. This raises the possibility that best planners may be doing controlled splitting of plan sets (rather than no splitting at all) to facilitate further constraint propagation. Figure 36 shows a generalized refinement planning template that supports controlled splitting of plansets. Step 3 in the algorithm splits a planset into k components. Depending upon 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 modeld by choosing k to be equal to the number of components of the planset. Planners like Graphplan that do no do any splitting can be modeled by choosing k to be equal to 1.

In between are the planners like Descartes and UCPOP-D that split a planset into some number of branches that is less than the number of planset components. One immediate question is exactly how many branches should a planset be split into? Since the extent of propagation depends on the amount of shared sub-structure between the disjoined planset components, one way of controlling splitting would be to keep plans with shared sub-structure together.

Refine (_P_: (a disjunctive) Planset)

0*. If «_P_» is empty, Fail.
1. If a minimal candidate of _P_ is a solution, terminate.
2. Select a refinement strategy _R_. Appply _R_ to _P_ to get a new plan set _P'_
3. Split _P'_ into k plansets
4. Simplify plansets by doing constraint propagation.
5. Non-deterministically select one of the plansets P'_i Call Refine(P'_i)

Figure 36. Refinement planning with controlled splitting.

Table 2. A spectrum of refinement planners

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

The relative support provided by various types of refinements for planning with disjunctive representations needs to be understood. The tradeoff analyses described in Section 5 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 while Graphplan, that does forward state space refinement on disjunctive plans, is very efficient, there is no comparable planner that does backward state space refinements.[5]

[5] Our informal studies show that straightforward extensions to Graphplan to make it work in the backward direction fail to compete with Graphplan. This is because while Graphplan's efficiency derives from the propagation of mutual exclusion constraints to focus refinement

Finally, the interaction between the refinements and solution extraction process needs to be investigated more carefully. As we indicated in the preceding discussion, if we have access to efficient solution extraction procedures, we can reduce the role of refinements significantly. For example, the disjunctive plan in Figure 34 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 work. The approach we sketched involves using the refinement strategies to reduce the amount of disjunction -- e.g. to realize that the 2nd 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.

---

of disjunctive plans, there seem to be no analogous constraints in the case of "backward" Graphplan.

Understanding Graphplan

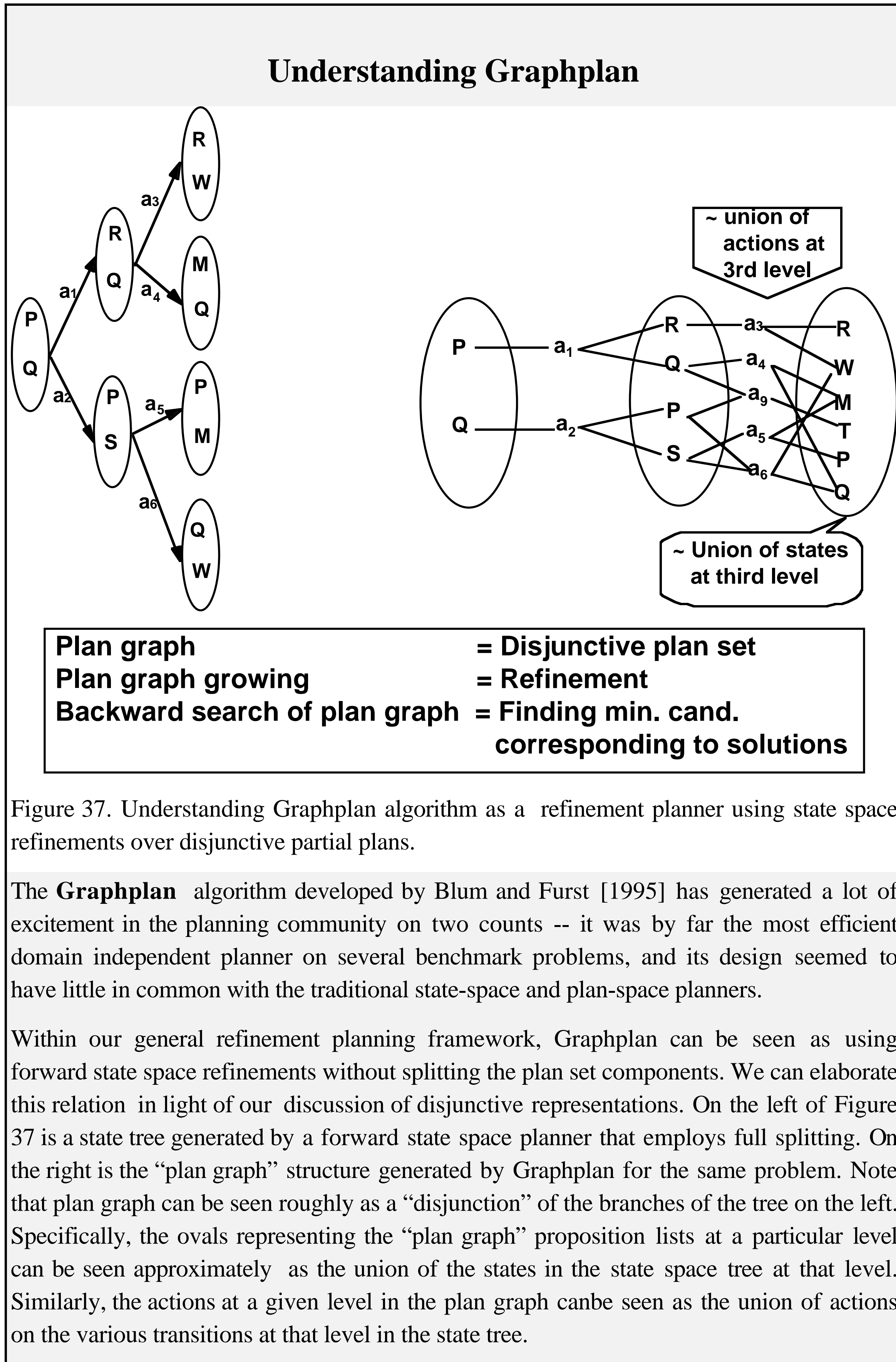

Figure 37. Understanding Graphplan algorithm as a refinement planner using state space refinements over disjunctive partial plans.

The **Graphplan** algorithm developed by Blum and Furst [1995] has generated a lot of excitement in the planning community on two counts -- it was by far the most efficient domain independent planner on several benchmark problems, and 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 Figure 37 is a state tree generated by a forward state space planner that employs full splitting. On the right is the “plan graph” structure generated by Graphplan for the same problem. Note that plan graph can be seen roughly 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 approximately as the union of the states in the state space tree at that level. Similarly, the actions at a given level in the plan graph canbe seen as the union of actions on the various transitions at that level in the state tree.

Understanding Graphplan (contd.)

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 Section 6.2.2, this is part of the price we pay for refining disjunctive plans directly. However, the propagation of of mutual exclusion constraints allows Graphplan to keep a reasonably close correspondence 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 in itself is not new. As described in Section 4.1.1, it is possible to generalize forward state space refinement to consider sets of non-interfering actions simultaneously. Indeed, Graphplan's big win over traditional state-space planners (that do full splitting of planset components into the search space) comes from its handling of planset components without splitting. This in turn is supported by its use and refinement of disjunctive plans. Our experiments with Graphplan confirm this hypothesis.

The strong connection between forward state space refinement and Graphplan also suggests that techniques such as "Means-ends analysis" which 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 few are relevant to the top level goals of the problem. It would thus be interesting to deploy the best methods of Means-ends analysis (such as the Greedy Regression Graph approach proposed by McDermott [1996]), to isolate potentially relevant actions and only use those to grow the plan graph.

Refinement planning vs. Encoding Planning as Satisfiability

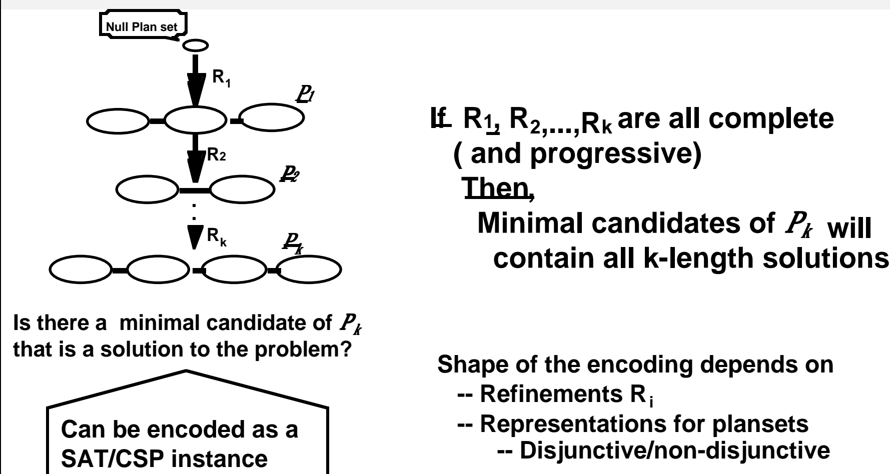

Figure 38. Relating refined plan at k-th level to SATPLAN encodings.

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

In the context of the general refinement planning framework, we 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 Figure 38, 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. So, 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 the type of refinements used in coming with the **k**-length plan-set and how the plansets themselves are represented (recall that disjunctive representations can reduce the progressivity of refinements).

Refinement planning vs. Encoding Planning as Satisfiability (contd)

Indeed, we can make rough comparisons between the different encodings explored by Kautz et. al., and the refinement strategies and representations of plan sets that could give rise to them. For example, Kautz et. al.'s linear and parallel encodings correspond to the use of forward state-space refinements, with the parallel ones corresponding to disjunctive plan-set representations. Similarly, the SNLP encodings correspond to the use of plan-space refinements. Given that interleaving refinement strategies has been shown to be a good idea in improving the cost of refinement planning (see Section 4.6), we can explore the utility of encodings based on interleaved refinements.

Figure 39. Relating encodings based on minimal candidates of refined plans to direct encodings generated by SATPLAN.

One question that needs to be answered is whether the relation between k-length encodings and the minimal candidates of k-level plan sets is just a matter of theoretical curiosity, or whether it has any practicial significance.

Refinement planning vs. Encoding Planning as Satisfiability (contd)

I believe that basing encodings on k^{th} level plan sets, derived by the application of k complete refinements, leads to SAT instances that are "smaller" on the whole. Specifically, both the number of variables in the SAT as well as the size of the individual clauses sizes can reduce by starting from k-level plansets.

Let me illustrate this point with an example shown in Figure 39 that involves forward state space refinements. Here we have three different ways of generating encodings based on the forward state space refinement. On the left, we do forward state space refinement on individual components of the plansets, generating all legal k-length prefixes (which can be searched to see if any of them correspond to a solution). On the righthand side, we avoid refinements and generate the encoding directly using the methods used by Kautz et. al. [1996] -- essentially assuming that any of the actions in the domain can occur at any step of the k-length plan, and setting up constraints to ensure that the actions that do occur will form a solution. The middle picture corresponds to doing forward state space refinement on the disjunctive plan representations (specifically, the structure here is similar to the k-level plan graph generated by Graphplan [Blum and Furst, 1995]-- which, as we observed earlier, can be seen as the disjunctive representation of k-level planset). It is interesting to note that as we go from left to right, the amount of disjunction increases. For example, if we ask the question--what can be the actions at level 2, the left most encoding will say they can only be one of 5. The right most one says they can be one of any available actions in the domain, while the middle one says that they can be one of six

Clearly, the encodings sizes will be largest for the left and smallest for the right. At the same time, the cost of generating the encoding is lowest on right and highest on left. Thus, a happy medium is likely to be reached in the middle--ie. encodings based on disjunctive refined plans. Further elaboration on these observations can be found in [Kambhampati and Yang, 1996].

7. Conclusion and Future Directions

Let me now conclude by re-iterating that refinement planning continues to provide the theoretical backbone to 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 tradeoffs and also 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 on planning under non-classical assumptions. Indeed, the operation of several non-classical planners described in the literature can be understood from refinement planning point of view. These 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]

There are a variety of exciting avenues open for further research in refinement planning.To begin with, we have seen that despite the large number of planning algorithms, there are really only two fundamentally different refinement strategies—plan-space and state-space ones. It would be very interesting to see if there are novel refinement strategies with better properties. To some extent this might 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 ones, although its overall operation conforms to the framework of refinement followed by 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 tradeoffs in plan synthesis, given the availability of planners using disjunctive representations. Concepts like subgoal interactions do not make too much sense in these scenarios.

Finally, there is a lot to be gained by porting the refinement planning techniques to planning under non-classical scenarios. For example, how can we use the analogues 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 HTN refinements, see [Kambhampati and Srivastava, 1996]. For a more technical discussion of the issues in disjunctive planning, see [Kambhampati and Yang, 1996]. Most of these papers can be found at URL **http://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 tradeoffs between state-space and plan-space planning algorithms. Minton et. al. [1994] provide a comparison between partial order and total order plan-space planning algorithms. This study can be seen as focusing on the effect of pre-ordering 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 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 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, plan reuse, as well as interleaving 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, the readers may refer to the on-line notes from the graduate level course on planning that I teach at Arizona State University. The notes can be found at the URL **http://rakaposhi.eas.asu.edu/planning-class.html**.

In addition to the national and international joint conferences, planning related papers appear in the bi-annual International Conference on AI Planning Systems, and European Conference (formerly European Workshop) on planning systems. There is a mailing list for planning-related discussions and announcements. For subscription, send an email to **planning@asu.edu**.

Acknowledgements:

This article is based on an invited talk given at the 1996 National Conference of Artificial Intelligence, held at Portland, OR. My understanding of refinement planning issues have matured over years of discussions with various colleagues. Notable among these are Tony Barrett, Mark Drummond, Kutluhan Erol, Jim Hendler, Eric Jacopin, Craig Knoblock, David McAllester , Drew McDermott, Dana Nau, Ed Pednault, David Smith, Austin Tate, Dan Weld, and Qiang Yang. My own students and participants of the ASU planning seminar have been invaluable as sounding boards and critiques of my half-baked ideas. I would especially like to acknowledge Bulusu Gopi Kumar, Suresh Katukam, Biplav Srivastava and Laurie Ihrig. Amol Mali, Laurie Ihrig, and Jude Shavlik read a version of this article and provided helpful comments. Finally, I would like to thank Dan Weld and Nort Fowler for their encouragement on this line of research. Some of this research has

been supported by grants from NSF (research initiation award IRI-9210997; NSF Young Investigator award IRI-9457634) and the ARPA Planning Initiative under Tom Garvey's management (F30602-93-C-0039, F30602-95-C-0247).

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 Fox, M. and Zweben, M., Editors. Knowledge based scheduling. Morgan Kaufman. San Mateo. California.

Bacchus, F. and Kabanza, F. 1995. Using temporal logic to control search in a forward chaining planner. In: *Proceedings of 3rd European Workshop on Planning*. IOS Press. Amsterdam.

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 International Joint Conference on Artificial Intelligence.* Morgan Kaufmann.

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 3rd International Conference on AI Planning Systems. AAAI Press.* 53-61.

Crawford, J. and Auton, L. 1996. Experimental results on the crossover poin in random 3SAT. *Artificial Intelligence,* 81.

Dean, T. and Kambhampati, S. 1996. Planning and Scheduling. In CRC Handbook or Computer Science and Engineering. CRC Press. 1996.

Drummond, M. 1989. Situated control rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning,* 103-113. Morgan Kaufmann.

Erol, K. Nau, D and Subrahmanian, V. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75-88.

Erol, K. 1995. Hierarchical task network planning systems: Formalization, Analysis and Implementation. Ph.D. Dissertation. University of Maryland, College Park, MD.

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 *First International Conference on expert planning systems,* 70-75. Brighton, UK. Inst itute of Electrical Engineers.

Ginsberg, M. 1996. A new algorithm for generative planning. In *Proceedings of the fifth International Conference on Principles of Knowledge Representation and Reasoning*. 186-197. Morgan Kaufmann.

Green, C. 1969. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence,* 219-239. Washington D.C.

Golden, K., Etzioni, O. and Weld, D. 1996. XII: Planning with universal quantification and incomplete information. Technical Report. Dept. of CSE, University of Washington .

Haas, A. 1987. The case for domain-specific frame axioms. In “The frame problem in Artificial Intelligence”, Proceedings of the 1987 workshop. Brown, F.M., Editor. Morgan Kaufmann publishers.

Ihrig, L. and Kambhampati, S. 1996. Design and implementation of a replay framework based on a partial order planner. In *Proceedings of National Conference on Artificial Intelligence.* 849-854. AAAI Press.

Joslin, D. and Pollack, M. 1996. Is “early commitment” in Plan Generation ever a good idea? In *Proceedings of National Conference on Artificial Intelligence.* 1188-1193. AAAI Press.

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. 1995. Univ ersal classical planner: An algorithm for unifying state-space and plan-space planning. In: *Proceedings of 3rd European Workshop on Planning*. IOS Press. Amsterdam.

Kambhampati, S., and Srivastava, B. 1996. Unifying classical planning approaches. Technical Report 96-006. Dept. of Computer Science and Engg., Arizona State University, Temple, AZ.

Kambhampati, S., Ihrig, L. and Srivastava, B. 1996. A candidate-set based analysis of subgoal interaction in conjunctive goal planning. In: *Proceedings of 3rd International Conference on AI Planning Systems. AAAI Press.* 125-133.

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., 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.

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. Morgan Kaufmann.

Kautz, H. and Selman, B. 1996. Pushing the envelope: Planning Propositional Logic and Stochastic Search. In *Proceedings of National Conference on Artificial Intelligence.* 1194-11201. AAAI Press.

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. 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 nith national conference on artificial intelligence.* 634-639. AAAI Press.

McDermott, D. A Heuristic estimator for means-ends analysis in planning. In: *Proceedings of 3rd International Conference on AI Planning Systems. AAAI Press.* 142-149.

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.

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.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Tioga Press. Palo Alto.

Pednault, E. 1988. Synthesizing plans that contains actions with context-dependent effects. *Computational Intelligence,* 4(4):356-372.

Pednault, E. 1994. ADL and the state-transition model of action. *Journal of logic and computation.* 4(5):467-512.

Penberthy, S. and Weld, D.1992. UCPOP: A sound, complete, partial order planner for ADL. In: *dings of Third International Conference on the principles of knowledge representation,* 103-114.

Penberthy, S. and Weld, D. 1994. Temporal planning with continuous change. In: *Proceedings of the 12th national conference on Artificial Intelligence*. 1010-1015. AAAI Press.

Sacerdoti, E. 1972. The non-linear nature of plans. In: *Proceedings Intl. Joint Conf. on Artificial Intelligence.*

Selman, D., Levesque, H.J., and Mitchell, D. 1992. GSAT: a new method for solving hard satisfiability problems. In: *Proceedings of National Conference on Artificial Intelligence*. 440-446. AAAI Press.

Srivastava, B. and Kambhampati, S. Synthesizing customized planners from specifications. Technical Report 96-014. Dept. of Comp. Sci. and Engg. Arizona State University.

Tate, A. 1975. Interacting goals and their use. In:*Proceedings of 4th International Joing Conference on Artificial Intelligence.* 215-218.

Tate, A. 1977. Generating project networks. In:*Proceedings of 5th International Joing Conference on Artificial Intelligence.* 888-893.

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.* Winter:27-61.

Wellman, M. 1987. Dominance and subsumption in constraint-posting planning. In: *Proceedings of Intl. Joint Conference on Artificial Intelligence*. 884-890.

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

Wilkins, D. 1988. *Practical Planning.* Morgan Kaufmann.