Unifying Classical Planning Approaches

Subbarao Kambhampati* & **Biplav Srivastava**
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
email: {rao,biplav}@asu.edu
WWW: http://rakaposhi.eas.asu.edu/yochan.html

ASU CSE TR 96-006; July 1996

Abstract

State space and plan space planning approaches have traditionally been seen as fundamentally different and competing approaches to domain-independent planning. We present a plan representation and a generalized algorithm template, called UCP, for unifying these classical planning approaches within a single framework. UCP models planning as a process of refining a partial plan. The alternative approaches to planning are cast as complementary refinement strategies operating on the same partial plan representation. UCP is capable of arbitrarily and opportunistically interleaving plan-space and state-space refinements within a single planning episode, which allows it to reap the benefits of both. We discuss the coverage, completeness and systematicity of UCP. We also present some preliminary empirical results that demonstrate the utility of combining state-space and plan-space approaches. Next, we use the UCP framework to answer the question "which refinement planner is best suited for solving a given population of problems efficiently?" Our approach involves using subgoal interaction analysis. We provide a generalized account of subgoal interactions in terms of plan candidate sets, and use it to develop a set of guidelines for choosing among the instantiations of UCP. We also include some preliminary empirical validation of our guidelines. In a separate appendix, we also describe how the HTN planning approach can be integrated into the UCP framework.

*This research is supported in part by NSF research initiation award (RIA) IRI-9210997, NSF young investigator award (NYI) IRI-9457634 and ARPA/Rome Laboratory planning initiative grants F30602-93-C-0039 and F30602-95-C-0247. We thank Laurie Ihrig and Amol Dattatreya Mali for helpful comments.

Contents

1 Introduction

Domain independent classical planning techniques fall into two broad categories– state space planners which search in the space of states, and plan space planners which search in the space of plans. By and large, these approaches have been seen as competing rather than complementary, and several research efforts [2, 35] have been focused on showing the relative superiority of one over the other. Our message in this paper is that the two approaches can coexist and complement each other, that hybrid planners that use both approaches can perform better than pure plan-space or state-space planners, and that a unified view clarifies our current understanding of classical refinement planning. To this end, we develop a unified framework that encompasses both state-space and plan-space planning approaches.

Previously, we have shown that viewing planning as a process of refinement search provides a powerful framework for unifying the large variety of plan-space planning approaches [19, 17, 18]. In this paper, we show that the same framework can be extended to unify state-space planning and plan-space planning approaches. In particular, we present UCP [21], a generalized algorithm for classical planning. UCP covers classical planning approaches through three complementary refinement strategies, corresponding respectively to the plan-space, forward state-space and backward state-space planning approaches. If it chooses to always use plan-space refinements, it becomes a pure plan-space planner, and if it chooses to always use state-space refinements, it becomes a pure state-space planner.

Since all three refinement strategies operate on the same general partial plan representation, UCP also facilitates opportunistic interleaving of the plan-space and state-space refinements within a single planning episode. Such interleaving could produce a variety of hybrid planners (including the traditional means-ends analysis planners such as STRIPS and PRODIGY [10]), and can thus help us reap the benefits of both state-space and plan-space approaches in a principled manner. The traditional question "when should a plan-space planner be preferred over state space planners?" [2, 35] can now be posed in a more sophisticated form: "*when should a plan-space refinement be preferred over a state-space refinement (or vice versa) within a single planning episode?*."

Since different instantiations of UCP correspond to a large variety of classical refinement planners, we can also use the UCP framework as a substrate to revisit the question of "which classical planner is best suited for efficiently solving a given population of problems?" Specifically, we will consider the issue of selecting the right type of UCP instantiation for a given problem population, and analyze it in terms of subgoal and subplan interactions. Although subgoal and subplan interactions have been studied in the past [2, 24], they have been couched in terms of the details of specific classical planners. We present a more generalized analysis based on candidate set of subplans for individual subgoals [22]. Since subgoals interact in terms of their subplans, we will start by characterizing two important ways in which partial plans may interact – mergeability and serial extensibility. Informally, two subplans are mergeable if there is a way of combining the constraints in them to generate a new plan that solves the subgoals solved by both subplans. A subplan is said to be serially extensible with respect to another subgoal if there is a way of adding further constraints to the subplan such that the new plan solves both subgoals. The concepts of independence and serializability of subgoals are derived by generalizing mergeability and serial extensibility over classes of partial plans. We will then present some guidelines, based on this analysis, for selecting among the instantiations of UCP, and validate the guidelines with some empirical results.

Finally, in an appendix at the end of the paper, we show how the HTN planning approach can be integrated into the UCP framework. This involves significant extensions to the UCP framework to handle non-primitive tasks in partial plans.

The rest of this paper is organized as follows. In Section 2, we review the preliminaries of refinement planning. Section 3 describes the representation and semantics for partial plans used in UCP. In Section 4, we describe the UCP algorithm and illustrate its operation through an example. Section 5 discusses the coverage, completeness and systematicity of the UCP algorithm. Section 6 describes several heuristic control strategies for UCP. Section 7 empirically demonstrates the utility of opportunistic interleaving of refinements in UCP. Section 8 describes two candidate set based characterizations of subplan interactions called mergeability and serial extensibility, generalizes these notions to subgoals, and shows how the analyses subsumes previous analyses of subgoal interactions. Section 9 discusses how the analysis of subgoal interactions can be useful in selecting an appropriate refinement planner given a domain. Section 10 discusses the related work and Section 11 presents the conclusions. Appendix A presents a comprehensive treatment on the integration of HTN planning into the UCP framework.

2 Planning as Refinement Search: Overview

A planning problem is a 3-tuple $\langle I, G, \mathcal{A} \rangle$, where I is the description of the initial state, G is the (partial) description of the goal state, and $\mathcal{A}$ is a set of actions (also called "operators"). An action sequence (also referred to as ground operator sequence) S is said to be a solution for the planning problem, if S can be executed from I, and the resulting state of the world satisfies all the goals.

Refinement planners [17, 18, 19] attempt to solve a planning problem by navigating the space of *sets of potential solutions (ground operator sequences)*. The potential solution sets are represented and manipulated in the form of "partial plans."[1] Syntactically, a partial plan $\mathcal{P}$ can be seen as a set of constraints (see below). Semantically, a partial plan is a shorthand notation for the set of ground operator sequences that are consistent with its constraints. For example, if the partial plan contains a single action o and a constraint that it be the first action, then any ground operator sequence that starts with o will be a candidate of the plan. The set of all candidates is called the *candidate set* of the partial plan, and denoted by $\langle\langle \mathcal{P} \rangle\rangle$. It can be potentially infinite.

Refinement planning consists of starting with a "null plan" (denoted by $\mathcal{P}_\emptyset$), whose candidate set corresponds to all possible ground operator sequences, and successively refining the plan (by adding constraints, and thus splitting their candidate sets) until a solution is reached. Semantically, a *refinement strategy* $\mathcal{R}$ maps a partial plan $\mathcal{P}$ to a set of partial plans $\{\mathcal{P}'_i\}$ such that the candidate sets of each of the child plans are subsets of the candidate set of $\mathcal{P}$ (i.e.,

[1] For a more formal development of the refinement search semantics of partial plans, see [17, 19]

$\forall_{\mathcal{P}'_i} \langle\langle \mathcal{P}'_i \rangle\rangle \subseteq \langle\langle \mathcal{P} \rangle\rangle$). A solution extraction function takes a partial plan and check to see if one of its candidates solves the problem at hand. Since the candidate sets of partial plans are potentially infinite, solution extraction functions restrict their attention to a distinguished finite subset of the candidate set – the set of the plan's *minimal candidates*. Informally, minimal candidates are ground operator sequences in the candidate set of a plan that do not contain any actions other than those explicitly mentioned in the partial plan (see Section 3.2).

Refinement planning involves repeatedly applying refinement strategies to a partial plan until a solution can be picked from the candidate set of the resulting plan. It should be clear that since termination occurs by checking the minimal candidates, for a refinement planner to terminate, the refinements it uses should have the property that the length (in terms of number of actions) of the minimal candidates increase when a plan is refined.

A refinement strategy R is said to be **progressive** if the minimal candidate of the partial plan can increase in length after the application of R. Otherwise, it is said to be a **non-progressive** refinement. Semantically, non-progressive refinements only partition the candidate set of the plan (i.e. if $\mathcal{P}$ is refined into n plans $\mathcal{P}_1 \cdots \mathcal{P}_n$, then $\langle\langle \mathcal{P} \rangle\rangle = \cup_{i=1}^{n} \langle\langle \mathcal{P}_i \rangle\rangle$). Their syntactic operation involves "re-arranging" the steps in the current partial plan. Progressive refinements may reduce the number of candidates in consideration ($\langle\langle \mathcal{P} \rangle\rangle \supseteq \cup_{i=1}^{n} \langle\langle \mathcal{P}_i \rangle\rangle$). Their syntactic operation involves adding new steps into the plan. Non-progressive refinements can thus be thought of as doing "scheduling" (action sequencing) type activity, while the progressive refinements can be thought of as doing "planning" (or action selection).

A refinement strategy $\mathcal{R}$ is said to be **complete** if every solution belonging to the candidate set of $\mathcal{P}$ belongs to the candidate set of at least one of the child plans. $\mathcal{R}$ is said to be **systematic** if the candidate sets of children plans are non-overlapping (i.e. $\forall_{\mathcal{P}_i, \mathcal{P}_j, i \neq j} \; \langle\langle \mathcal{P}_i \rangle\rangle \cap \langle\langle \mathcal{P}_j \rangle\rangle = \emptyset$).

It is easy to see that as long as a planner uses only refinement strategies that are complete, it never has to backtrack over the application of a refinement strategy. Similarly, if all the refinement strategies are systematic, the search space of the planner will be systematic in that no ground operator sequence will belong to the candidate sets of plans in more than one branch of the search tree [25].

3 Representation of partial plans in UCP

In this section, we will develop a syntactic and semantic representation of partial plans that is adequate to support both state-space and plan-space refinements. We start by observing that whatever the representation for partial plans, a partial plan $\mathcal{P}$ is semantically a set of ground operator sequences. Thus, if $\mathcal{U}$ is the universe of ground operator sequences, then $\mathcal{P} \subseteq 2^{\mathcal{U}}$. We consider a partial plan to be a set of constraints. The semantics of each of the constraints are specified by stating which ground operator sequences satisfy that constraint. Given this, the candidate set of a partial plan is the set of ground operator sequences that satisfy all the constraints. The specific constraints that we use below are not meant to be exhaustive, but they are sufficient to model partial plans used in existing classical planners. New constraint types can be defined by simply providing their semantics in terms of satisfying action sequences.

A partial plan is a 5-tuple $\langle T, O, \mathcal{B}, \mathcal{ST}, \mathcal{L} \rangle$ where: T is the set of steps in the plan; T contains two distinguished step names t_0 and t_∞. $\mathcal{ST}$ is a symbol table, which maps step names to actions. (Note that multiple steps can be mapped to the same action.) We will assume that actions are modeled in the familiar STRIPS/ADL representation [29] with precondition and effect (aka postcondition) lists. The special step t_0 is always mapped to the dummy operator `start`, and similarly t_∞ is always mapped to `finish`. The effects of `start` and the preconditions of `finish` correspond, respectively, to the initial state and the desired goals of the planning problem. O is a partial ordering relation over T. $\mathcal{B}$ is a set of codesignation (binding) and non-codesignation (prohibited binding) constraints on the variables appearing in the preconditions and post-conditions of the operators. A ground linearization of $\mathcal{P}$ is a permutation on its steps that is consistent with O, with all variables instantiated to values that are consistent with $\mathcal{B}$.[2] $\mathcal{L}$ is a set of auxiliary constraints that restrict the allowable orderings and bindings among the steps. Three important types of auxiliary constraints are:

Interval Preservation Constraints: An *interval preservation constraint* (IPC) is specified as a 3-tuple: $(t \overset{p}{-} t')$. Syntactically, it demands that the condition p be preserved between t and t' in every ground linearization of the plan. Semantically, it constrains the candidates of the partial plan such that in each of them, p is preserved between the operators corresponding to steps t and t'.

Point Truth Constraints: A *point truth constraint* (PTC) is specified as a 2-tuple: $\langle p, t \rangle$. Syntactically, it demands that the condition p be necessarily true [6] in the situation before the step t. Semantically, it constrains all solutions of the partial plan to have p in the state in which the operator corresponding to t is executed.

Contiguity Constraints: A contiguity constraint is specified as a relation between two steps: $t_i * t_j$. Syntactically, it demands that no step intervene between t_i and t_j in any ground linearization of $\mathcal{P}$. Semantically, it constrains all candidates of the partial plan to have no operators intervening between the operators corresponding to t_i and t_j. From the definition, if t_i is immediately before t_j, it also means that $t_i \prec t_j$. Further, since t_i and t_j are contiguous, there cannot be any step t' such that $t_i \prec t' \prec t_j$, or $t_i * t'$ or $t' * t_j$.

Readers who are familiar with the refinement search view of partial order planning developed in [19, 17, 18] may note that *the only extension to the plan representation is the addition of the new type of auxiliary constraints called contiguity constraints*. We will see here that this extension is enough to handle state space refinements.

Safe Ground Linearizations: A ground linearization is said to be a **safe ground linearization** if it syntactically satisfies all the contiguity constraints, and the interval preservation constraints [19]. The semantic notion of the candidate set of the partial plan is tightly related to a

[2] Note that a ground linearization is in terms of step names while a ground operator sequence is in terms of actions. A ground linearization corresponds to a ground operator sequence, modulo the step-action mapping $\mathcal{ST}$.

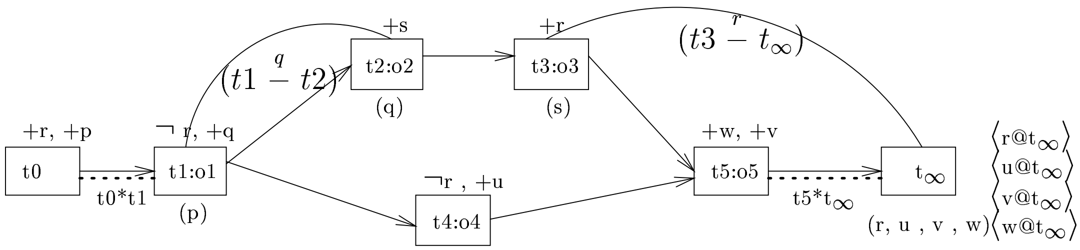


Figure 1: Example Partial Plan $\mathcal{P}_E$. The effects of the steps are shown above the steps, while the preconditions are shown below the steps in parentheses. The ordering constraints between steps are shown by arrows. The interval preservation constraints are shown by arcs, while the contiguity constraints are shown by thick dotted lines. The PTCs are used to specify the goals of the plan

syntactic notion of safe ground linearization [19, 17]. Specifically, safe ground linearizations correspond to minimal length candidates of the partial plan [19]. If a partial plan has no safe ground linearizations, it has an empty candidate set.

Example: Figure 1 illustrates these definitions through an example plan $\mathcal{P}_E$, which contains the steps $\{t_0, t_1, t_2, t_3, t_4, t_5, t_\infty\}$, the symbol table $\{t_0 \rightarrow \texttt{start}, t_\infty \rightarrow \texttt{end}, t_1 \rightarrow o_1, t_2 \rightarrow o_2, t_3 \rightarrow o_3, t_4 \rightarrow o_4, t_5 \rightarrow o_5\}$, the ordering constraints $\{t_1 \prec t_2, t_2 \prec t_3, t_3 \prec t_5, t_1 \prec t_4, t_4 \prec t_5\}$, the contiguity constraints $\{t_0 * t_1, t_5 * t_G\}$, the interval preservation constraints $\{(t_1 \overset{q}{-} t_2), (t_3 \overset{r}{-} t_\infty)\}$, and the point truth constraints $\{\langle r, t_\infty\rangle, \langle u, t_\infty\rangle, \langle v, t_\infty\rangle, \langle w, t_\infty\rangle\}$. The ground operator sequence $o_1 o_2 o_4 o_3 o_2 o_2 o_5$ is a candidate of the plan $\mathcal{P}_E$, while the ground operator sequences $o_3 o_1 o_2 o_3 o_3 o_5$ and $o_1 o_2 o_3 o_4 o_5$ are not candidates of $\mathcal{P}_E$ (the former violates the contiguity constraint $t_0 * t_1$, and the latter violates the interval preservation constraint $(t_3 \overset{r}{-} t_\infty)$). (Notice that the candidates may have more operators than the partial plan.) $t_0 t_1 t_2 t_4 t_3 t_5 t_\infty$ is a safe ground linearization, while $t_0 t_1 t_2 t_3 t_4 t_5 t_\infty$ is not a safe ground linearization (since the interval preservation constraint $(t_3 \overset{r}{-} t_\infty)$ is not satisfied by the linearization). The safe ground linearization corresponds to the minimal candidate (ground operator sequence) $o_1 o_2 o_4 o_3 o_5$.

Goal Achievement: A ground operator sequence S is said to achieve a goal g_1 if executing S from the initial state leads to a world state where g_1 is true. A partial plan P is said to bear a solution if one of its safe ground linearizations corresponds to a ground operator sequence that achieve g_1.

3.1 World states and partial plans

We will now define some structural attributes of a partial plan that relate the partial plan to states of the world that are expected during the execution of its candidates. These definitions will be useful in describing the UCP algorithm.

Given a plan $\mathcal{P}$, a step $t_{\mathcal{H}}$ is said to be the **head step** of the plan if there exists a sequence of steps $t_1 \cdots t_n$ such that $t_0 * t_1 * t_2 \cdots t_n * t_{\mathcal{H}}$ and there is no step t' such that $t_{\mathcal{H}} * t'$. The sequence of steps $t_0, t_1, \cdots t_n, t_{\mathcal{H}}$ is called the **header** of the plan. The state resulting from the application of the operators corresponding to the header steps, in sequence, to the initial state, is called the **head state**. The **head fringe** of a plan $\mathcal{P}$ is the set of all steps t that can possibly come immediately after the $t_{\mathcal{H}}$ in some ground linearization of the plan (i.e., all t such that $t_{\mathcal{H}} \prec t$ and there is no step t' such that $\Box(t_{\mathcal{H}} \prec t' \prec t)$).

Similarly, a step $t_{\mathcal{T}}$ of a plan $\mathcal{P}$ is said to be the **tail step** of the plan if there exists a sequence of steps $t_1 \cdots t_n$ such that $t_{\mathcal{T}} * t_1 * t_2 \cdots t_n * t_\infty$ and there is no step t' such that $t' * t_{\mathcal{T}}$. The sequence of steps $t_{\mathcal{T}}, t_1, \cdots t_n, t_\infty$ is called the **trailer** of the plan. The state resulting from the backward application of the operators corresponding to the trailer steps, in sequence, to the goal state, is called the **tail state**. The **tail fringe** of a plan $\mathcal{P}$ is the set of all steps t that can possibly come immediately before the tail step $t_{\mathcal{T}}$ in some ground linearization of the plan (i.e., all t such that $t \prec t_{\mathcal{T}}$ and there is no step t' such that $\Box(t \prec t' \prec t_{\mathcal{T}})$).

Example: In the example plan $\mathcal{P}_E$ shown in Figure 1, t_1 is the head step and t_5 is the tail step. $t_0 t_1$ is the header and $t_5 t_\infty$ is the trailer. $\{t_2, t_4\}$ is the head fringe, and $\{t_3, t_4\}$ is the tail fringe. Head state is $p \wedge q$ while the tail state is $r \wedge u$.

3.2 Relating Partial Plans and Refinements

Figure 2 explicates the schematic relations between a partial plan, its candidate set, and refinements. Each partial plan corresponds to a set of ground linearizations GL. The subset of GL that satisfy the auxiliary constraints of the plan are said to be safe-ground linearizations SGL of the plan. Each safe ground linearization of the plan corresponds to a ground operator sequence S_m which is a minimal candidate of the partial plan. A potentially infinite number of additional candidates may be derived from each minimal candidate of the plan by augmenting (padding) it with additional ground operators, without violating any auxiliary constraints. Thus, the candidate set of a partial plan is potentially infinite, but the set of its minimal candidates is finite. The solution extractor functions start by looking at the minimal candidates of the plan to see if any of them are solutions to the planning problem. Refinement process can be understood as incrementally increasing the length of these minimal candidates so that ground operator sequences of increasing lengths are examined to see if they are solutions to the problem. This description points out that for a refinement strategy to form the basis for a complete refinement planning algorithm, it should be able to increase the length of the minimal candidates. Not all refinements used by refinment planners have this property. This is why we distinguish between *progressive* refinements which have the property and *non-progressive* refinements which do not.

Notice the dual view of plans that is being developed here. The correctness and the execution semantics of a plan P are defined in terms of its minimal candidates alone, while its refinements are defined in terms of its full candidate set. The candidate set of P may contain

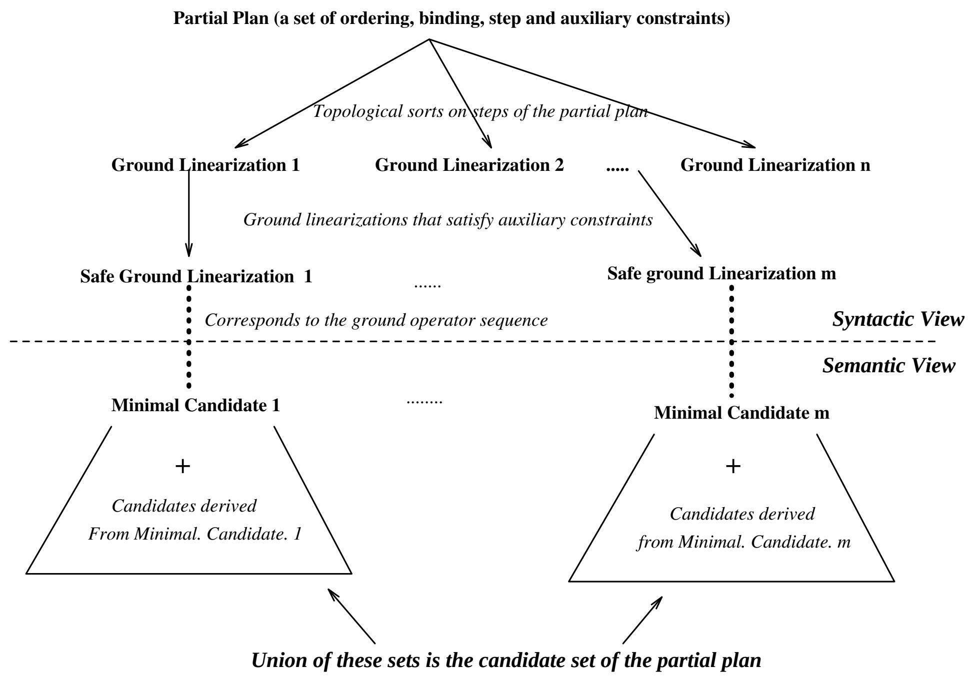


Figure 2: A schematic illustration of the relation between a partial plan and its candidate set. The candidate set of a partial plan contains all ground operator sequences that are consistent with its constraints. These can be seen in terms of minimal candidates (which correspond to the safe ground linearizations of the partial plan) and ground operator sequences derived from them by adding more operators.

possibly infinitely more other ground operator sequences each of which may or may not be solutions.[3]

In what follows, we will show that state-space planning and plan-space planning approaches can essentially be modeled as two different varieties of refinement strategies operating on the same partial plan representation.

3.3 Classification of Partial Plans

The plan representation discussed in this section is fairly general to allow many varieties of plans. In the following we identify subclasses of partial plans which have interesting properties from subgoal interaction point of view. The subclasses will be identified in terms of the syntactic restrictions on the plan constraints (see Figure 3).

A plan P for achieving a subgoal g from the initial state I is called a **prefix plan** if all the steps of the plan, except t_∞, are all contiguous. Any feasible prefix plan will also have the property that the prefix steps can be executed from the initial state in sequence (as otherwise, their candidate sets cannot contain any executable operator sequences). For example, a prefix

[3]In [11], Ginsberg argues for an elimination of this separation saying that a "good" plan will have a large candidate set most of which will be able to achieve the goals of the problem.

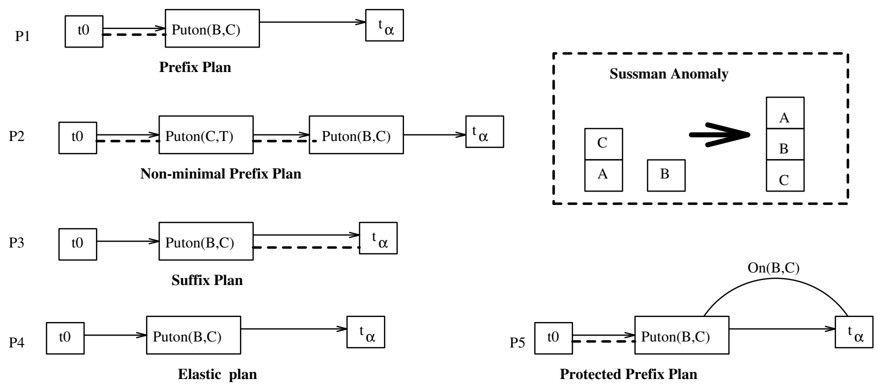


Figure 3: The Sussman Anomaly problem, and a variety of plans for solving the subgoal $On(B, C)$. Each plan class is identified with a descriptive name. The solid arrows signify precedence relations while the solid arrows with dashing under them signify contiguity relations.

plan for the subgoal $On(B, C)$ in the Sussman anomaly problem is P_1: $t_0 * Puton(B, C) \prec t_\infty$ shown in Figure 3. Planners that do forward search in the space of states produce feasible prefix plans.

Similarly, a plan is said to be a **suffix plan** if all of the steps of the plan except t_0 are contiguous to each other. Any feasible suffix plan will have the property that the result of regressing the goal state through the plan suffix is a feasible state. A suffix plan for $On(B, C)$ is P_3: $t_0 \prec Puton(B, C) * t_\infty$ shown in Figure 3. Suffix plans are produced by planners that do backward search in the space of states.

Planners that search in the space of plans typically generate plans in which actions are ordered only by precedence ("$\prec$") relations. Since any arbitrary number of new steps can come in between two steps ordered by a precedence relation, we shall call such plans "**elastic plans**."[4] An elastic plan for the subgoal $On(B, C)$ in Sussman anomaly is P_4: $t_0 \prec Puton(B, C) \prec t_\infty$.

A plan P for a subgoal g is called a **protected plan** if it has IPC constraints to protect the subgoal as well as every precondition of every step of P. Protection of a condition c at a step s is done by adding an IPC ($s' \overset{c}{-} s$) where s' is a step in P which gives the condition c to s. For example, a protected prefix plan for $On(B, C)$ in Sussman anomaly is P_5 : $t_0 * Puton(B, C) \prec t_\infty$ with the IPCs $(Puton(B, C) \overset{On(B,C)}{-} t_\infty)$, $(t_0 \overset{clear(B)}{-} Puton(B, C))$, and $(t_0 \overset{Clear(C)}{-} Puton(B, C))$. (Note that since the partial plan is a prefix plan, no new steps can come before $Puton(B, C)$ in the plan. Thus, the last two IPCs are redundant, as they can never be violated by new steps.)

Finally, it is useful to distinguish another kind of plan that we call a **blocked plan**. A blocked plan P contains blocked subplans for each of its individual subgoals. A blocked sub-

[4] We use the term elastic rather than "partially ordered" since the latter has been used to refer to protected elastic plans also (see below).

plan for a goal g_1 contains contiguity constraints that set the absolute distance between every pair of steps in P except the t_0 and t_∞ steps. As an example, a blocked plan for the subgoal $On(A, B)$ in Sussman Anomaly problem will be $t_0 \prec Puton(C, Table) * Puton(A, B) \prec t_\infty$.

3.4 Candidate Sets and Least Commitment

Note that all the plans shown in Figure 3 solve the subgoal $On(B, C)$ in the sense that if $On(B, C)$ were the only goal, then a solution extractor looking through minimal candidates will be able to terminate on each of them. They all have different candidate sets however. As we shall discuss at length in Section 8, the candidate set of the plan determines whether or not the plan can be refined to cover other subgoals. For example, the plan P_3 can be extended to cover the subgoal $On(B, C)$ only if its candidate set contains action sequences that achieve both $On(A, B)$ and $On(B, C)$ (which it doesn't in the current case – since no solution can end with the action $Puton(B, C)$).

The larger the candidate set of a plan, the higher the chance that it will contain an action sequence that also covers another subgoal. The size of the candidate set of a plan is determined by how many constraints it has and what type of constraints it has. Given two plans P and P' such that the constraints in P are a proper subset of the constraints in P', then clearly $\langle\langle P' \rangle\rangle \subseteq \langle\langle P \rangle\rangle$[5] (since some of the candidates of P may not satisfy the additional constraints of P'). Thus, in Figure 3, $\langle\langle P_2 \rangle\rangle \subseteq \langle\langle P_1 \rangle\rangle$ and $\langle\langle P_5 \rangle\rangle \subseteq \langle\langle P_4 \rangle\rangle$

Another factor affecting the size of the candidate set is the "type" of constraints that are present in the plan. For example, consider the following two plans: $(i) t_0 * t_1 : o_1 \prec t_\infty$, $(ii) t_0 \prec t_1 : o_1 \prec t_\infty$. The first plan has in general a smaller candidate set than the second since the contiguity constraint is a stronger ordering constraint than the precedence constraint (in particular $t_1 * t_2$ *implies* $t_1 \prec t_2$ but not vice versa). Thus, in Figure 3, we have $\langle\langle P_1 \rangle\rangle \subseteq \langle\langle P_4 \rangle\rangle$ and $\langle\langle P_3 \rangle\rangle \subseteq \langle\langle P_4 \rangle\rangle$.

The commonly used phrase "least commitment" is best understood in terms of candidate sets – a *least committed partial plan has a larger candidate set*. Among the plans shown in Figure 3 P_4 is the least committed and P_2 is the most committed.

The oft-repeated statement that partial order planners are less committed than state space planners actually just means that the partial plans produced by planners using plan space refinement have larger candidate set sizes than those using state space refinements. This makes sense because, as we will see in the next section, the state space refinements post contiguity constraints and the plan-space refinements post precedence constraints.

For example, in Figure 3, the plans P_1 and P_2 will be produced by forward state space refinements, P_3 will be produced by backward state space refinements, and P_4 will be produced by plan space refinements. P_5 will be produced by a combination of PS and FSS strategies.

3.4.1 Quantifying least commitment

Looking at least commitment in terms of candidate sets allows us to quantify the relative commitment of two partial plans. In this section, we will illustrate this by seeing how partial

[5] We avoid making statements about the actual sizes of the candidate sets since they can both be infinite

plans with precedence relations (generated by PS refinements) compare to partial plans with contiguity relations (generated by FSS and BSS refinements). Since partial plans can have infinitely large candidate sets, we will calculate the candidates of length L or smaller, and take the limit of the ratio of the candidate set sizes as L tends to ∞.

Consider the consider the following two plans: $P_1' : t_0 * t_1 : o_1 \prec t_\infty$, $P_2' : t_0 \prec o_1 \prec t_\infty$. Suppose there are a total of a distinct actions in the domain. Let us enumerate the candidates of size i for both the plans. For P_1', the candidates consist of all action sequences which start with o_1. Since we are looking for candidates of size i, this leaves $i-1$ positions in the sequence which can be filled by any of the a actions, giving rise to $a^{(i-1)}$ i-length candidates. For P_2', we are looking for action sequences that contain at least *one* instance of o_1 any where in the sequence. There are $(a-1)^i$ i-length action sequences that contain no instances of operator o_1. Thus, the number of sequences that contain at least one instance of o_1 are $a^i - (a-1)^i$.

Now, the number of candidates of size L or smaller for P_1' is

$$\phi_1^i = \sum_{i=1}^{L} a^{i-1} = \frac{a^L - 1}{a - 1}$$

those for P_2' are

$$\phi_2^i = \sum_{i=1}^{L} \left(a^i - (a-1)^i\right) = \frac{(a^{L+1} - 1)}{(a-1)} - \frac{(a-1)^{L+1} - 1}{(a-2)}$$

We want $\lim_{L\to\infty} \frac{\phi_2^i}{\phi_1^i}$. Since, typically, the number actions in a realistic domain a is $\gg 1$, we have, $\frac{a-1}{a} < 1$, $(a-1) \approx (a-2)$, $(a^{L+1} - 1) \approx a^{L+1}$ and the limit becomes,

$$\lim_{L\to\infty} \frac{\phi_2^i}{\phi_1^i} \approx \lim_{L\to\infty} a \left(1 - \left(\frac{a-1}{a}\right)^L\right) = a$$

What this means is that asymptotically, partial plans produced by planners using plan space refinements have candidate sets that are a times larger than those produced by planners using state space refinements. Similar computations can be done for partial plans belonging to other plan classes (e.g. P_4 and P_5 in Figure 3).

4 The UCP planning algorithm

The UCP algorithm uses the plan representation developed in the previous section to combine plan-space and state-space approaches under one framework. Figure 4 shows the top level control strategy of UCP. The algorithm starts by checking to see if a solution can be extracted from the given plan. In step 1, UCP chooses among three different refinement strategies, corresponding, respectively, to the forward state-space planning, backward state-space planning and plan-space planning approaches. We shall refer to them as FSS , BSS and PS refinements respectively. Since all three refinements are complete, the choice here is *not* non-deterministic

Algorithm UCP($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */

Parameters: `sol`: the solution extraction function
`pick-refinement`: a strategy for picking refinements

0. Termination Check: If `sol`($\mathcal{P}$) returns a ground operator sequence that `solves` the problem, return it and terminate.

1. Progressive Refinement: Using `pick-refinement` strategy, select any one of (*not* a backtrack point):

- Refine-plan-forward-state-space($\mathcal{P}$)
- Refine-plan-backward-state-space($\mathcal{P}$)
- Refine-plan-plan-space($\mathcal{P}$) {Corresponds to a class of refinements }

Nondeterministically select one of the returned plans. Let this be $\mathcal{P}'$.

2. (optional) Non-progressive (Tractability Refinements) Select zero or more of:

- Refine-plan-conflict-resolve($\mathcal{P}'$)
- Refine-plan-pre-ordering($\mathcal{P}'$)
- Refine-plan-pre-positioning($\mathcal{P}'$)

Nondeterministically select one of the returned plans. Let this be $\mathcal{P}'$.

3. (optional) Consistency Check: If the partial plan $\mathcal{P}'$ is inconsistent (i.e., has no safe ground linearizations), or non-minimal (e.g. has state loops) prune it.

4. Recursive Invocation: Call `UCP`($\mathcal{P}'$) (if $\mathcal{P}'$ is not pruned).

Figure 4: UCP: A generalized algorithm template for classical planning

Algorithm Refine-plan-Forward-State-space ($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */

1.1 Operator Selection: Nondeterministically select one of the following (choice point):

1. Nondeterministically select a step t_{old} from head-fringe of $\mathcal{P}$, such that all preconditions of the operator $\mathcal{ST}(t_{old})$ are satisfied in head-state of $\mathcal{P}$. (If t_{old} is the tail step, then select it only if the tail state is a subset of head-state.) **or**
2. Nondeterministically select an operator o from the operator library, such that all preconditions of the operator o are satisfied in head-state of the plan. Make a step name t_{new}, and add the mapping $[t_{new} \rightarrow o]$ to $\mathcal{ST}$.

1.2 Operator Application: Let t_{sel} be the step selected above. Add the auxiliary constraint $t_{\mathcal{H}} * t_{sel}$. (This implicitly updates the head step to be t_{sel}, and head state to be the result of applying $\mathcal{ST}(t_{sel})$ to head state).

Figure 5: Forward State Space Refinement

Algorithm Refine-plan-backward-state-space ($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */

1.1 Operator Selection: Nondeterministically select one of the following (choice point):

1. Nondeterministically select a step t_{old} from tail-fringe of the plan, such that (a) none of the effects of the operator $\mathcal{ST}(t_{old})$ are negating the facts in the tail state and (b) at least one effect of the operator $\mathcal{ST}(t_{old})$ is present in the tail state. (If t_{old} is the head step, then select it only if the tail state is a subset of head-state.) **or**
2. Nondeterministically select an operator o from the operator library, such that (a) none of the effects of the operator o are negating the facts in the tail state and (b) at least one effect of the operator $\mathcal{ST}(t_{old})$ is present in the tail state of $\mathcal{P}$. Make a step name t_{new}, and add the mapping $[t_{new} \rightarrow o]$ to $\mathcal{ST}$.

1.2 Operator Application: Let t_{sel} be the step selected above. Add the auxiliary constraint $t_{sel} * t_{\infty}$. (This implicitly updates the tail step and tail state).

Figure 6: Backward State Space Refinement

(i.e., does not have to be backtracked over). The refinements themselves are described in Figures 5, 6 and 7. UCP accepts an arbitrary control strategy `pick-refinement` as a parameter. In each iteration, this control strategy is used to select one of the three refinement strategies (see below). The selected refinement strategy is applied to the partial plan to generate the refinements. Since all three refinements are complete (see Section 5), *UCP never has to backtrack on the choice of the refinement strategy*. In step 2, UCP optionally applies one or more non-progressive refinements. These refinements essentially partition the candidate sets, without increasing their minimal candidate sizes. Step 3 checks to see if the plan is consistent (i.e., has non-empty candidate set), and step 4 invokes UCP on it recursively.

4.1 Progressive Refinements

4.1.1 State-space Refinements

Informally, the FSS refinement involves advancing the header state by applying an operator to it (which involves putting a contiguity constraint between the current head step and the new step corresponding to the operator that we want to apply). A step is considered to be applicable if all its preconditions are satisfied in the head state of the partial plan. Completeness is ensured by considering both the steps in the head fringe of the plan, and the operators in the operator library. One special case arises when the tail step of the plan is one of the steps on the head fringe. In this case, we can avoid generating partial plans that will not contain any solutions by requiring a stronger check. Specifically, we will require that tail step be applied to head state only when all the conditions in the tail state (rather than the preconditions of the tail step) are present in the head state. This is because, once the tail step is introduced into the header, no further steps can be added to the partial plan. (Of course, whether or not we use this stronger check will have no bearing on the completeness of the refinement).

BSS refinement (Figure 6) is very similar to the FSS refinement, except that it regresses the tail state by backward-applying new operators to the tail state. An operator is considered

Algorithm Refine-plan-plan-space ($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */
Parameters: `pick-open`: the routine for picking open conditions.

1.1 Goal Selection: Using the `pick-open` function, pick an open prerequisite $\langle C, t \rangle$ (where C is a precondition of step t) from $\mathcal{P}$ to work on. *Not a backtrack point.*

1.2. Goal Establishment: Non-deterministically select a new or existing establisher step t' for $\langle C, t \rangle$. Introduce enough constraints into the plan such that (i) t' will have an effect C, and (ii) C will persist until t. *Backtrack point; all establishers need to be considered.*

(Optional) [1.3. **Bookkeeping:**] Add interval preservation constraints noting the establishment decisions. Specifically do one of:

Interval Protection: Add the IPC $(t' \overset{C}{-} t)$.

Contributor Protection: Add two IPCs $(t' \overset{C}{-} t)$ and $(t' \overset{\neg C}{-} t)$.

Figure 7: Plan Space Refinement

Algorithm Refine-plan-pre-order ($\mathcal{P}$)

Use some given static ordering strategy, `pre-order`, to select a pair of unordered steps s_1 and s_2 and generate two refinements of P: $P + (s_1 \prec s_2)$ and $P + (s_1 \not\prec s_2)$ (note that the constraint $s_1 \not\prec s_2$ is equivalent to the constraint $(s_2 \prec s_1)$ since candidates are all completely ordered).

Algorithm Refine-plan-pre-position($\mathcal{P}$)

Use some given static pre-positioning strategy, `pre-position`, to to select a pair of steps s_1 and s_2 and generate two refinements of P: $P + s_1 * s_2$ and $P + s_1 \not* s_2$. In the special case where s_1 is always chosen to be the head step of the plan, the refinement is called the **lazy FSS** refinement and when s_2 is always chosen to be the tail step of the plan, it is called the **lazy BSS** refinement.

Algorithm Refine-plan-pre-satisfy($\mathcal{P}$)

Refine P such that some chosen auxiliary constraint C will hold in all the ground linearizations of P. If the chosen constraint is an IPC, we call the refinement the Conflict Resolution refinement, which is done as follows. For every IPC $(s_1 \overset{p}{-} s_2)$ in plan P, and every step s_t which has an effect that unifies with $\neg p$, generate refinements of P where s_t's effect cannot violate the IPC. The refinements consist of $P + (s_t \prec s_1) \vee s_2 \prec s_t)$ and $P + \pi^p_{s_t} @ s_t$, where $\pi^p_{s_t}$ denotes the preservation preconditions of s_t with respect to p [29]. The first refinement is typically further split into two refinements, called promotion and demotion refinements, to push the disjunction in the ordering constraints into the search space.

Figure 8: Non-progressive Tractability Refinements: Pre-ordering, Pre-positioning and Pre-satisfaction refinements

.

to be backward-applicable if it does not delete any conditions in the tail state, and adds at least one condition in the tail state (this latter part makes it goal-directed). The state resulting from the backward application of the operator o contains all the conditions of the previous state, minus the conditions added by o, plus the preconditions of o.

Making FSS Goal Directed: Although as stated, the FSS refinement is purely "data directed" and "BSS " refinement is purely "goal directed", the issue of goal/data direction is actually orthogonal to the issue of FSS /BSS refinements. The distinguishing properties of FSS and BSS are that one adds actions to the prefix and the other adds them to the suffix. For example, it is possible to focus both refinements by using state difference heuristics, which prefer the refinements where the set difference between the tail state and the head state is the smallest.

While the state difference heuristic works well enough for regression refinements, it does not provide sufficient focus to progression refinements. The problem is that in a realistic planning problem, there potentially may be many operators that are applicable in the current head state, and only a few of them may be relevant to the goals of the problem. Thus, the strategy of generating all the refinements and ranking them with respect to the state difference heuristic can be prohibitively expensive. We need a method of automatically zeroing on those operators which are possibly relevant to the goals.

One popular way of generating the list of relevant operators is to use *means-ends analysis*. The general idea is the following. Suppose we have an operator o whose postconditions match a goal of the problem. Clearly, o is a relevant operator. If the preconditions of o are satisfied in the head state of the current partial plan, we can apply it directly. Suppose they are not all satisfied. In such a case, we can consider the preconditions of o as subgoals, look for an operator o' whose postconditions match one of these subgoals, and check if it is applicable to the head state. This type of recursive analysis can be continued to find the set of relevant operators, and focus progression refinement. See [26] for an example of a FSS planner that uses this type of subgoaling trees to focus its attention.

4.1.2 Plan-space Refinements

Both FSS and BSS refinements attempt to guess the *relevance* (i.e., is this step going to be part of a solution for this problem) and *position* (i.e., when, during the execution of the solution plan will this step be executed). Often, our guess about the relevance of an operator to a problem is more informed than our guess about its position. For example, if one of our goals is to be in San Francisco (SF), we might guess that the action of "taking a flight to SF" would be part of our eventual solution plan. But, exactly when this action will occur in the solution plan depends on what other goals we are trying to achieve, and how they interact with this goal. This calls for a type of refinement that reasons about the relevance of the operator, without fixing its position. The plan-space refinement (PS refinement for short), does this. PS refinement strategy comes in several varieties, and a comprehensive discussion of these can be found in [19]. Here, we will summarize the main points.

The heart of the PS refinement, given in Figure 7, is an establishment operation [19]. In the establishment phase, a precondition p of a step s is selected, and a sufficient number of step, ordering and binding constraints are added to ensure that the point truth constraint $\langle p, s\rangle$ is satisfied (i.e., p will be necessarily true at s). This involves selecting a step s', either new or existing one, and ensuring that, $s' \prec s$, s' gives p, and that no steps that come between s' and s delete p. Additionally, if s' is a new step, its preconditions will have to be established later. See [19] for further details. The choice of which precondition to achieve does not have to be backtracked over, but all possible ways of establishing the selected precondition must be considered for completeness. Thus *PS refinement corresponds to a family of complete refinements*, each distinguished by the precondition that is considered for establishment.

While PS refinement allows us to defer the decision about the position at which an operator will occur in the final plan, the penalty we pay for this flexibility is that we do not have a complete picture of the world preceding and following the execution of an action. Because of this, a precondition that has been established once, might get deleted later when new steps are introduced between the contributor and the consumer step. PS refinements handle this by using an optional bookkeeping step where they keep track of intervals within the plan where certain conditions need to be protected. The optional bookkeeping step can add auxiliary (interval preservation) constraints to *protect* the establishment decisions. Specifically, when a precondition p of a step s is established using another step s', PS adds an IPC $(s' \overset{p}{-} s)$. Some planners, like SNLP [25] not only protect the condition p from being deleted, but also ensure that no other step intervening between s' and s will give p. This is to ensure that s' will be a unique contributor of p to s, which in turn ensures the systematicity of plan space refinement. This "contributor protection" can be done by simply posting two IPCs $(s' \overset{p}{-} s)$ and $(s' \overset{\neg p}{-} s)$.

4.2 Non-progressive (Tractability) refinements

It is easy to see that FSS , BSS and PS refinements are progressive refinements in that all of them can increase the length of the minimal candidates of the plan. We will now discuss a family of non-progressive refinements, shown in Figure 8. Unlike progressive refinements, which are motivated in terms of progressing towards a solution, non-progressive refinements by themselves do not take a refinement planner towards a solution. Non-progressive refinements are not required for termination if we have a solution extraction function that looks at all minimal candidates. However, they are motivated by the need to reduce the plan handling costs (specifically, the costs of solution extraction and consistency checking) [19]. Non-progressive refinements have thus been called "tractability refinements" [19] and "critics" [32].

There are essentially three types of non-progressive refinements used in refinement planning. The first two, called **pre-ordering** and **pre-positioning** refinements constrain the relative order and position of a pair of steps in mutually exclusive and exhaustive ways in the different branches. The third type, called **pre-satisfaction** refinement attempts to make an auxiliary constraint hold in all ground linearizations of the partial plan by adding additional constraints.

Specifically, the pre-ordering refinements fix the relative ordering between two steps s_1 and

s_2 by considering separately the case where $s_1 \prec s_2$ and the case where $s_1 \not\prec s_2$. Similarly, the pre-positioning refinements fix the relative position of the chosen pair of steps, t_1 and t_2 by considering separately the case where $t_1 * t_2$ and the case where $t_1 \not* t_2$. To ensure that the resulting plan is consistent, we must make sure that t_1 does not delete any preconditions of t_2 (if it did, then no candidate of the plan will be executable). Two special cases of pre-positioning refinements are of particular interest. When one of the pair of steps is either always the head step or always the tail step, the pre-positioning refinements extend the header and trailer sequences respectively, and thus lead to forward and backward state information. Although they provide state information, unlike FSS and BSS however, these refinements are not progressive– they do not consider adding steps into the plan, but merely order existing steps in the plan. Because of this, we call them the **lazy FSS** and **lazy BSS** refinements respectively. To ensure consistency of resulting plans, a lazy FSS is used only when the step being made contiguous to the head step has all its preconditions satisfied in the head state. In the case of lazy BSS , we make sure that the step being made contiguous to the tail step does not delete any condition in the tail state.

Pre-ordering and pre-positioning refinements reduce the plan handling costs by reducing the number of linearizations of the plan. The pre-satisfaction refinements reduce plan handling costs by ensuring that all ground linearizations of the partial plan are safe with respect to a given auxiliary constraint. An example of pre-satisfaction refinement is the **conflict resolution** refinement which ensures that a plan satisfies an IPC. Specifically, given an IPC $(s_1 \overset{p}{-} s_2)$ and a step s_t which has an effect that negates p, conflict resolution refinement considers either ordering s_t out of the range between s_1 and s_2 or adding "preservation preconditions" to s_t so that it will not negate p. The latter is called the *confrontation refinement*.[6] The former involves putting a disjunctive ordering constraint $(s_t \prec s_1) \vee (s_2 \prec s_t)$ on the plan. Traditionally, planners pushed this disjunction into the search space by generating two different plans, one in which $s_t \prec s_1$ is present and the other in which $s_2 \prec s_t$ is present.[7]

4.3 Termination Check

At each refinement cycle, the UCP algorithm checks to see if the search can be terminated successfully. As we mentioned earlier, the maximum amount of computation to be done in solution extraction involves checking if any of the minimal candidates of the partial plan are solutions. Since minimal candidates correspond to safe ground linearizations, of which there can be exponentially many, this will be an NP-hard computation. In fact, viewing the termination check this way allows us to consider it a sort of "scheduling" activity, and it is possible in theory to use the techniques used to solve constraint satisfaction problems. that are effective for scheduling problems, to do solution extraction. The termination check can thus be

[6]The so-called separation refinement [2] is subsumed under the confrontation option as one way of ensuring that s_t does not negate p is to add non-codesignation constraints that prevent the effect of s_t from unifying with p.

[7]This splitting does not lead to loss of completeness since the candidates are all ground operator sequences, and the only way the disjunctive ordering constraint can be satisfied is by satisfying one of the disjuncts.

done by any of the combinatoric search procedures used for supporting scheduling (e.g., SAT procedures).

Notice that this termination check is enough to successfully terminate UCP whether it uses only state space refinements, only plan-space refinements, or a combination of both. There may of course be more specialized realizations of the termination check that are more efficient for specific instantiations of UCP. For example, pure plan space planners using causal links can use a causal link based termination check used in SNLP [25, 19], which continue to refine a plan until all preconditions of all steps have been considered for establishment, and none of the IPCs are violated. Similarly, if we are using only the FSS and BSS refinements, then the plan can terminate as soon as the head step is introduced into the trailer, or vice versa.

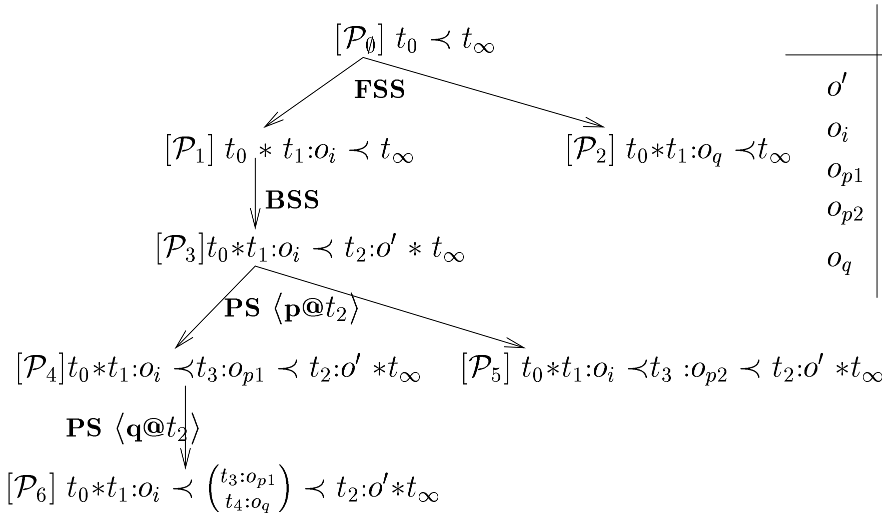
4.4 Consistency Check

At each refinement cycle, UCP uses an optional consistency check to prune out unpromising refinements. One important type of unpromising refinements are those partial plans which have no safe ground linearizations (and thus have empty candidate sets). In addition, in the presence of FSS and BSS refinements, we can also check for and prune plans containing *state looping*. Forward state looping occurs when there are two steps t_1 and t_2 in the header of the plan such that t_1 precedes t_2, and the state after t_1 contains all the conditions that are present in the state after t_2. Similarly, backward sate looping occurs when there are two steps t' and t'' in the trailer of the plan such that t' precedes t'' and the state preceding t' contains all the conditions that are present in the state preceding t''. In either case, it can be shown that the candidate sets of the corresponding partial plans will not contain any minimal[8] solutions, and the partial plans can thus be pruned.

4.5 An example planning trace of UCP

Figure 9 illustrates UCP's execution trace on a simple example. The problem has the initial state $\{i'\}$ and the goal state $\{G'\}$. The operators in the domain are described in the table on the right. The search starts with the null plan $\mathcal{P}_\emptyset$ in which t_0 precedes t_∞. Suppose the `pick-refinement` function suggests FSS refinement in the first iteration (such a suggestion may be based on the expected cost of the refinement; see below). There are only two library operators that are applicable to the header state (the tail fringe consists only of the tail step t_∞ which is not applicable). Two refinements, $\mathcal{P}_1$ and $\mathcal{P}_2$, each corresponding to the application of the respective operator to the header of $\mathcal{P}_\emptyset$, are produced. Next, suppose UCP picks $\mathcal{P}_1$ from the search queue. At this point, a BSS refinement strategy is chosen. This produces a single refinement $\mathcal{P}_3$, involving the application of the library operator o' to the trailer. Next, $\mathcal{P}_3$ is chosen from the search queue, and the PS refinement is selected, with the precondition $\langle p, t_2\rangle$ to be established. (To make the discussion simple, we assume a PS refinement that does not impose any optional bookkeeping constraints.) Two refinements, $\mathcal{P}_4$ and $\mathcal{P}_5$ result. UCP

[8] A solution is minimal, if and only if no ground operator sequence derived by deleting some elements from it is also a solution.



	Prec	add	del
o'	p,q	G$'$	-
o_i	i$'$	i1	-
o_{p1}	i1	p	-
o_{p2}	i2	p	-
o_q	-	q	p

Figure 9: An example illustrating the refinement process of UCP. The domain description is provided in the table on the right. The problem is specified with the initial state $\{i'\}$ and the goal state $\{G'\}$. The partial plans are shown in terms of their steps, ordering and contiguity constraints.

chooses $\mathcal{P}_4$ from the search queue and refines it further with another PS refinement with $\langle q, t_2 \rangle$ as the precondition to be established. This results in a single refinement, $\mathcal{P}_6$ which satisfies the termination test (it has a single safe ground linearization which is a solution for the problem). The search ends successfully when $\mathcal{P}_6$ is picked from the search queue in the next iteration (note that in $\mathcal{P}_6$, the steps t_3 and t_4 are unordered with respect to each other).

In the current example we did not have to use any non-progressive refinements. Suppose however that we had a less powerful solution extraction function of the type used in planners such as SNLP, which wait until all the ground linearizations are safe, and the corresponding candidates are solutions (this is not true in $\mathcal{P}_6$ since the linearization in which t_3 comes before t_4 will be deleting the precondition p of step t_2). In this case, we could have used[9] non-progressive refinements (e.g., pre-ordering, or if IPCs were being used by the PS refinement, conflict resolution), until we end up with two refinements of $\mathcal{P}_6$ one in which t_3 precedes t_4 and the other in which it follows t_4. The latter will then be a plan on which we can terminate.

The foregoing reinforces the point that nonprogressive refinements are needed only when one is using specialized termination criterion (of possibly lower complexity).

[9]It is also possible to use FSS and BSS refinements until the steps becomes contiguous.

5 Coverage, Completeness and Systematicity of UCP

5.1 Coverage

It is easy to see that by choosing the `pick-refinement` function appropriately, we can model the pure state space planners, as well as the pure plan-space planners as instantiations of UCP. In particular, in [19, 17, 18], we showed that the plan space refinement template, given in Figure 7 covers the complete spectrum of plan space planners, including UA, TO [28], SNLP [25], and TWEAK [6]. What is more interesting, as shown in Section 4.5, is the fact that the UCP algorithm also allows hybrid planners that can opportunistically apply plan-space as well as state-space refinements within a single planning episode. For example, the classical means-ends analysis planners such as STRIPS [9], or their descendents such as PRODIGY [10] can be modeled by a `pick-refinement` strategy such as the following: *If there is a step in the head-fringe of the plan that is applicable to the head-state, pick FSS or lazy FSS . Else, pick PS refinement.* Finally, it is also possible to use a more ambitious `pick-refinement` strategy: *pick the refinement that has the least expected cost* (see below) [13].

5.2 Completeness

To make matters simple, we will consider the completeness of UCP instantiations which use a solution extraction function that looks through the minimal candidates for a solution for the problem. All these instantiations will be complete (i.e., they will find solutions for all solvable problems) as long as all the refinements used by the instantiation are individually complete (i.e., they do not lose a solution candidate in the candidate set of the plan they refine). Since all the non-progressive refinements partition the candidate set, they are by definition complete. The three progressive refinements may lose some of the the candidates, but they can all be shown to be complete in that they do not lose any potential solutions. Since all three refinements are individually complete, any instantiation of UCP that uses these refinements will also be complete (see Section 2). The completeness of UCP depends primarily on the completeness of the refinement strategies.[10] Recall that a refinement strategy $\mathcal{R}$ is complete as long as every solution belonging to the candidate set of a plan $\mathcal{P}$ is guaranteed to belong to the candidate set of at least one of the refinements produced by $\mathcal{R}$ from $\mathcal{P}$.

The completeness of PS , BSS and FSS refinements is well known, if they start with a null plan (as they would if we use instantiations of UCP that use a single refinement). Specifically, the FSS refinement considers all possible executable prefixes, and BSS refinement considers all possible suffixes that can end in a goal state. The completeness of PS refinement is somewhat more subtle, but follows from Pednault's work [30].

The next order of business is to convince ourselves that these refinements remain complete even when they are applied to an arbitrary plan in the UCP represenstation. In the case of PS

[10] Strictly speaking, it also does depends on the specific termination criterion used by UCP. However, this dependency can be ignored as long as UCP uses a termination criterion that effectively checks if a minimal candidate corresponds to a solution (see [19])

refinement, this follows readily, since the only constraint types that UCP plans have, and the plans used in pure plan-space planners do not [19] are the contiguity constraints. However, since contiguity constraints are strictly stronger than precedence constraints (in that the former imply the latter), the completeness results will hold for UCP plans too. (This is why the PS refinement in UCP is identical to the one given in [19]).

The completeness of FSS and BSS refinements on UCP plans is not as obvious since the plans used by pure FSS and BSS planners do not contain precedence or IPC constraints. We will sketch how FSS continues to be complete even when applied to an arbitrary plan. Similar arguments apply to BSS refinement. The main difference between the plans that FSS operates on in a pure forward state space planner, and those it operates on in UCP is that the former contain no steps outside the plan prefix, while the latter do. If FSS only considers growing the prefix of the plan by adding new operators from the library, then it may miss some solutions. For example, consider a simple 1-way rocket domain, where the goal is to get the rocket from earth to moon, and the domain contains a single action: $Fly(E, M)$. Suppose we used a PS refinement and came up with a partial plan $\mathcal{P} : t_0 \prec t_1 : Fly(E, M) \prec t_G$. Suppose at this point we apply the FSS refinement to $\mathcal{P}$. If we only consider growing prefix with operators from the library, we will get a single refinement $\mathcal{P}' : t_0 * t_2 : Fly(E, M) \prec t_1 : Fly(E, M) \prec t_G$. Unfortunately, the only solution for the problem, which is the action $Fly(E, M)$ is no longer a candidate of $\mathcal{P}'$ (it was a candidate of $\mathcal{P}$). This loss of completeness is averted in our generalized FSS since it considers growing prefix with the help of all steps on the head fringe also. Thus, we get a second refinement $\mathcal{P}'' : t_0 * t_1 : Fly(E, M) \prec t_G$, which does contain the solution. Since both $\mathcal{P}'$ and $\mathcal{P}''$ are produced as refinements of $\mathcal{P}$, completeness holds.

Perceptive readers may have noticed that in the rocket example the candidate set of one of the refinements is a proper subset of the other ($\langle\!\langle \mathcal{P}' \rangle\!\rangle \subset \langle\!\langle \mathcal{P}'' \rangle\!\rangle$). This might suggest that we could retain completeness even if we ignore library operators which have been considered as part of the head-fringe (thus, we would not have generated $\mathcal{P}'$). This is unfortunately not true – the candidate set of a plan generated by putting a step $t_i : o_i$ in the head-fringe into the header will not always be a superset of the candidate set of a plan generated by putting a new instance of o_i, $t_{new} : o_i$ (from library) into the header. Informally, this is because $t_i : o_i$ may already be taking part in other auxiliary constraints in the plan, which do not burden the new instance $t_{new} : o_i$. To see this, consider a scenario where we have two operators o_1 and o_2 such that o_1 has two effects p, r, and and o_2 has two effects q and $\neg p$. o_1 has no preconditions and o_2 has one precondition r. We are interested in achieving the goal $p \wedge q$, starting from an empty initial state.

Consider a partial plan $\mathcal{P}$ which has the steps and orderings $t_0 \prec t_1 : o_1 \prec t_\infty$, and an IPC $(t_1 \overset{p}{-} t_\infty)$ (presumably generated by a PS refinement using interval protection, and working on the subgoal $p@t_\infty$.) Suppose we apply the FSS to $\mathcal{P}$. We will get two plans:

$$\mathcal{P}_1 : (t_0 * t_1 : o_1 \prec t_\infty) + (t_1 \overset{p}{-} t_\infty)$$

and

$$\mathcal{P}_2 : (t_0 * t_{new} : o_1 \prec t_1 : o_1 \prec t_\infty) + (t_1 \overset{p}{-} t_\infty)$$

with the first generated by putting t_1 into the header, and the second generated by introducing a fresh instance of o_1 from library into the header. In this case, it is clear that $\langle\langle\mathcal{P}_2\rangle\rangle \not\subset \langle\langle\mathcal{P}_1\rangle\rangle$. In fact, it is easy to see that $\langle\langle\mathcal{P}_1\rangle\rangle$ does not contain *any* solutions (since a solution must have o_2 and we cannot introduce o_2 into $\mathcal{P}_1$ without violating the IPC. Only $\langle\langle\mathcal{P}_2\rangle\rangle$ contains solutions! Thus, if we did not generate $\mathcal{P}_2$, we would have lost completeness.

In summary, FSS refinement must generate plans corresponding to both existing instances as well as fresh instances of operators that are applicable in the head state. (This situation is similar to plan-space planners which must consider both existing and new instances of a step during establishment). Of course, this may mean that the plans generated by FSS refinement will have overlapping candidate sets (thus leading to loss of systematicity). In the next section, we will show that as long as the PS refinement uses contributor protections, systematicity will still be guaranteed.

5.3 Systematicity

It is possible to prove that *any instantiation of UCP that uses a systematic PS refinement is systematic, regardless of the way it chooses to interleave the FSS , BSS and PS refinements*. Recall that as pointed out by McAllester (see [25, 19]), PS refinement is systematic as long as the book-keeping step uses contributor protections (that is, whenever a precondition p of a step t is established through the effects of another step t', two IPCs $(t' \overset{p}{-} t)$ and $(t' \overset{\neg p}{-} t)$ are added to the list of auxiliary constraints of the plan [19]).

We sketch the proof of systematicity of any instantiation of UCP that uses a PS refinement with contributor protections. To make exposition simple, we will concentrate on the systematicity of FSS refinement both in isolation, and in the presence of other refinements. Similar arguments hold for the systematicity of BSS and PS refinements.

If UCP uses only FSS refinement, its head fringe will always consist only of the goal step t_∞, and thus the only way FSS refines the plan is by introducing operators from library into the header. Since each FSS refinement will have a different library operator instance added to the end of the header, and since the operators in the header of a partial plan will form the prefix of each candidate of that partial plan, the candidate sets of different FSS refinements will be disjoint.

The above argument about differing prefixes may not hold when FSS is being used in conjunction with BSS and PS since in such cases FSS refinements will involve transferring steps from both the library and the *head fringe* into the header. It is possible in such cases to have the same operator instance (say o) introduced into the header in more than one refinement– once from the head fringe and once from the library. We will now show that systematicity still holds as the candidates of different refinements differ in terms of the number of instances of the operator o. We need to consider two cases – one in which the instance of o on header fringe has been introduced by a previous PS refinement, and the other in which it is introduced by a previous BSS refinement.

Case 1 o introduced by BSS : The illustration on the left hand side of Figure 10 shows a scenario where the step $t_o : o$ on the head fringe has been introduced by BSS (and thus it is part of the trailer). Now consider two refinements $\mathcal{P}_1$ and $\mathcal{P}_2$ generated by FSS by introducing

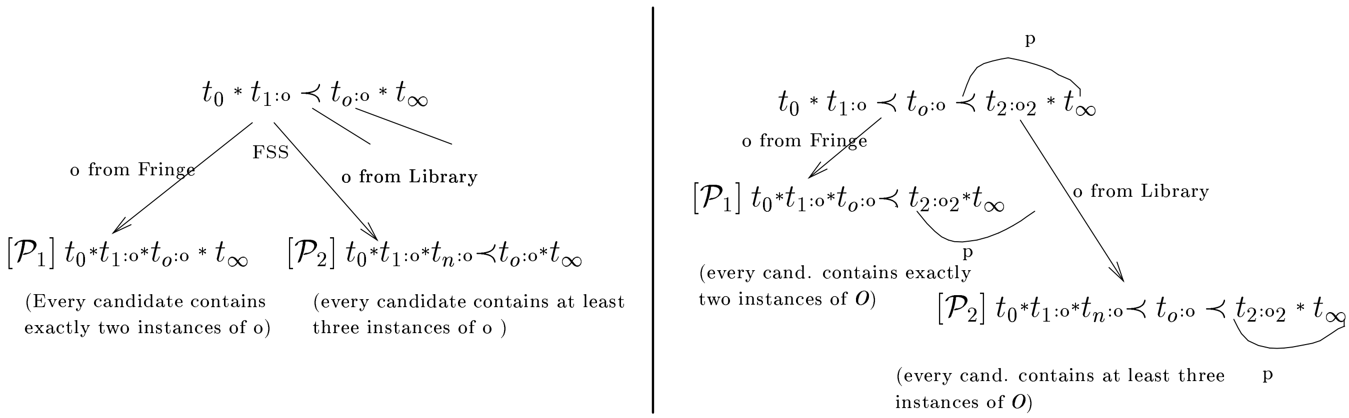


Figure 10: Examples illustrating the systematicity of UCP. The curved lines on the right correspond to IPCs on the plan.

$t_o : o$ from the fringe, and $t_n : o$ from library, respectively into the header. It is easy to see that the candidates of $\mathcal{P}_1$ will have exactly two instances of o, while candidates of $\mathcal{P}_2$ will have three or more instances of o.

Case 2 o **introduced by PS :** The illustration on the right hand side of Figure 10 shows a scenario where the step $t_o : o$ on the head fringe has been introduced by PS (using contributor protection, as per our premise). Suppose without loss of generality that $t_o : O$ was introduced to contribute some condition p to t_∞. This means that the partial plan contains two IPCs $(t_o \overset{p}{-} t_\infty)$ and $(t_o \overset{\neg p}{-} t_\infty)$. Now consider two refinements $\mathcal{P}_1$ and $\mathcal{P}_2$ generated by FSS by transferring an instance of o from the fringe, and from the library, respectively to the header. Once again the candidates of $\mathcal{P}_1$ and $\mathcal{P}_2$ will differ in terms of the number of instances of o. In the example plan shown in Figure 10, every candidate of $\mathcal{P}_1$ will have exactly two instances of o (no new instances o can come after the header since they will violate the IPC $(t_o \overset{\neg p}{-} t_\infty)$), whereas every candidate of $\mathcal{P}_2$ will at least have three instances of o. ■

6 Controlling UCP

In this section, we will consider the types of control strategies (heuristics, pruning techniques, selection strategies etc.) that are appropriate for UCP. In general, UCP requires heuristic guidance in selecting a partial plan to be refined next, and in picking the refinement strategy to apply to the selected partial plan. Additionally, if a plan-space refinement strategy is chosen, UCP needs guidance regarding which goal to select for establishment. The first is a backtrackable decision, while the latter two don't need to be backtracked.

For plan selection, in addition to the heuristics that are applicable to pure plan space and state space planners, UCP can also use hybrid heuristics tailored to its partial plan representation. For example, its plan selection heuristics could prefer FSS refinements that correspond to applying head fringe operators (rather than new operators from library) to the header; or evaluate the promise of the plan in terms of the set difference between the tail state and the head state. Further more, among the head-fringe steps, we can prefer those that are not taking part in any conflicts currently [1]. This is to ensure that there is no reason to suspect that the step will move out of head-fringe after conflict resolution.

The selection of refinement strategy can be done in many ways, and the tradeoffs offered by the various strategies is still an open question. In our preliminary studies (reported in the next section), we experimented with three hybrid strategies. The first, called UCP-MEA prefers FSS whenever a head fringe step is applicable to the head state, and PS otherwise. UCP-MEA has a generalized means-ends analysis flavor and thus simulates planners such as STRIPS and PRODIGY. The second, called UCP-MBA is similar to UCP-MEA, with the exception that before picking PS , it checks to see if a step on tail-fringe is applicable to the tail state, and if so, picks BSS . The third one, called UCP-LCFR, estimates the number of refinements generated by each of the three refinement strategies and selects the one that has the least number of refinements. This strategy is inspired by the least cost flaw refinement strategy, that was recently suggested in [13].

If a PS refinement is selected, UCP still faces the question of which goal to achieve first (each choice corresponds to a complete PS refinement with respect to that goal). In addition to the goal selection strategies used by pure plan space planners, UCP could also use the head state information. For example, it might prefer those goals that are not already true in the head state, or give preference to satisfying the preconditions of operators on the head fringe that have least number of unsatisfied preconditions.

7 Empirical studies on the utility of interleaving refinements

We have implemented the UCP algorithm on top of the Refine-Plan implementation described in [19]. Since UCP provides a framework to interleave the three different refinements within a single problem episode, we conducted several preliminary experiments to evaluate the advantages of such interleaving.

Experimental Setup: We considered six different instantiations of UCP. The first three, UCP-PS , UCP-FSS and UCP-BSS , always pick the same type of refinement, and correspond respectively to plan-space, forward state-space and backward state-space planners. The other three, UCP-MEA, UCP-MBA and UCP-LCFR correspond to the three hybrid refinement strategies described in the previous section.

Although the plan space refinement can have considerable variation [19] based on the protection and goal selection strategies used, in our experiments, we kept them constant. We used a simple LIFO strategy for selecting the open-condition to be established, contributor protections for bookkeeping, and conflict resolution for tractability refinements. This is equivalent to the refinement strategy used by SNLP [25, 19]. (In [19] we discuss the performance tradeoffs offered by the other ways of instantiating the plan-space refinement).

All the experiments used best-first search, with the ranking function defined as the sum of number of steps, open conditions, unsafe links, and the number of conditions of tail state

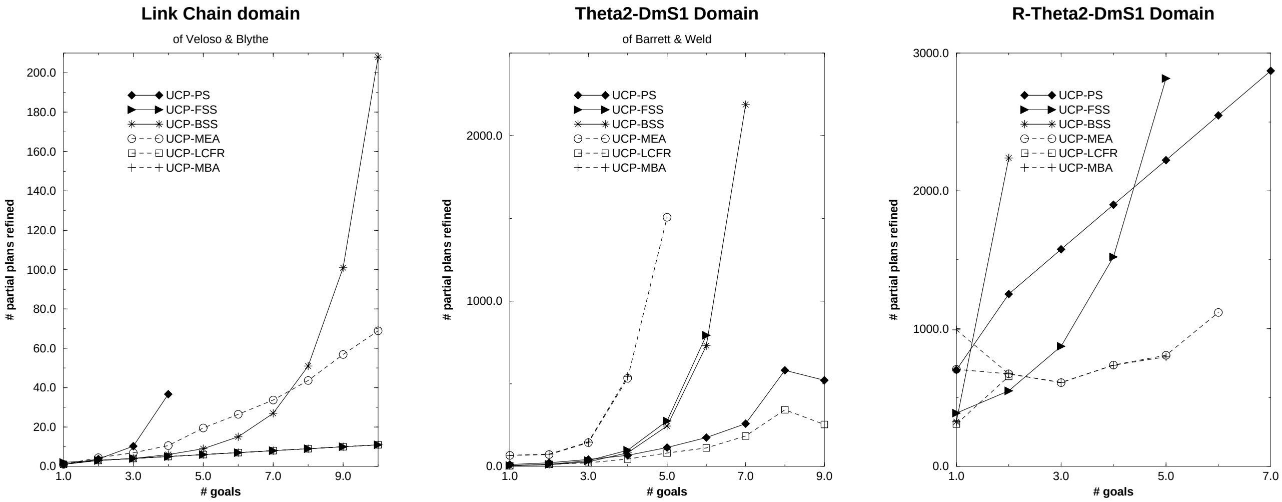


Figure 11: Plots illustrating the performance of various instantiations of UCP (measured in terms of the number of refinements made). Each point in the plot corresponds to the average over ten problems of a given number of goals. Missing points in a plot signify that the planner could not solve some of the ten problems within the allotted cpu time.

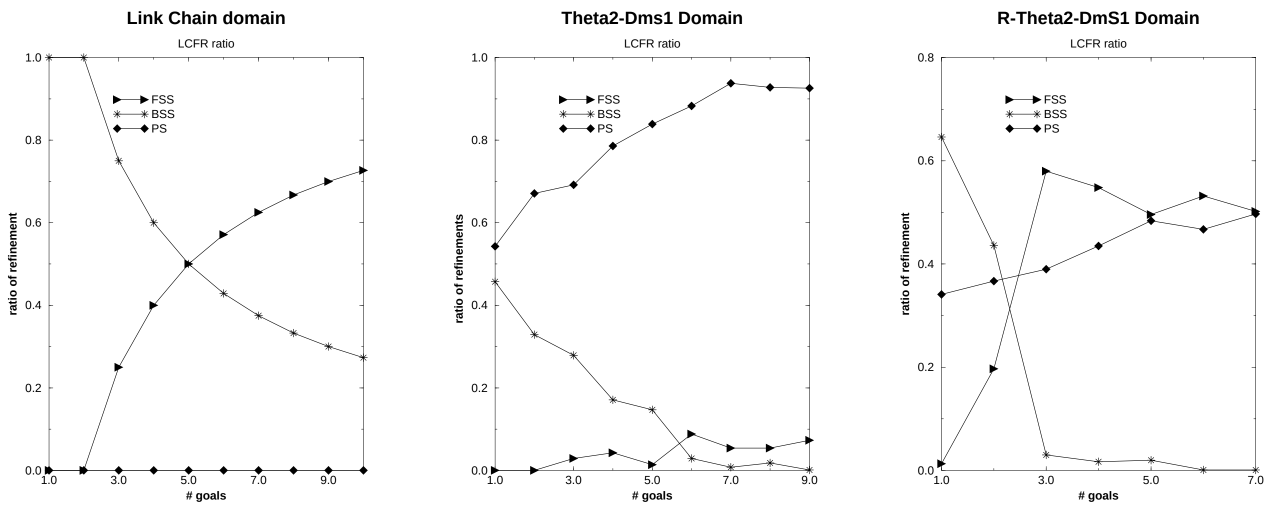


Figure 12: Plots showing the fractions of various refinements used by UCP-LCFR (Domain names shown on top of the plots)

not present in the head state. Additionally, partial plans that are inconsistent, or contain state looping (see Section 4) are pruned. Each planner was given a cpu time limit of 120 seconds for solving any problem. The time limit was increased to 300 seconds in the case of UCP-LCFR as our simple implementation estimates the branching factors of each refinement by actually simulating the refinement (other more efficient approximate estimation methods are of course possible; see [13]).

Domains and Results: We conducted experiments both in blocks world and in a variety of artificial domains designed by various researchers to illustrate the advantages of one planning approach over another. Our intent was to show that appropriate hybrid instantiations of UCP may do well in all such domains.

The first domain, called *link-chain* domain, was designed by Veloso & Blythe [35] to showcase the advantages of state space means-ends analysis planners over plan-space planners. This domain contains ten actions A_1 to A_{10}. Each action A_i requires the preconditions $G_1, G_2, \cdots G_{i-1}$, adds G_i, and $G_1, G_2, \cdots G_{i-2}$ and deletes G_{i-1}. The leftmost plot in Figure 11 shows the results of our experiments in this domain. We note that UCP-MBA and UCP-LCFR outperform all the other planners including UCP-MEA. (the plots of UCP-MBA, UCP-LCFR and UCP-FSS are all together as their performance was very close). The first plot in Figure 12 shows the fraction of times the three individual refinements were employed by UCP-LCFR during planning. We note that as the problem size increases the relative frequency of FSS increases with respect to BSS . This correlates well with the fact that UCP-LCFR tracks the performance of UCP-FSS in terms of number of partial plans refined, while UCP-BSS worsens its performance as the problem size increases.

The second domain, called $\theta_2 D^m S^1$, is one of the domains designed by Barrett & Weld [2] to demonstrate the advantages of plan space planning over state space planning. In this domain, each top level goal G_i can be achieved by either of two actions A_i^1 or A_i^2. All actions of type A_i^1 require P_α while all actions of type A_1^2 require P_β as a precondition. The initial state contains either P_α or P_β (but not both). The results of our experiments in this domain are shown in the second plot in Figure 11. Once again, we note that a hybrid instance of UCP, viz., UCP-LCFR, outperforms UCP-PS. An analysis of the pattern of refinements used by UCP-LCFR, shown in the second plot in Figure 12 reveals that it outperforms UCP-PS by opportunistically using BSS and FSS refinements in a small percentage of iterations. In other words, even a domain that motivates pure plan-space planning over pure state space planning [2], can benefit from a proper mix of the two types of refinements!

Finally, Figure 13 compares the performance of pure and hybrid instantiations of UCP in the standard blocks world domain (which, unlike the three artificial domains, is a non-propositional domain). The problems were generated using the random blocks world problem generator described in [27]. Each data point represents the average over 10 random problems containing a specified number of blocks. The data points in the graph correspond to situations where all 10 random problems were solved by the specific instantiation of UCP. UCP-BSS failed to achieve 100% solvability anywhere in this problem population, while UCP-FSS does quite well. Moreover, the performance of UCP-LCFR is close to that of UCP-FSS . The second plot shows the distribution of individual refinements used by UCP-LCFR. We note that UCP-LCFR achieves its performance by judiciously combining a small fraction of FSS and BSS refinements with a majority of PS refinements.

7.1 Discussion

Our results show that hybrid instances of UCP, that opportunistically interleave state space and plan space refinements, can do better than either the pure plan space or the pure state space

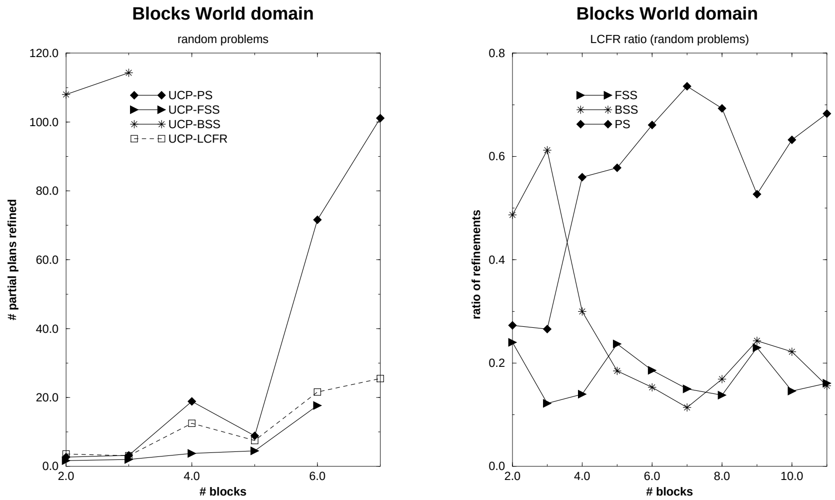


Figure 13: Performance of the instantiations of UCP in blocks world

refinements. They thus demonstrate the potential benefits of finding appropriate strategies for interleaving different refinements. Although our hybrid control strategies showed some promise, we believe that *the question of the right control strategy is still an open one*. In particular, even though UCP-LCFR did well in both *link-chain* and $\theta_2 D^m S^1$ domains, we found that it is by no means infallible. Our experiments with a variant of $\theta_2 D^m S^1$, called $R\theta_2 D^m S^1$, results of which are shown in the third plots in Figures 11 and 12 demonstrate that UCP-LCFR can be mislead under some circumstances[11] as the branching factor of a refinement is not a fool-proof indicator of its heuristic utility. We have also experimented with a variant of LCFR strategy that weights the branching factors of different refinements with factors that are proportional to their relative candidate set sizes. For example, if there are n_f FSS refinements and n_p PS refinements, we could weight n_p by a factor a (where a is the number of actions in the domain) (see Section 3.4.1). The results were not uniformly encouraging. In the next section, we explore a different track – that of using subgoal interactions to decide which instantiation of UCP is best suited to a given problem population.

8 Customizing UCP based on an analysis of subgoal interactions

Given that the instantiations of the UCP algorithm template represent a large variety of classical and hybrid planners, an important question is how one should choose among the different instantiations, in solving a set of problems. Discovering control strategies, such as LCFR, that automatically adapt the UCP algorithm template to a given problem, is one way of doing this.

[11] In this domain, the initial state contains either Q_α or Q_β. There are two sets of five actions each of which, when done in sequence will convert Q_α into P_α and Q_β into P_β. Finally, there are a set of dummy actions which add P_α or P_β but their preconditions are never satisfied.

Another, more macro-level, approach is to predict which of a variety of pre-specified instantiations of UCP will perform best on a given problem population. In this section, we will attempt to answer this question using an analysis of subgoal interactions in the problem population. To understand the idea of subgoal interactions, we start with the notion of a sub-plans. Consider a problem with initial state I and the goal state specified as a conjunction of subgoals $g_1 \wedge g_2$. A plan $\mathcal{P}$ is said to be a subplan for the goal g_1 if all the ground linearizations of $\mathcal{P}$ are safe and their corresponding ground operator sequences achieve g_1. Subgoal interactions arise when the subplan made for one subgoal is extended to also achieve another subgoal, or when the independently produced subplans for two subgoals are "merged" to create a plan that solves the conjunction of the two subgoals.

Although considerable work has been done on understanding subgoal interactions [24, 2, 35, 14], much of this has been done in terms of the specifics of state space [24] or plan-space planners [2]. It turns out that both subplan extension and merging could be understood cleanly in terms of the candidate set semantics of the partial plans developed in Section 3. In the following, we provide such an analysis. In Section 9, we show how this analysis can be used to predict the fit between an instantiation of UCP and a given problem population.

8.1 Candidate Set Based definitions of Subplan Interactions

Given two goals g_1 and g_2 to be achieved conjunctively from an initial state I, and specific subplans for achieving either of those goals, there are two scenarios in which the combinatorics of finding a plan achieving both goals is controlled:

1. We can find a plan P_1 for g_1 and a plan P_2 for g_2 independently, and then merge the plans together to produce a plan for g_1 and g_2. When subplans are mergeable this way, then we can parallelize the planning effort by working independently on the subgoals first and then working on merging the plans.

2. We can find a plan P_1 for g_1 that can be refined into a new plan for achieving both g_1 and g_2 without violating any commitments made in P_1. When serial extension is possible, we will essentially be able to work on the second subgoal without ever having to backtrack on (i.e., undoing) the refinements made in planning for the first subgoal.

We will capture these two notions from a candidate set perspective in the definitions of mergeability and serial extension below.

Definition 1 (Mergeability) *We say that a plan P_1 for achieving a goal g_1 from an initial state I is mergeable with respect to a plan P_2 for achieving goal g_2 from I, if there is a plan P' that achieves both g_1 and g_2 (from the same initial state), and $\langle\langle P' \rangle\rangle \subseteq \langle\langle P_1 \rangle\rangle \cap \langle\langle P_2 \rangle\rangle$. (Thus syntactically, P' contains all the constraints of P_1 and P_2).*

In addition, the plans are said to be **simple mergeable** *if every step in P' is present in either P_1 or P_2 (i.e., P' does not contain any new steps), and the number of steps in P' is the sum of number of steps in P_1 and P_2. Finally, the plans are said to be* **trivial mergeable** *if P' contains no more or less constraints (steps, orderings, bindings) than P_1 and P_2.*

In general, merging two plans involves either combining steps in the plans being merged, or adding steps that are not present in either of them. Simple mergeability essentially ensures that the plans can be merged without adding any new steps, or combining existing steps (thus bounding the amount of effort required in the merging phase). In contrast, the merging involving addition and combination of steps can be as costly as planning itself [37]. Even simple mergeability can lead to costly merging phases (as a possibly exponential number of combined linearizations of the two plans need to be considered). Trivial mergeability is the most restrictive as it requires that the merging operation only involve unioning the constraint sets of the two plans.

To illustrate these ideas, consider the blocks world situation with four blocks A, B, C and D all on table in the initial state. The plan $t_0 \prec Puton(A, B) \prec t_\infty$ for subgoal $On(A, B)$ is trivial mergeable with the plan $t_0 \prec Puton(C, D) \prec t_\infty$ for $On(C, D)$. In contrast, the plan $t_0 \prec Puton(A, B) \prec t_\infty$ for $On(A, B)$ is simple mergeable with the plan $t_0 \prec Puton(B, C) \prec t_\infty$ for $On(B, C)$ (but not trivial mergeable). The plan $t_0 * Puton(A, B) \prec t_\infty$ for $On(A, B)$ is not simple mergeable with the plan $t_0 \prec Puton(B, C) \prec t_\infty$ for $On(B, C)$, although it is mergeable. This is because the only way of merging these plans will be to insert additional steps giving rise to a plan such as $t_0 * Puton(A, B) \prec \underline{Puton(A, Table)} \prec Puton(B, C) \prec \underline{Puton(A, B)} \prec t_\infty$.

Finally, an example of mergeability that requires combining steps in the plans being merged is the "one-way rocket" problem which involves transporting objects A and B from the earth to the moon with the help of a single one-way rocket. The plan for taking object A to the moon will be $P_1 : t_0 \prec load(A) \prec Fly(Rocket) \prec unload(A) \prec t_\infty$, and the plan for taking object B to Moon will be $P_2 : t_0 \prec load(B) \prec Fly(Rocket) \prec unload(B) \prec t_\infty$, However, merging P_1 and P_2 to solve both the goals requires combining the two instances of $Fly(Rocket)$, since every complete plan can only have one instance of $Fly(Rocket)$.

Next, we will consider the idea of serial extension:

Definition 2 (Serial Extension) *We say that a plan P for achieving g_1 from a given initial state I (i.e., executing P from I will get us to a state where g_1 is true) is serially extensible with respect to a second goal g_2 if $\langle\langle P\rangle\rangle \cap L(g_1 \wedge g_2) \neq \emptyset$, where $L(g_1 \wedge g_2)$ is the set of all ground operator sequences that can achieve both g_1 and g_2 from the initial state.*

Any plan P' whose candidate set is a subset of $\langle\langle P\rangle\rangle \cap L(g_1 \wedge g_2)$ will achieve both g_1 and g_2. Since all candidates of P' are also candidates of P, P' has all the constraints of P (plus more). Thus, we never have to backtrack over any refinements that lead to P in coming up with P'.

Continuing the three block stacking example, the plan $t_0 \prec Puton(A, B) \prec t_\infty$ for $On(A, B)$ is serially extensible with respect to the goal $On(B, C)$, but the plan $t_0 \prec Puton(B, C) * t_\infty$ for $On(B, C)$ is not serially extensible with respect to the subgoal $On(A, B)$. To see the latter, note that no solution for Sussman anomaly can have $Puton(B, C)$ as the final step.

8.2 The Role of Planner vs. Partial Plans in subplan interactions

Perhaps surprisingly, our characterization of subgoal interactions shifts the attention from the type of planner to the nature of partial plans that are being merged or extended. The critical role played by a "planner" is in coming up with a plan for an initial subgoal. If the candidate set of that plan does contain a solution for the two goals together, then any refinement planner which uses only complete refinement strategies – be they forward state space, backward state space, plan-space or a combination thereof – will be able to extend the plan. This distinction may seem artificial given that most traditional planners use the same refinement strategy to generate the first subplan as well as to extend it. However, the distinction becomes more useful in the case of UCP, which can use multiple refinements.

For example, we noted that the plan $P_1 : t_0 \prec Puton(B,C) \prec t_\infty$ for subgoal $On(B,C)$ in the Sussman anomaly is serially extensible with respect to subgoal $On(A,B)$. This means that any complete refinement – including forward and backward state space refinement strategies – can extend this plan into a solution. To illustrate, here is a series of forward state space refinements that will convert P_1 into a solution. (i) Apply an instance of the operator $Puton(C,Table)$, to the head state giving P_2: $t_0 * Puton(C,Table) \prec Puton(B,C) \prec t_\infty$. (ii) Apply the $Puton(B,C)$ step to the head state of P_2 giving P_3: $t_0 * Puton(C,Table) * Puton(B,C) \prec t_\infty$ and finally (iii) Apply an instance of the step $Puton(A,B)$ to the head state of P_3 giving rise to a solution. It is *not* the case that we need a plan space refinement for this purpose.

8.3 Characterizing Subgoal interactions

Mergeability and serial extension are defined in terms of specific plans for individual subgoals. Since a particular subgoal may have a variety of plans, given two subgoals g_1 and g_2, some of the plans for g_1 may be mergeable with some of the plans of the second subgoal g_2, while some others may be serially extensible with respect to g_2. In order to exploit the computational advantages of mergeability and serial extension, we need to consider "all possible" plans of g_1 (and g_2). Since, as discussed in Section 3.3, there are in general a variety of partial plans and, depending on the refinements one uses, only a subset of these plans may actually be realized by a refinement planner, it makes more sense to qualify the claims with respect to a class of plans, as we do below:

Definition 3 (Parallelizability) *We will say that two subgoals g_1 and g_2 are parallelizable modulo a class of plans $\widehat{\mathcal{P}}$, if each plan $P_1 \in \widehat{\mathcal{P}}$ for achieving g_1 is mergeable with any plan $P_2 \in \mathcal{P}$ that achieves g_2.*

The subgoals g_1 and g_2 are said to be **simple parallelizable** *modulo the class of plans $\widehat{\mathcal{P}}$ if any plan of g_1 in $\widehat{\mathcal{P}}$ is simple mergeable with any plan of g_2 in $\widehat{\mathcal{P}}$ to give rise to a plan for g_1 and g_2. The subgoals are* **trivial parallelizable** *if the plans for the subgoals are trivially mergeable.*

The subgoals g_1 and g_2 are said to be **optimal parallelizable** *modulo the class of plans $\widehat{\mathcal{P}}$ if any optimal plan of g_1 from $\widehat{\mathcal{P}}$ is mergeable with any optimal plan of g_2 of $\widehat{\mathcal{P}}$ to give rise to an optimal plan for g_1 and g_2.*

From the complexity point of view, parallelizability allows us to use divide-and-conquer approaches for planning. If g_1 and g_2 are parallelizable, then the cost of solving the conjunctive goal is additive in the cost of solving the individual goals, plus the cost of merging the plans. However, parallelizability does not in itself imply that actually parallelizing the goals is either efficient (since the merging phase can be costly) or desirable (since the merged plan may be in optimal). For parallelization to be a win, the cost of merging should be small. The cost depends upon the type of merging (trivial, simple or non-simple). While trivial mergeability takes constant time, and simple mergeability can be NP-hard [37], merging involving the addition and deletion of actions can be as costly as planning itself.

Given any domain where all the actions are reversible, any pair of subgoals from that domain will be parallelizable (since we can always "undo" the actions of the individual plans for both the goals and then add actions to find a correct plan for the overall problem). Of course, this not only makes the merging step costlier than the original planning problem, but also leads to very inefficient plans. Thus, for parallelizability to be desirable, we need to have the guarantee that the divide and conquer approach will find "optimal plans" for the conjunctive goal by starting from optimal plans for the individual goals. This leads to the notion of optimal parallelizability. The optimality and efficiency restrictions on parallelizability can of course be combined. In fact, Korf's definition of subgoal independence [24], implies optimal and trivial parallelizability of all subgoals.

Definition 4 (Serializability) *Given a class $\widehat{\mathcal{P}}$ of plans, we will say that g_1 is* **serializable** *with respect to g_2 modulo $\widehat{\mathcal{P}}$ if every plan $P_1 \in \widehat{\mathcal{P}}$ of g_1 is serially extensible with respect to g_2.*

Serializability does not necessarily give rise to savings in planning effort. The main reason is that while parallelizability is a commutative relation, serializability is non-commutative. It is possible for g_1 to be serializable with respect to g_2 but for g_2 not to be serializable with respect to g_1. Thus for the planner to be able to exploit serializability, it needs to work on g_1 first and then on g_2. When there are multiple subgoals, the chance of picking the correct goal order is low and thus serializability does not imply improvements in planning cost. Following Barrett and Weld [2], we thus further extend the notion of serializability to consider *trivial* and *laborious* serializability.

Definition 5 (Serialization Order [2]) *Given a set of n subgoals $g_1, g_2 \ldots g_n$, a permutation π on these subgoals is considered a serialization order (modulo a class of plans $\widehat{\mathcal{P}}$), if every plan for achieving $\pi[1]$ can be serially extended to $\pi[2]$ and any resulting plan can be serially extended to $\pi[3]$ and so on.*

The set of subgoals are considered trivially serializable if all the permutations correspond to serialization orders, and are considered laboriously serializable if a significant number of permutations ($> \frac{1}{n}$) correspond to non-serialization orderings.

Relation Between Serializability and Parallelizability: Finally, it is instructive to note that while any form of parallelizability implies serializability, even trivial serializability does not guarantee any form of parallelizability. To see this, consider a simple domain with only two goals g_1 and g_2 and four operators defined below:

A trivially serializable but un-parallelizable domain			
Op	Prec	Add	Del
o_1	p	g_1	w
o_1'	r	g_1	q
o_2	w	g_2	p
o_2'	q	g_2	r

Suppose in a given problem, the initial state contains p, q, r and w and we want to achieve g_1 and g_2. It is easy to see that if g_1 is achieved by o_1 then we cannot achieve g_2 using o_2 and have to use o_2'. Thus not all plans of g_1 are mergeable with all plans of g_2. However, g_1 and g_2 are trivially serializable since any plan for g_1 or g_2 can be extended into a plan for both goals.

8.4 Coverage of the candidate-set based analysis of subgoal interactions

Readers familiar with previous efforts on characterization of subgoal interaction will note that our notions of serializability and parallelizability are defined modulo a class of plans. In this section, we will explain how our characterization subsumes the existing work by identifying the specific classes of plans over which the existing characterizations of subgoal interactions are implicitly based.

8.4.1 Korf's Subgoal Interactions

Korf [24] defines two subgoals to be serializable if there exists an ordering among the subgoals such that they can be planned for sequentially, such that once the first goal is achieved, the agent never passes through a state where the first goal is violated.[12]

The Sussman anomaly has non-serializable subgoals according to this definition. For example, suppose we work on $On(B, C)$ first and then $On(A, B)$. A state in which $On(B, C)$ is true is: $S\colon On(B, C) \wedge On(C, A)$. However, we cannot go from S to any goal state without violating $On(B, C)$.[13]

The following proposition shows that Korf's definition can be seen as a special case of our subgoal serializability for the class of protected prefix plans.

Proposition 1 (Korf's Serializability) *Two subgoals g_1 and g_2 are Korf-Serializable, if they are serializable with respect to the class of protected prefix plans.*

[12]Joslin and Roach [14] give a similar analysis of subgoal interactions in terms of the state space graph. In particular, they consider the state transition graph of the domain, and identify each subgoal with a subgraph of the transition graph (where all the states in that subgraph satisfy that goal). These subgraphs in general may contain several connected components. The set of subgoals is said to be nonlinear if any of the subgraphs corresponding to the subgoals have a connected component that does not contain a goal state. The idea is that if the planner finds itself in a component of the sub-graph of the goal, it cannot achieve the second goal without undoing the first.

[13]There is of course another state $On(B, C) \wedge On(C, Table) \wedge On(A, Table)$ from which we can reach a solution state without violating $On(B, C)$. However, serializability requires that this be true of *every* state that has $On(B, C)$ true.

The qualification about "protected plans" is needed as Korf requires that the goal g_1 remains achieved while P_1 is being extended. The "prefix plan" qualification is needed since Korf's analysis assumes that the search is being done in the space of world states and that the plan for the first goal takes us to a completely specified world state. Indeed, in Sussman anomaly, the plan $t_0 \prec Puton(B, C) \prec t_\infty$, with the IPC $(Puton(B, C) \stackrel{On(B,C)}{-} t_\infty)$ is serially extensible with respect to $On(B, C)$; although the plan $t_0 * Puton(B, C) \prec t_\infty$, with the same IPC is not.

8.4.2 Barrett and Weld's Serializability

Barett and Weld [2] extended Korf's [24] subgoal interaction analysis to plan space planners, and showed that problems like Sussman anomaly are serializable for the partial order planner SNLP. Although their analysis concentrated on specific planning algorithms, it can also be understood in terms of the class of plans with respect to which serializability is being defined. Specifically, we have:

Proposition 2 (Barrett and Weld's Serializability) *Two subgoals g_1 and g_2 are serializable by Barrett and Weld's definition if they are serializable with respect to the class of protected elastic plans.*

Since prefix plans have smaller candidate sets than elastic plans with the same set of steps, the latter naturally have a higher likelihood of being serially extensible with the second goal. Indeed, the Sussman anomaly problem is "serializable" for the class of elastic plans.

8.4.3 Relaxing the protected plan requirement

We saw that both Korf's and Barrett & Weld's notions of serializability implicitly involve protected plans, which post IPCs to protect the establishment of all the preconditions in the plan. Relaxing the requirement for protected plans leads to classes of problems that may not be serializable for the class of protected plans, but are serializable for the class of unprotected plans. This should not be surprising, given the discussion above, since everything else being equal, plans with IPCs have higher commitment (and thus smaller candidate sets) than plans without IPCs.

Note that the Sussman anomaly problem is indeed serializable for the class of un-protected prefix plans. In particular, the plan $t_0 * Puton(B, C) \prec t_\infty$ is serially extensible to the solution plan $t_0 * Puton(B, C) * Puton(B, Table) * Puton(C, Table) * Puton(B, C) * Puton(A, B) \prec t_\infty$ (which happens to be a non-minimal solution).[14]

We note that the protected plans made by a causal link planner such as SNLP are more constrained than the unprotected plans made by non-causal link planners such as TWEAK [6]

[14] On the other hand, the one way rocket problem is not serializable even without the protection restriction; the critical issue for this problem is the prefix plan requirement. Since by Korf's original definition, both these problems are non-serializable, and thus indistinguishable, we note that by considering serializability in terms of classes of plans, we can make finer-grained distinctions among different problems.

and UA [28], or planners that use disjunctive protections (e.g. multi-contributor causal links) such as MP and MP-I [16]. Thus, there may be domains with subgoals that are not serializable with respect to SNLP but are serializable with respect to these latter planners. ART-MD-RD, first described in [16], and shown below, is one such domain:

ART-MD-RD domain from [16]			
Op	Prec	Add	Del
A_i (i even)	I_i, he	G_i, hf	$\{I_j \mid j < i\} \cup \{he\}$
A_i (i odd)	I_i, hf	G_i, he	$\{I_j \mid j < i\} \cup \{hf\}$

To see this, consider a problem where the initial state contains $I_1, I_2, \ldots I_n$ and he and we want to achieve two subgoals g_1 and g_2. If we consider the class of protected elastic plans, we are forced to decide which step gives the condition he to the step A_2 achieving g_2, and since some of the possibilities, such as the initial state, eventually become infeasible in the presence of the second subgoal g_1, the problem will not be trivially serializable for this class. This difficulty goes away when we restrict our attention to the class of unprotected elastic plans.

In [35], Veloso and Blythe also provide a range of domains where protection commitments become the critical issues with respect to serializability. In particular, they compare SNLP with a state space means ends analysis planner which doesn't use any protection and conclude that in these domains, the protections can hurt the performance of SNLP.

Although Veloso and Blythe's observations have been made in the context of a state-space vs. plan-space planner comparison, they can be generalized by noting that the critical issue once again is the commitment inherent in the plan for the first subgoal, rather than the planner being used to extend it. In particular, similar performance tradeoffs would be observed if the planners being compared were both state-space or both plan-space planners (e.g., one being TWEAK [6] and the other being SNLP), as long as one planner uses protections and the other doesn't.

9 Factors Influencing the Selection of a Refinement Planner

Based on our characterization of subgoal interactions, a general method for selecting a particular refinement planner (instantiation of UCP) involves (a) delineating the class of plans with respect to which most of the goals of the domain are trivially serializable, and (b) finding a refinement planner that is capable of generating exactly that class of plans. However, "a" and "b" are not completely independent. Since the serializability and parallelizability notions are defined in terms of a given class of plans (see Sections 3.3 and 8.3), and we have not put any restrictions on the legal classes of plans, it is theoretically possible to find a sufficiently small class of plans with respect to which any given set of subgoals are trivially serializable. This is not useful in and of itself if there is no (domain-independent) refinement planner that can generate exactly that class of plans.

Consider, for example, the artificial domain shown below (originally described as the $D^*S^1C^2$ domain by Barrett and Weld [2]):

$D^*S^1C^1$ domain of Barrett and Weld			
Op	Prec	Add	Del
A_i^1	I_i	G_i	G^*
A_i^2	I_i	G_i	

All problems have the initial state where all I_i and G^* are true. Whenever the goals of the problem include both G_i ($i \leq n$) and G^*, the problems will not be trivially serializable for the class of elastic plans, since some of the elastic plans for the G_i subgoals will contain the steps A_i^1 which delete G^*. The problems will however be serializable for the subclass of elastic plans that do not contain any of the A_i^1 steps. Unfortunately, this latter fact is of little help in exploiting the subgoal interactions in the domain since elastic plans are generally generated by plan space refinements, and domain independent plan-space refinement will not be able to avoid generating the plans with A_i^1 steps. The domain is also trivially serializable with respect to the class of feasible suffix plans (see Section 3.3), which can be produced by BSS refinements. This leads to the following guideline:

Guideline 1 *Given a domain, we should try to select a subclass of plans with respect to which the subgoals are trivially serializable, such that there is a domain independent way of generating that subclass of plans.*

9.1 Least committed plans and Serializability

Often, we have more than one subclass of plans that satisfy the requirements of the guideline 1. One feature that affects the selection of subclasses in such cases is the commitment level inherent in the various subplan classes. The conventional wisdom is that least committed plans lead to more efficient planning, and we will evaluate this idea here.

Although given two plans with the same set of steps, the plan with less commitment is always more likely to be serially extensible with respect to another goal than a more committed one, this dominance does not directly translate to subgoal serializability, which is defined in terms of all plans of a specific class.[15] In particular, as we saw above, the domain $D^*S^1C^2$ is not trivially serializable for the class of elastic plans, but is trivially serializable for the class of feasible suffix plans (even though the latter are more constrained).

The intuition about least commitment being more conducive to efficient planning is true in general however, especially when the domain contains a set of goals, not all of which are serializable with respect to any one subclass of plans. To see this, consider the domain below which is similar to the domain $D^*S^1C^2$ except for the augmented delete lists of the actions.

Variant of $D^*S^1C^1$ domain			
Op	Prec	Add	Del
A_i^1	I_i	G_i	G^*, I_{i-1}
A_i^2	I_i	G_i	I_{i-1}

[15] If we are considering case-based planning, rather than generative planning, then generating and storing partial plans with fewer constraints is more likely to be a win, as they can be reused and extended in more novel situations; see [12].

These delete lists ensure that every solution for a problem involving the goals G_i and G_j ($i < j$) will have the action achieving G_i before the action achieving G_j. Now, in this situation, if the problem contains more than one of the G_i subgoals, then it will not be trivially serializable with respect to the class of suffix plans, whereas any problem that does not contain G^* will be trivially serializable for the class of elastic plans. If we have to pick a plan for the first subgoal without knowing what the next subgoal is going to be, we are still better off in general picking a less constrained partial plan. The empirical results of Barrett and Weld [2] in domains that are laboriously serializable for all their planners, do support this view to some extent.

Although less constrained plans are more likely to be serially extensible, more constrained plans do have their advantages. They are typically easier to "handle" in terms of consistency and terminations checks [19]. Given a domain containing subgoals that are trivially serializable for two classes of plans $\widehat{\mathcal{P}}_1$ and $\widehat{\mathcal{P}}_2$, it can be more efficient to do planning with the more constrained class of plans. The foregoing discussion can be summarized as follows:

Guideline 2 *If the set of subgoals in a domain are trivially serializable with respect to two different subclasses of plans, choose the subclass containing more constrained plans.*

Guideline 3 *If the set of subgoals in a domain are not trivially serializable with respect to any one subclass of plans, and no subclass is a clear winner in terms of percentage of problems that will be trivially serializable with respect to it, choose the subclass containing the less constrained plans.*

9.2 An Empirical Validation of the Guidelines

Although the guidelines 1-3 above are still very high level and need to be fleshed out further, they do extend our understanding about the selection of refinement planners. For example, they tell us that given a planning domain containing arbitrary subgoals, a reasonable way of improving average case performance would be to consider the most constrained class of subplans with respect to which the maximum number of subgoals are trivially serializable, and use a refinement planner, which only produces plans in this class. Given a domain, where the individual subgoals are all serializable by Korf's definition, but the subplans for these subgoals can contain many potential interactions, the set of goals will be trivially serializable for both the class of elastic plans and the class of blocked plans. However, the latter are more restrictive class of plans, and thus using a refinement strategy that produces them can lead to improved planning performance.

We tested this hypothesis in a variation of Barrett and Weld's D^1S^2 domain shown below:

Variant of D^1S^2 domain			
Op	Prec	Add	Del
A_i (i odd)	I_i	M_i, he	hf
A_i (i even)	I_i	M_i, hf	he
B_i (i odd)	M_i, he	G_i	he
B_i (i even)	M_i, hf	G_i	hf

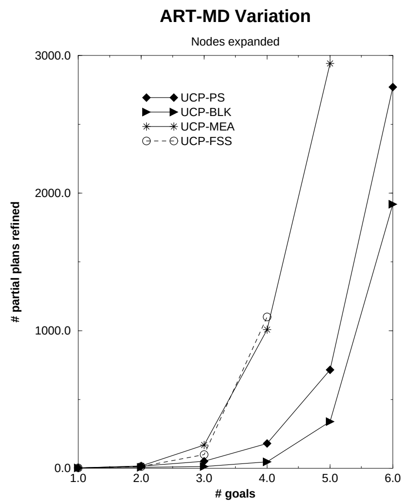


Figure 14: Results showing that the right class of subplans for a given domain may have intermediate level of commitment

The domain contains a set of goals of the form g_i which can be achieved by actions B_i. Each B_i in turn needs the condition M_i given by action A_i. A_i also provides he or hf conditions to B_i, and B_i. Because he and hf conditions are deleted by many steps, the subplans for individual top-level goals will have many interactions, even though the overall plans are all serially extensible. We tested this domain on several instantiations of UCP [21] that generate different subclasses of plans, including prefix plans (UCP-FSS), protected elastic plans (UCP-PS) and blocked plans (UCP-BLK). The results are shown in the plot in Figure 14. Our results show that blocking of steps of a top-level goal in a serializable domain improves performance both over plan-space (PS) refinements alone or over state-space refinements alone. The former is because of the plan handling cost while the latter is because the domain is not trivially serializable for prefix or suffix plans.

The results are also interesting in that they put the role of "least commitment" in a sharper perspective. As suggested in our guidelines, least commitment is a secondary rather than a primary indicator of the fit between the planner and the problem population. Indeed both UCP-PS which produces least committed plans, and UCP-FSS which produces most committed plans, are out performed by UCP-BLK which produces plans that are more committed than those of UCP-PS and less committed than those of UCP-FSS.

10 Related Work

Earlier work on unifying classical planning approaches includes Rosenchien's work [31] on *bigression* planner, which combines a forward state space search and a backward state space search; and our own more recent work [19, 17, 18] unifying a variety of plan-space planning frameworks into one algorithm template. To our knowledge, this paper is the first to rationally place the plan-space and state-space refinements in one unifying framework.

The semantic model of refinement planning used in this paper is mostly similar to the one developed in our earlier work [19]. One key difference is the differentiation of *progressive* and *non-progressive* refinements that is introduced in this paper. In [19] the description of PS refinement is folded with the description of the non-progressive refinements, while in this paper they are separated. As the discussion in this paper should show, non-progressive refinements do not help the refinement planner in progressing towards termination. Their main purpose is to partition the candidate set to make the plan handling cheaper. As such any set of mutually exclusive, and exhaustive constraints can form the basis of a non-progressive refinement. In contrast, we only have three basic types of progressive refinements.

Fink and Veloso [10] describe the PRODIGY 4.0 planner which does an interesting combination of forward state space refinement combined with means-ends analysis. The partial plans maintained by PRODIGY 4.0 contain a header sequence, and many backward chaining subgoal-trees starting from the goal state (called the "tail"). During each iteration, the planner selects a partial plan from the search queue, and modifies it in one of two ways. The first consists of growing one of the subgoaling branches in the tail. The second modification consists of shifting a leaf-level step on one of the subgoaling branches of the tail, that is applicable in the world state at the end of header sequence, to the header part of the plan. Planning ends when the current header state contains the goal state, regardless of the state of the tail. In terms of UCP algorithm template, Prodigy 4.0 approach corresponds (loosely) to the instantiation that uses a lazy FSS whenever there is a step in the head fringe that is applicable in the head state, and a plan space refinement otherwise.

Finally, our candidate set based subgoal interaction analysis builds on existing work on subgoal interaction analysis [24, 2, 35]. A detailed discussion of the relations can be found in Section 8.4.

11 Conclusion and Future Work

In this paper, we presented a generalized algorithm template, called UCP, which allows plan-space and state-space refinements on a single partial plan representation. UCP provides a parameterized planner template whose completeness and systematicity are ensured by the corresponding properties of the individual refinements. It thus provides a framework for opportunistic combination of plan-space and state-space refinements within a single planning episode. Depending upon the control strategy used, instantiations of UCP correspond to pure state-space, pure plan-space as well as hybrid planners that facilitate strategies such as means-ends analysis. We have discussed the issues of coverage, completeness and systematicity of the instantiations of UCP. Apart from its significant pedagogical advantages, our unified framework also promises considerable algorithmic advantages. To begin with, our implementation of UCP provides a normalized substrate for comparing the various refinement strategies. Our experiments with this implementation also demonstrate the potential benefits of opportunistically interleaving the plan-space and state-space refinements.

We have used the UCP framework to take a fresh look at the question "which refinement planner is best suited for a given problem population." For us, the "which refinement planner" part of the question translates to "which instantiation of UCP." To predict the fit between an instantiation of UCP and the given problem population, we use the notion of subgoal interactions. This involved first generalizing the existing accounts of subgoal interactions. We then developed some guidelines for choosing among the instantiations of UCP in terms of the generalized subgoal interaction analysis. Finally, we provided some preliminary results to demonstrate the predictive power of our analysis.

Finally, we have made considerable progress towards integrating task reduction refinements into the UCP framework, so that they too can be interleaved with the other refinements. Specifically, we have completed the theoretical formulation, but the implementation is pending. Appendix A provides the details of our formalization of HTN planning within UCP. Our approach differs from that advocated by Erol et. al. [8], where HTN planning is seen as fundamentally different from STRIPS-action based planning. We take the view that HTN planning simply involves introducing non-primitive actions into the domain model. Accordingly, the general scheme used in integrating HTN approach into UCP framework is (a) to augment plan-space refinements to consider primitive as well as non-primitive actions in establishing conditions and (b) to add a new non-progressive refinement called "Pre-reduction" refinement that replaces a non-primitive refinement with plan-fragments that contain primitive and non-primitive actions with the help of user supplied reduction schemas. The augmented plan-space refinement, which we call the HPS refinement, considers both primitive and non-primitive actions during establishment. It is thus well-suited for planning even in domains that are partially hierarchicalized, and need both establishment and reduction refinements. In augmenting the plan-space refinement, care is taken to ensure that the new refinement remains systematic, and that it respects the user-intent inherent in the specification of the task reduction schemas. Our treatment clarifies several misconceptions surrounding HTN planning, including its completeness and the need for phantom reductions (which we argue are really an artifact of doing plan-space refinment in the presence of non-primitive tasks).

11.1 Future Work

An interesting research direction involves using the UCP framework to unify and explain the relation between some of the newer refinement planning algorithms such as Graphplan [5] and Descartes [15]. Although these planners look very different from the conventional refinement planners that are covered by UCP, we believe that it is possible to extend UCP in such a way that conventional techniques as well as the newer planners become a special case of the extended framework. We have initiated work in this direction. Specifically, in [23], we show that algorithms like Graphplan fit into the UCP framework quite squarely, once we realize that a partial plan representation can involve "disjunctive constraints." We show there that Graphplan can be seen as a planner that uses FSS refinement, but keeps the refinements of a plan together by disjoining them (rather than pushing the disjunction into the search space). Such disjunctive representations have traditionally been shunned with the assumption that handling disjunctive plans will be too costly. However, we show that by using constraint propagation techniques, we can reduce the plan handling costs, while sharply reducing the search space size.

Indeed, our current view is that disjunctive representations come by default, and that it is only a historical accident that all classical planners split the disjuction in the plan representation into the search space. Specifically, a (progressive) refinement is best seen as operating on a set of partial plans to give rise to a new set of partial plans, the union of whose candidate sets is closer to the set of all solutions than the candidate set of the original plan set. We can handle the partial plan sets directly (which leads to disjunctive plan representations), or consider different plans in the set in different search branches (which leads to splitting of the disjunction into the search space). The formal properties of soundness, completeness and systematicity will hold no matter which approach is taken. The efficiency tradeoff is guided by issues such as the ease of constraint propagation in the disjunctive representation. We are currently actively engaged in sharpening this understanding.

References

[1] J.A. Ambros-Ingerson and S. Steel. Integrating Planning, Execution and Monitoring. In *Proc. 7th AAAI*, 1988.

[2] A. Barrett and D. Weld. Partial Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence*, Vol. 67, No. 1, 1994.

[3] A. Barrett and D. Weld. Schema Parsing: Hierarchical Planning for Expressive Languages. In *Proc. AAAI-94*.

[4] A. Barrett. Frugal Hierarchical Task-Network Planning. Ph.D. Thesis. Department of Computer Science and Engineering. University of Washington. 1996.

[5] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proc. IJCAI-95*, 1995.

[6] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[7] S. Chien. Static and Completion analysis for planning knowledge base development and verification. In *Proc. AIPS-96*, 1996.

[8] K. Erol, J. Hendler, D.S. Nau and R. Tsuneto. A critical look at critics in HTN planning. In *Proc. IJCAI-95*, 1995.

[9] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Readings in Planning*. Morgan Kaufmann, 1990.

[10] E. Fink and M. Veloso. Formalizing the Prodigy Planning Algorithm. CMU CS Tech. Report, Fall 1994.

[11] M. Ginsberg. Approximate Planning. *Artificial Intelligence*, Special Issue on Planning, Scheduling and Control. 1995.

[12] L. Ihrig and S. Kambhampati. On the Relative Utility of Plan-space vs. State-space planning in a case-based framework ASU CSE TR 94-006; Dec 1994. (Submitted for publication)

[13] D. Joslin and M. Pollack. Least-cost flaw repair: A plan refinement strategy for partial order planning. *Proceedings of AAAI-94*, 1994.

[14] D. Joslin and J. Roach. A Theoretical Analysis of Conjunctive Goal Problems. Research Note, *Artificial Intelligence*, Vol. 41, 1989/90.

[15] D. Joslin and M. Pollack. Passive and active decision postponement in plan generation. In *Proc. 3rd European Workshop on Planning*, 1995.

[16] S. Kambhampati. Multi-Contributor Causal Structures for Planning: A Formalization and Evaluation. *Artificial Intelligence*, Vol. 69, 1994. pp. 235-278.

[17] S. Kambhampati. Refinement search as a unifying framework for analyzing planning algorithms. In *Proc. KR-94*, May 1994.

[18] S. Kambhampati. Design Tradeoffs in Partial Order (Plan Space) Planning. In *Proc. 2nd Intl. Conf. on AI Planning Systems (AIPS-94)*, June 1994.

[19] S. Kambhampati, C. Knoblock and Q. Yang. Planning as Refinement Search: A Unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence* special issue on Planning and Scheduling. Vol. 76. 1995.

[20] S. Kambhampati. A comparative analysis of partial-order planning and task reduction planning. ACM SIGART Bulletin, Special Section on Evaluating Plans, Planners and Planning agents, Vol. 6., No. 1, January, 1995.

[21] S. Kambhampati and B. Srivastava. Universal Classical Planner: An algorithm for unifying state space and plan space approaches. In New Trends in AI Planning: EWSP 95, IOS Press, 1995.

[22] S. Kambhampati, L. Ihrig and B. Srivastava. A Candidate Set based analysis of Subgoal Interactions in conjunctive goal planning In *Proc. AIPS-96*, 1996.

[23] S. Kambhampati and X. Yang. On the role of Disjunctive Representations and Constraint Propagation in Refinement Planning In *Proc. KR-96*, 1996. (In Press).

[24] R. Korf. Planning as Search: A Quantitative Approach. *Artificial Intelligence*, Vol. 33, 1987.

[25] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proc. 9th AAAI*, 1991.

[26] D. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proc. AIPS-96*, 1996.

[27] S. Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[28] S. Minton, J. Bresina and M. Drummond. Total Order and Partial Order Planning: a comparative analysis. Journal of Artificial Intelligence Research 2 (1994) 227-262.

[29] E.P.D. Pednault. Synthesizing Plans that contain actions with Context-Dependent Effects. *Computational Intelligence*, Vol. 4, 356-372 (1988).

[30] E.P.D. Pednault. Generalizing nonlinear planning to handle complex goals and actions with context dependent effects. In *Proc. IJCAI-91*, 1991.

[31] S. Rosenchien. Plan Synthesis: A logical perspective. *Proc. IJCAI-81*, 1981.

[32] E. Sacerdoti. The nonlinear nature of plans. In *Proc. IJCAI-75*, 1975.

[33] A. Tate. Generating Project Networks. In *Proceedings of IJCAI-77*, pages 888–893, Boston, MA, 1977.

[34] D. Wilkins. *Practical Planning*. Morgan Kaufmann (1988).

[35] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. *Proceedings of AIps-94*, 1994.

[36] D. Weld. Introduction to Partial Order Planning. *AI Magazine*, Vol. 15, No. 4, 1994.

[37] Q. Yang, D. Nau and J. Hendler. Merging separately generated plans with restricted interactions. Computational Intelligence, 8(2):648-676, February 1992

[38] E. Fink and Q. Yang. Planning with Primary Effects: Experiments and Analysis. In *Proc. IJCAI-95*, 1995.

[39] R.M. Young, M.E. Pollack and J.D. Moore. Decomposition and Causality in Partial-Order Planning. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, 1994.

A Extending UCP to include HTN Planning

In this section, we will show how the hierarchical task network planning (HTN) can be integrated into the UCP framework. The exercise also illuminates many little-understood properties of HTN refinements, and their relations to the FSS , BSS and PS refinements. The general scheme in modeling HTN refinements into UCP framework will be (a) to augment plan-space refinements to consider primitive as well as non-primitive actions and (b) to add a new non-progressive refinement called "Pre-reduction" refinement that replaces a non-primitive step with plan-fragments that contain primitive and non-primitive actions.

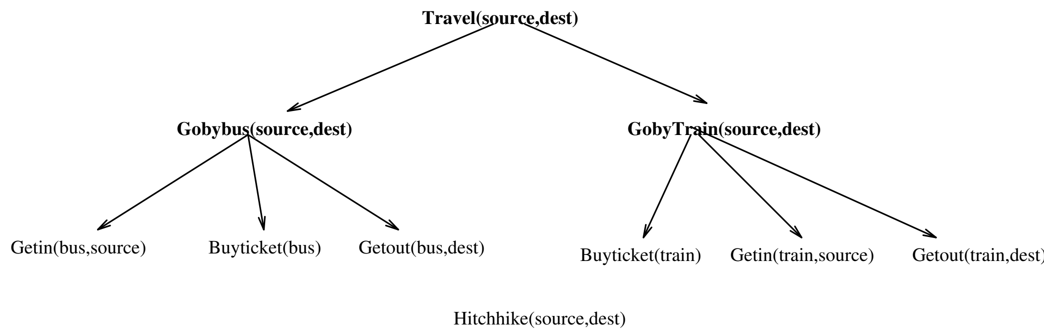


Figure 15: Hierarchy of actions in a sample travel domain

A.1 Motivation for Task Reduction Refinements

In both the state-space and plan-space refinements, the only knowledge that is available about the planning task is in terms of primitive actions (that can be executed by the underlying hardware), and their preconditions and postconditions. Often, one has more structured planning knowledge available in a domain. For example, in a travel planning domain (see Figure 15) we might have the knowledge that one can reach a destination by either "taking a bus" or by "taking a train". We may also know that "taking a train" in turn involves making a reservation, buying a ticket, getting into the train at the source, and getting off of it at the destination etc. In such a situation, we can consider "taking a train" as an abstract (non-primitive) action that cannot be directly executed by the hardware. This abstract action can then be reduced to a plan fragment consisting of other abstract or primitive actions (in this case "making a reservation", "buying a ticket", "going to the airport", "getting on the plane"). This way, if there are some high-level problems with the "taking flight" action and other goals, (e.g. there is not going to be enough money to take a flight as well paying the rent), we can resolve them *before* we work on low level details such as getting to the airport. This idea forms the basis for task reduction refinement.

Note that in order to facilitate task reduction refinements, we need to allow non-primitive actions into partial plans, and supply ways of reducing the non-primitive actions to plan fragments containing more primitive actions. Planners that use task reduction refinements are commonly called HTN planners. HTN planners such as SIPE [34] and NONLIN [33] have been used in many fielded applications, and it is commonly believed that HTN planners provide a more flexible basis for encoding and reasoning with "realistic" planning problems. Another aspect of task reduction planning is that it also provides a natural way of modeling planning problem at various levels of detail, and delegating the lower level details to specialized planners.

In [20] we examine these claims and conclude that the main advantage of task-reduction refinements are that they allow the user/designer a natural way of expressing knowledge about desirable ways of constructing partial plans. In particular, by specifying the task reduction schemas appropriately, the user can control the planner's access to the primitive actions of the domain. Through this flexibility the user is able to both bias the planner to efficiently generate desired plans, and also avoid generating undesirable plans. For example, in Figure 15, the user is able to disallow traveling by hitchhiking since the top level travel action can never be reduced into a plan containing hitchhiking action.

While inferring task reduction schemas given the description of domain in terms of primitive actions is a learning task and can thus be non-trivial, task reduction schemas may be readily available in domains where there exists human expertise (essentially the humans do the the task of inducing good task reduction schemas through experience and design). This view also implies that it is possible for task reduction refinements to co-exist with other types of refinements. In particular there exist many domains where human expertise exists but is incomplete. In encoding such domains, we have task reduction knowledge for parts of the domain, and the planner needs to rely on primitive actions for other parts. (The tricky thing here would be ensuring that we respect the user-intent as much as possible. See Section A.5.1).

In the rest of this section, we discuss how task reduction refinements can be modeled within UCP. Our formalization allows task reduction refinements to co-exist with other types of refinements. This is in contrast to most existing work that see HTN planners and other refinement planners as competing rather than complementary.

A.2 Extending plan representation to allow non-primitive tasks

To allow task reduction refinements within UCP, we need to extend our partial plan representation to allow for non-primitive steps (also called *tasks* in HTN parlance) in the partial plan. Specifically, the steps T in the partial plan $\langle T, O, \mathcal{B}, \mathcal{ST}, \mathcal{L}\rangle$, are mapped to two types of actions: *primitive actions* which correspond to the usual executable actions, and *non-primitive (abstract) actions*. The non-primitive actions have similar precondition/effect structure as the primitive actions, but an operator sequence containing non-primitive actions is not executable (and can thus not be a solution). Candidates of the plan will still be primitive operator sequences.

A partial plan is said to be *primitive* if it contains only primitive actions, and *non-primitive* otherwise.

The domain specification links each non-primitive action o to a set of reduction schemas. Each reduction schema $\mathcal{S}_i$ can be seen as a two tuple: $\langle \mathcal{P}_i, \mathcal{M}_i^{\mathcal{L}}\rangle$ where $\mathcal{P}_i$ is the partial plan fragment which can replace o, and $\mathcal{M}_i^{\mathcal{L}}$ maps the auxiliary constraints involving o into new auxiliary constraints involving steps of $\mathcal{P}_i$.

Consider for example the reduction schema for reducing a non-primitive action $Travel(Phx, SF)$ (see Figure 15):

$$\left\langle \begin{array}{c} \mathcal{P} : \left\langle \begin{array}{c} t_1 : Getin(bus, PHX) \prec t_2 : Buyticket(bus) \prec t_3 : Getout(bus, SF) \\ \mathcal{L} : \{(t_1 \overset{In(bus)}{-} t_3), (t_2 \overset{have(ticket)}{-} t_3)\} \end{array} \right\rangle \\ \mathcal{M}^{\mathcal{L}} : \{(? \overset{have(money)}{-} t_2), (t_3 \overset{at(SF)}{-} ?)\} \end{array} \right\rangle$$

This schema specifies that $Travel(Phx, SF)$ action can be replaced by the plan fragment containing three actions of getting into the bus, buying the ticket, and getting out of the bus at the destination. The plan fragment also contains IPCs specifying that the ticket should be kept throughout the journey and that the agent should stay in the bus throughout the journey. Finally, the mapping part of the schema states that any IPC incident on the $Travel(Phx, SF)$ action which preserves the condition $have(money)$ should be redirected to t_2 of the new plan fragment (since the identity of the source of the IPC is not known until reduction time, the schema denotes it by "?." Note that the plan fragment specified by the reduction method not only contains steps and orderings, but also auxiliary constraints, such as those corresponding to the IPCs, PTCs and contiguity constraints recommended in the reduction schema.

Two specializations of this general redirection strategy are widely used in the implemented planners. Both of them standardize the redirection so that the reduction schemas do not have to specify them. These strategies are:

Start-Finish redirection: Here it is assumed that the IPC $(t_1 \stackrel{p}{-} t_2)$ signifies that the condition p must be protected from the last primitive action of t_1 to the first primitive action of t_2. This can be achieved by incrementally translating the IPCs every time a reduction occurs. Specifically, the IPCs incident on the non-primitive action t are redirected to the start step t'_0 of the plan fragment $\mathcal{P}$ specified by the reduction schema $\mathcal{S}$, and the IPCs emanating from t are redirected to the finish step t'_∞ of $\mathcal{P}$. This is the most widely used strategy [33, 34].

Unique main action redirection: In this strategy [37], it is assumed that each reduction schema $\mathcal{P}'$ that can reduce a non-primitive action t names a unique main action $t'_u \in \mathcal{P}'$ that requires all the preconditions of t and gives all effects of t. Thus, the IPCs emanating as well as terminating at the non-primitive action t are redirected to t'_u.

Whether or not these standardized redirection schemes are appropriate for a set of reduction schemas is for the domain specialist to decide. Unless specified otherwise, in the remaining we will assume that the redirection is is specified by the reduction schemas.

A.2.1 Reduction of non-primitive actions

Armed with the specification of reduction schemas and non-primitive partial plans, in this section, we consider the details of how a reduction schema is used to reduce a non-primitive action.

Consider a plan $\mathcal{P} : \langle T, O, \mathcal{B}, \mathcal{ST}, \mathcal{L} \rangle$ containing an abstract action t. Let

$$\mathcal{S} : \langle \mathcal{P}' : \langle T', O', \mathcal{B}', \mathcal{ST}', \mathcal{L}' \rangle, \mathcal{M}^{\mathcal{L}} \rangle$$

be a reduction schema for t.

The partial plan that results from the reduction of the action t in $\mathcal{P}$ with the reduction schema $\mathcal{P}'$ is denoted by $Reduce(\mathcal{P}, t, \mathcal{S})$, and is given as follows:

$$Reduce(\mathcal{P}, t, \mathcal{S}) = \mathcal{P}_R : \left\langle \begin{array}{c} \{(T - t) \cup T'\}, \{(O - O_t) \cup O' \cup O_m\}, \\ \{\mathcal{B} \cup \mathcal{B}' \cup \mathcal{B}'_m\}, \{\mathcal{ST} \cup \mathcal{ST}'\}, \\ \{(\mathcal{L} - \mathcal{L}_t) \cup \mathcal{L}' \cup \mathcal{M}(\mathcal{L}_t)\} \end{array} \right\rangle$$

Where O_t and $\mathcal{L}_t$ are the ordering and auxiliary constraints involving t. These are replace by O_m and $\mathcal{M}(\mathcal{L}_t)$ during reduction. Notice that in replacing t with its reduction, we need to redirect any constraints that explicitly name t, to steps in its reduction. The redirection of IPCs is done in terms of the mapping $\mathcal{M}$ specified by the reduction schema. For the ordering constraints, the redirection is done automatically as follows. When t is reduced, the last step(s) of $\mathcal{P}'$ are e forced to precede all the steps t' such that $t \prec t'$ before the reduction. Similarly, the first step(s) of $\mathcal{P}'$ are forced to follow all steps t'' such that $t'' \prec t$ before the reduction. The redirection of contiguity constraints is similar to that of ordering constraints.

A.3 Candidate Set Semantics for non-primitive plans

As we shall see below, the presence of non-primitive actions necessitates extensions to the semantic model of partial plans .

We will start by providing semantics for the candidate set of a non-primitive partial plan. (The semantics developed in Section 3 hold only for primitive partial plans). We will do this in terms of the *concretizations* of a non-primitive partial plan. Informally, a concretization of a partial plan $\mathcal{P}$ is a primitive partial plan $\mathcal{P}'$ that is obtained by repeatedly reducing (replacing) the non-primitive actions in $\mathcal{P}$ with the help of the specified reduction schemas. Thus,

$$Concretizations(\mathcal{P}) = \{\mathcal{P}\} \qquad \text{if } \mathcal{P} \text{ is a primitive plan}$$

$$Concretizations(\mathcal{P}) = \bigcup_{\substack{\mathcal{P}'' = \mathit{Reduce}(\mathcal{P}, t, \mathcal{S}), t \in \mathcal{P}, \\ non-primitive(t), \\ \mathcal{S}\ is\ a\ reduction\ schema\ for\ t}} Concretizations(\mathcal{P}'')$$

Clearly, the concretizations of $\mathcal{P}$ are all concrete partial plans, whose candidate sets are well defined. We now define the candidate set of a non-primitive partial plan $\mathcal{P}$ in terms of its concretizations as:

$$\langle\!\langle \mathcal{P} \rangle\!\rangle = \bigcup_{\mathcal{P}'' \in concretizations(\mathcal{P})} \langle\!\langle \mathcal{P}'' \rangle\!\rangle$$

This definition means that the candidate set of a plan is empty if it has no concretizations. It thus handles the case of partial plans which contain abstract actions that cannot eventually be reduced to concrete plan fragments through repeated reductions.

We will also define the concretizations of a non-primitive action as the concretizations of a plan containing that action alone. That is

$$Concretizations(o_n) \triangleq Concretizations(\mathcal{P}_{o_n} : (t_0 \prec t : o_n \prec t_\infty))$$

Notice that the candidate set of a plan containing a non-primitive action depends on, and is constrained by, the ways of reducing non-primitive actions. This has consequences on the completeness of task reduction refinements and the consistency checks for non-primitive partial plans, as we shall see later.

A.3.1 Presence of non-primitive actions in the plan and Least Commitment

We know that the presence of a primitive step $t : o$ in a partial plan constraints its candidate set by disallowing all ground operator sequences that do not contain at least one instance of the operator corresponding to o. How does the presence of non-primitive steps in the plan affect its candidate set? The definition of the candidate set of non-primitive partial plans above answers this question. Specifically, the presence of a non-primitive step $t' : o'$ in a plan $\mathcal{P}$ constrains its candidate set by disallowing all ground operator sequences which do not contain all the operators corresponding to at least one concretization of o'.

This has some ramifications on the complexity of solution extraction based on inspection of minimal candidates. We have seen earlier that a primitive partial plan has at most an exponential number of minimal candidates. In the case of a non-primitive partial plan, the number of minimal candidates depends on the number of concretizations of the plan. If every non-primitive action has at most an exponential number of concretizations, then we will still have an exponential number of minimal candidates (exponential number of primitive plans with exponential number of minimal candidates each). The restriction of exponential number of concretizations will hold as long as the reduction schemas are "non-recursive" (in that a non-primitive action o can never be reduced into another non-primitive plan fragment containing o).

We can also explain the conventional wisdom that non-primitive partial plans are less committed than primitive partial plans. The presence of a non-primitive action only requires that at least one of its concretizations be present in each candidate, which is a weaker requirement than asking that a specific operator be present in all of them. To see this, consider a non-primitive action o, which can be reduced eventually into one of the single primitive actions $o_1, o_2 \cdots o_n$ (we are assuming that the reductions replace each action by a single sub-action, for simplicity). In this case, assuming that the number of primitive actions is a, the number of candidates of length L or smaller for the plan $\mathcal{P}_1 : t_0 \prec t : o \prec t_\infty$ will be:

$$\phi_3^i = \sum_{i=1}^{L} \left(a^i - (a-n)^i\right) = \frac{(a^{L+1}-1)}{(a-1)} - \frac{(a-n)^{L+1}-1}{(a-2)}$$

In contrast the number of candidates of length L or less for the primitive plan $\mathcal{P}_1 : t_\infty \prec t : o_1 \prec t_\infty$ will be (from Section 3.4.1):

$$\phi_2^i = \sum_{i=1}^{L} \left(a^i - (a-1)^i\right) = \frac{(a^{L+1}-1)}{(a-1)} - \frac{(a-1)^{L+1}-1}{(a-2)}$$

It is easy to see that ϕ_3^i is larger than ϕ_2^i since the quantity being subtracted is smaller in the former. Combined with the results of Section 3.4.1, this shows that plans produced by HTN planners are less committed than those of plans produced by plan space refinements which in turn are less committed than those produced by state space refinements.

A.3.2 Semantics of IPCs and Ordering constraints between non-primitive actions

We know the semantic import of ordering and IPC constraints between two primitive steps in a plan (see Section 3). The semantics of these constraints is not so clear if one of the steps involved is a non-primitive step. Their semantics are ultimately fixed by the way they are redirected during reduction.

Consider the case of ordering constraints (the case of contiguity constraints is similar). When the plan contains an ordering between two non-primitive actions $t_1 \prec t_2$, it is signifies that every primitive step resulting from the expansion of t_1 shall precede every primitive step resulting from the expansion of t_2.

An IPC $(t_1 \stackrel{p}{-} t_2)$ involving two non-primitive actions t_1 and t_2 means that p needs to be protected between *some* (not necessarily the last) primitive step in the eventual expansion of t_1 and *some other* (not necessarily the first) primitive step in the eventual expansion of t_2. As we have mentioned earlier, it is the responsibility of individual reduction schemas to specify how the IPCs incident and emanating from the reduced step are distributed.

A.4 Modeling Task Reduction as a refinement

We will model the HTN approach in terms of a non-progressive refinement called *Pre-Reduce*, which picks a non-primitive action and reduces it with the help of the reduction schemas. The reduction schemas prescribe plan fragments that will replace the non-primitive action. The pre-reduction refinement is shown in Figure 16. It takes a partial plan $\mathcal{P}$ containing non-primitive and primitive actions, picks a non-primitive action t corresponding to an abstract action o, and for each reduction schema $\mathcal{S}$ that can be used to reduce o, generates a refinement of $\mathcal{P}$, $Reduce(\mathcal{P}, t, \mathcal{S})$.

The pre-reduction is non-progressive, since by considering the different reductions of the non-primitive step separately, it in effect partitions the set of concretizations of the non-primitive step, which in turn leads to the partition of the candidate set of the plan containing that non-primitive step.

Given the definition of the candidate set of a partial plan P containing non-primitive actions, it is easy to see that pre-reduction refinement is complete in that it considers all possible ways of reducing the non-primitive action, and thus the set of concretizations of P is subsumed by the union of the sets of concretizations of the task reduction refinements.

A.5 Combining HTN refinement with other refinements

A.5.1 Extending PS refinement to consider non-primitive actions

The straightforward idea of introducing HTN approach into UCP would be to say that the Pre-Reduction refinement will be selected whenever the partial plan contains non-primitive actions. Non-primitive actions can enter the partial plan in two different ways. First off, the initial planning problem may be specified in terms of some non-primitive actions that need to be carried out (rather than just in terms of top level goals to be achieved). For example,

Algorithm Pre-reduce($\mathcal{P}$) **Parameters**: `pick-action`: the routine for picking non-primitive actions for reduction

1.1′. Action Selection: Using the `pick-action` function, pick an unreduced action $t \in T$ from $\mathcal{P}$ to work on. *Not a backtrack point.*

1.2′. Task Reduction: Non-deterministically select a reduction schema $S : \mathcal{P}'$ for reducing t. Replace t in $\mathcal{P}$ with $\mathcal{P}'$ (This involves removing t from $\mathcal{P}$, merging the step, binding, ordering, symbol table and auxiliary constraints fields of $\mathcal{P}'$ with those of $\mathcal{P}$, and modifying the ordering and auxiliary constraints in $\mathcal{P}$ which refer to t so that they refer to elements of $\mathcal{P}'$.

Backtrack point; all reduction possibilities need to be considered

Figure 16: Pre-reduction Refinement

the initial plan for planning a round trip from Phoenix to San Francisco may have two non-primitive actions $Travel(Phx, SFO)$ and $Travel(SFO, Phx)$. The second way is for a non-primitive action to be introduced into the plan as a part of establishing a goal.

For this to happen, the PS refinement which does establishment, must consider non-primitive actions. Writing such a hybrid establishment (we shall call it the hybrid plan space refinement or HPS refinement) poses several challenges. Consider the travel domain shown in Figure 15. Suppose our top level goal is $At(SF)$, and we have $At(Phx)$ in the initial state. The seven actions – $Getout(bus, SF)$, $Getout(train, SF)$, $Gobybus(Phx, SF)$, $GobyTrain(Phx, SF)$, and $hitchhike(car, SF)$ all are capable of achieving this condition. The question is which of these actions should HPS refinement consider in its different establishment branches?

We start by noting that if HPS refinement only considers the primitive actions, then it will already be a complete refinement. Thus, if we add the branches corresponding to non-primitive actions also, then we will have a redundant search space. As an example, if HPS refinement considered all seven actions as establishment possibilities in this example, then two of its refinements would be $P_1 : t_0 \prec Gobybus(Phx, SF) \prec t_\infty$ and $P_2 : t_0 \prec Getout(bus, SF) \prec t_\infty$. Since the non-primitive action $Gobybus(Phx, SF)$ ultimately gets reduced into a plan containing the primitive action $Getout(bus, NY)$, the plans have overlapping candidate sets (i.e. $\langle\!\langle P_1 \rangle\!\rangle \cap \langle\!\langle P_2 \rangle\!\rangle \neq \emptyset$), thus making HPS refinements non-systematic. At the same time, ensuring systematicity by considering only primitive actions during HPS refinement will violate the user's intent in providing the non-primitive actions and reduction schemas. In the travel domain example, by including the non-primitive actions, $Gobybus$ and $Gobytrain$, and providing their task reductions, the user is telling the planner that legal ways of traveling include going by bus and train (which involve buying tickets). This excludes solutions such as $hitchhike(Phx, SF)$ which are legal, but do not have any parse in terms of non-primitive actions.

Considering undominated actions to ensure systematicity: To avoid non-systematicity, and to preserve user-intent, we use the convention that a goal should be achieved by considering the dominance relations between actions. We consider an action t to be dominated by

Algorithm Refine-plan-hybrid-plan-space ($\mathcal{P}$) /*Returns refinements of $\mathcal{P}$ */
Parameters: `pick-open`: the routine for picking open conditions.
`pre-order`: the routine which adds orderings to the plan to make conflict resolution tractable.
`conflict-resolve`: the routine which resolves conflicts with auxiliary constraints.

1.1 Goal Selection: Using the `pick-open` function, pick an open prerequisite $\langle C, t\rangle$ (where C is a precondition of step t) from $\mathcal{P}$ to work on. *Not a backtrack point.*

1.2. Goal Establishment: Non-deterministically select a new or existing establisher step t' for $\langle C, t\rangle$ which is capable of giving C. If t' is a new step, make sure that it is maximally non-primitive. (i.e., *there is no non-primitive action t'' which can also give C and which can eventually be reduced into t'*) Introduce enough constraints into the plan such that (i) t' will have an effect C, and (ii) C will persist until t. *Backtrack point; all establishers need to be considered.*

1.3. Bookkeeping: (Optional) Add interval preservation constraints noting the establishment decisions, to ensure that these decisions are not violated by latter refinements. This in turn reduces the redundancy in the search space.

1.4. Phantom Establishment: In addition to the plans generated above, consider also the "phantom plan" $\mathcal{P}$ with the PTC $C@t$ (this will never again be picked for establishment).

Figure 17: Plan Space Refinement Modified to work in the presence of non-primitive actions

another action t', if there exists a reduction of t' that contains t. In the travel domain shown in Figure 15, the action $Gobybus(source, dest)$ dominates $Getout(bus, dest)$ and is dominated by $Travel(source, dest)$. We can ensure that the plans generated by HPS refinement have non-overlapping candidate sets by requiring that neither of actions considered during the establishment of a given goal are dominated by the other actions being considered.

Considering maximally non-primitive actions to respect user-intent: An action t is said to be *maximally non-primitive* if it is not dominated by any other action. We can respect user intent if we ensure that all the actions considered for establishment are maximally non-primitive. Thus, we will select $Travel(Phx, SF)$ action rather than the $Gobybus(Phx, SF)$ or $Gobytrain(Phx, SF)$ actions to establish $At(Phx)$ conditon. The maximal non-primitiveness restriction applies only to step-addition establishment and not for simple establishment refinements. This makes sense since once a primitive action capable of giving the condition is already in the plan, we may as well use it.

Handling partially hierarchicalized domains: In satisfying the "maximal non-primitiveness" restriction, there is a question as to whether or not HPS refinement should consider primitive actions which can establish the condition under question, but are not dominated by any other actions (e.g. the $hitchhike()$ action in the travel domain example). If the user feels that the domain has been completely hierarchicalized with respect to that particular conditio (in the sense that all desired solutions for achieving that goal will be parseable by the reduction schemas), then no primitive actions need be considered directly for any goals that are also given by non-primitive actions. Primitive actions will be considered if they are the only ones capable of giving the condition. Else, if the domain is still being hierarchicalized, then we will allow un-dominated primitive actions, but *bias* the search such that establishment branches corresponding to non-primitive actions are preferred over the branches corresponding to primitive ones.

Indexing non-primitive actions under primary effects to avoid non-minimal plans: Another potential concern is that selecting maximally non-primitive actions to establish a goal may lead us into scenarios where a complex plan is being performed only because the agent is interested in a secondary side effect of the plan. For example, suppose the planner has a single top level goal "$have(trainticket)$" (presumably because the agent is an avid ticket collector). The action $buytrainticket()$ will give the condition, and so presumably does the more abstract action $Gobytrain(x, y)$. Since the latter dominates the former, maximal non-primitive step restriction would require us to select the $Gobytrain()$ action to achieve $have(trainticket)$, which leads to a non-minimal (and inoptimal) plan for the agent. We can handle this type of scenario by noting that non-primitive actions do not have to be indexed under every one of their effects, but only some distinguished subset of them we call "primary effects." Thus, we index the $Gobytrain()$ action only under the $At(x)$ goal and not under $have(trainticket)$ goal. During the establishment phase, the planner will consider all actions that are indexed under the goal that is being established. Note that the primitive actions will continue to be indexed under all their effects. Since PS refinement is complete with just the primitive actions anyway, indexing non-primitive actions in terms of primary effects does not affect completeness. This is in contrast to the use of primary effects in primitive plans, as described by Fink and Yang [38].

Phantom Establishment to handle Non-primitive actions: Probably the most significant change to PS refinement that is necessitated because of the presence of non-primitive actions in the plan is with respect to condition establishment. When HPS refinement is working on a condition $p@t$ and action library contains non-primitive actions, the normal step addition and simple establishment operations will not suffice to guarantee completeness of the establishment. To see this consider a case where we have one non-primitive action A with a single effect p. Suppose there are only two primitive actions, A_1 and A_2, and a single reduction schema that reduces A to the plan fragment $A_1 \prec A_2$. A_1 requires the condition w and has the effects l, r and $\neg w$. A_2 has the precondition l and the effects p and q. Consider the planning problem where the initial state has the single condition w and the goal state has the conditions $p \wedge r$. When we consider p for establishment, we will make a single plan $t_0 \prec t_1 : A \prec t_\infty$ (since A is the maximally non-primitive action capable of giving p). Next, we consider the subgoal r, and since the step A does not give p, we have a single refinement $t_0 \prec \begin{pmatrix} t_1{:}A \\ t_2{:}A_1 \end{pmatrix} \prec t_\infty$. At this point, we decide to use pre-reduction refinement on A, and we will get the single plan:

$$t_0 \prec \begin{pmatrix} t_1^1 : A_1 \prec t_1^2 : A_2 \\ t_2 : A_1 \end{pmatrix} \prec t_\infty$$

This plan will not lead to a solution since the precondition every candidate of this plan will have two instances of A_1 and any such sequence will be unexecutable since A_1 deletes its precondition w and no other operator can add it back. Thus, the planner will decide that the problem has no solution. However, it does have a solution since the single action plan $t_0 \prec t_1 : A_1 \prec A_2 \prec t_\infty$ solves it.

What went wrong? Essentially, at the time we introduced A_1 to support r, we did not anticipate the possibility that the non-primitive action A will eventually be reduced in such a way that one of the steps in its reduction can give the condition r (which is exactly what happened here).

It might seem that the problem would not have arisen if the partial plan did not have any non-primitive actions at the time the condition r was picked up for establishment. Alas, this is not the case. If we considered r first, we would have the first plan $t_\infty \prec t_1 : A_1 \prec t_\infty$, and next when we consider p, we would have the plan $t_0 \prec \binom{t_2 : A}{t_1 : A_1} \prec t_\infty$, which after reduction becomes:

$$t_0 \prec \begin{pmatrix} t_2^1 : A_1 \prec t_2^2 : A_2 \\ t_1 : A_1 \end{pmatrix} \prec t_\infty$$

Again a plan containing no solutions. (The only way to be sure that this problem will not arise is if all actions are primitive since every action explicitly advertises all the conditions it can support.)

The traditional way of taking care of this problem is to use the so-called "phantom-establishment". This involves returning the current plan with $p@t$ as a PTC. Since the condition $p@t$ has been picked for establishment once, it will never again be picked up for establishment. Thus, the only way the PTC $p@t$ will hold (and it must for a candidate to be a solution) will be if at some later point, the reductions of the steps in the plan will wind up giving that condition.[16]

Other alternative ways of handling this problem is to make the pre-reduction refinement "context sensitive," or allow action merging to occur during the pre-reduction refinement. Allowing action merging can have significant ramifications on the refinement semantics. As an illustration of the context sensitive reduction, when reducing A, in the plan $t_0 \prec \binom{t_2 : A}{t_1 : A_1} \prec t_\infty$, the pre-reduction refinement should take into account the fact that the existing instance of A_1 can play the role of A_1 specified in the reduction schema (and thus only part of the reduction viz., A_2, needs to be introduced into the plan). Barrett [4] is the first to propose this idea. He calls it "frugal hierarchical task reduction" and argues that it cannot be avoided when one has actions with conditional and quantified effects. It is not clear how the performance will be affected by the increased complexity of context sensitive pre-reduction refinements.

The HPS refinement, which is a generalized version of PS refinement that incorporates the changes described above, is shown in Figure 17.

A.5.2 Modifications to BSS and FSS refinements

The BSS and FSS refinements need also be generalized in the presence of partial plans containing non-primitive actions. For example FSS refinement, as given, should not be attempted

[16] In [19, 20], we use a similar idea to handle "filter" conditions.

until the head fringe contains only primitive actions. The former restriction is required since a non-primitive action on the head-fringe may give rise to primitive actions on the head-fringe after some task reduction refinements, which cannot be considered if FSS is done earlier. In the travel domain, suppose we have only the actions corresponding to train travel, and we are considering refining a partial plan $P_1 : t_0 \prec Gobytrain() \prec t_\infty$. Applying FSS refinement to this plan will lead to the plan $P_2 : t_0 * buy(trainticket) \prec Gobytrain() \prec t_\infty$ (since $buy(trainticket)$ is the only action that can be done in the current initial state. However, the solution $Buy(trainticket) \prec Getintrain(x) \prec Getouttrain(y)$, which belongs to the candidate set of P_1 does not belong to the candidate set of P_2 (because each minimal candidate of P_2 will have two instances of $buy(trainticket)$, with the second instance coming from the reduction of $Gobytrain()$. Similar restrictions have to be applied for the BSS refinements.

A.5.3 Modifications to non-progressive Refinements

The easiest way of extending non-progressive refinements to handle non-primitive actions is to continue to restrict their use to primitive actions. In particular, pre-order and pre-positioning refinements should be allowed for only primitive actions, while the pre-satisfaction refinements should be applied only when an IPC involving two primitive actions is threatened by another primitive action.

It is also possible to extend pre-position and pre-ordering refinements to non-primitive actions if we extend the semantics of positioning and ordering refinements to non-primitive actions. Thus, we can pre-order two actions t_i and t_j by generating two refinements one in which $t_i \prec t_j$ and the other in which $t_i \not\prec t_j$. Notice that the latter constraint is not equal to $t_j \prec t_i$ since unlike primitive actions, two non-primitive actions may not only come before or after each other, but also may come interspersed with each other in any eventual solution.

Same type of treatment can be given to positioning constraints. Notice that when we allow such generalized positioning constraints, lazy FSS and lazy BSS routines do not necessarily give us state information. In particular, putting a contiguity constraint between a non-primitive action t_i in the head fringe, the head step extends the prefix, but does not uniquely define the state at the end of the prefix.

Once we are ready to disassociate state information from positioning constraints, we can also extend FSS and BSS refinements to consider putting non-primitive actions in the header and trailer. Of course, this involves handling both primitive and non-primitive actions for extension of the header, and raise the issues similar to those handled in designing the HPS refinement.

Generalizing pre-satisfaction refinements to non-primitive actions presents more problems. For example, consider the conflict resolution refinement involving an IPC $(t_1 \overset{p}{-} t_2)$ which is threatened by the step t_t, where all the steps are non-primitive. To avoid this threat, we need only ensure that the specific primitive step t_t^p, which actually deletes p in the eventual reduction of t_t should come outside the interval between t_1^p and t_2^p, where the latter two are the primitive steps resulting from reduction of t_1 and t_2 that give and take p respectively. Since the identity of t_t^p, t_1^p and t_2^p will not be known until the three non-primitive actions are completely reduced, there is no way we can post simple ordering relation between these non-primitive actions to resolve the threat. Thus, pre-satisfaction refinements can be applied only with respect to IPCs and threats where the producer and consumer steps of the IPC as well as the threatening step are all primitive.[17]

A.6 Ramifications of non-primitive actions on the consistency and completeness of refinement planners

Introduction of task reduction refinements has some ramifications on the completeness of UCP as well as the consistency checks used by UCP.

A.6.1 Completeness of UCP in the presence of Task Reduction Refinements

In Section 2, we noted that the completeness of the planner is guaranteed as long as the refinement strategies are complete. Given the definition of the candidate set of a non-primitive partial plan in Section A.2, it is easy to see that the Pre-Reduction refinement is complete. In particular, since the task reduction refinement strategy computes all possible reductions of t in $\mathcal{P}$, the union of candidate sets of the refinements of $\mathcal{P}$ is thus equal to the candidate set of $\mathcal{P}$ (as defined in Section A.3). Thus, every solution that belongs to the candidate set of $\mathcal{P}$ also belongs to the candidate set of one of the refinements of $\mathcal{P}$.

There is however one twist. In the case of state-space and plan-space refinements, it can be shown that UCP will eventually find every minimal ground operator sequence that is a solution. However, when UCP uses pre-reduction and HPS refinements (with HPS ignoring primitive actions during establishment), this guarantee does not hold – there may be ground operator sequences that are solutions, but UCP using task reduction refinements may be unable to find them. For example, in Figure 15, the solution $hitchhike(Phx, SF)$ will never be found, even though it does solve the problem.

Specifically, a ground operator sequence S that solves the problem is generated as a solution if and only if there is a way of reducing the initial null plan to S in terms of the reduction schemas provided to the planner. In [3], Barrett explains this in terms of the "parseability" of potential solutions in terms of the reduction schemas. From the refinement search point of view, task reduction refinements split the candidate set of a partial plan in a way that automatically prunes all ground operator sequences that will not have such a parse. Because of this, the solution space explored by task reduction refinement is different from that of plan space and state space refinements, and depends on the reduction schemas.

It is important to understand the practical utility of this difference in solution spaces. In many realistic planning problems, not every operator sequence that solves a problem may be an acceptable solution, as the users tend to have strong preferences about the *types* of solutions they are willing to accept. Consider the following two examples:

[17] Erol et. al. [8] provide a more elaborate description of handling pre-satisfaction refinements. However, they seem to suggest that pre-satisfaction refinements have to be postponed until there are no non-primitive actions in the interval between the producer and consumer. This is not strictly necessary as in our approach we merely require that the threatening steps and producer/consumer steps be primitive.

Example 1. A travel domain, where the user wants to eliminate travel plans that involve building an airport to take a flight out of the city (or even, stealing money to buy a ticket).

Example 2. A process planning domain with the task of making a hole, where there are two types of drilling operations, D1 and D2 and two types of hole-positioning operations H1 and H2. A hole can be made by selecting one hole-positioning and one hole-drilling operation. The user wants only those hole-making plans that pair D1 with H1 or D2 with H2.

In both the examples, the user is not satisfied with every plan that satisfies the goals, but only a restricted subset of them. Handling such restrictions in partial order planning would involve either attempting to change the domain specification (drop operators, or change their preconditions); or implementing complex post-processing filters to remove unwanted solutions. While the second solution is often impractical, the first one can be too restrictive. For example, one way of handling the travel example above is to restrict the domain such that the "airport building" action is not available to the planner. This is too restrictive since the there may be other "legitimate" uses of airport building operations that the user may want the planner to consider.

HTN approach provides the user a more general and flexible way of exercising control over solutions. In particular, by allowing non-primitive actions, and *controlling their reduction through user-specified task-reduction schemas, HTN planning allows the user to control the planner's access to the actions in the domain*.

This flexibility does not come without a price however, since the specification of reduction schemas needs to take into account the possible interactions between high level tasks. Errors in task-reduction schema specification may lead to loss of desirable solutions. Often times, the errors can be subtle and require an understanding of various interactions that may occur. For example, suppose one is modeling a transportation domain, and wants to write a reduction schema for the $Transport - by - plane(plane, package, dest)$ action. The straightforward way of specifying this would involve a reduction $load(package, plane) \prec fly(plane, dest) \prec unload(package, plane)$. While this reduction is reasonable, direct usage of it in situations where multiple packages are required, will lead to inoptimal plans that use two planes and two trips. Avoiding such inoptimality will involve specifying schemas in such a way that phantom establishments can be used to avoid the second trip [8] (which is not possible given the way the reduction schema is written), or using more complex context dependent forms of pre-reduction refinements [4] (see Section A.5.1). The former requires complex domain engineering and the latter may increase the complexity of pre-reduction refinement. There is some recent work [7] that addresses the automated debugging of task reduction schemas, but much needs to be done.

A.6.2 Checking the Consistency of a Partial Plan

One consequence of the use of task reduction refinements in UCP is on the consistency check used by UCP. Specifically, checking whether a non-primitive partial plan is inconsistent (i.e., has an empty candidate set) would involve reasoning with all the concretizations of the plan. It would seem at first that non-primitive plans with empty candidate sets can be detected by

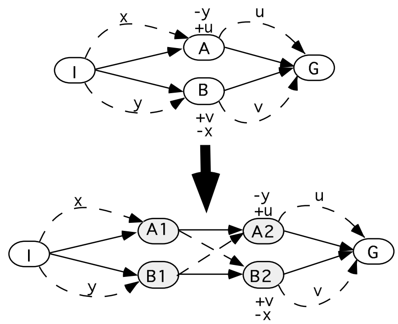


Figure 18: Example showing that the candidate set of a plan containing non-primitive actions may be non-empty even if it has no safe ground linearizations

checking for the absence of safe ground linearizations (as is done for primitive partial plans). Unfortunately however, unlike the safe ground linearizations of primitive plans, which have a direct relation to minimal candidates of the plan (see Figure 2), partial plans containing non-primitive actions correspond to many concretizations and thus do not directly correspond to any minimal candidates. So, the existence or the lack of safe ground linearizations of a partial plan containing non-primitive actions has no relation to its inconsistency.

Another way of looking at this is in terms of evaluation of the truth of a constraint in a partial plan. Given an IPC and a primitive partial plan, we can check if the plan currently satisfies the IPC by checking if any of its ground linearizations satisfy it. However, this type of check does not work with non-primitive partial plans, since it may well be that the concretizations of the plan do have safe ground linearizations. We illustrate this by the example in Figure 18.

This is illustrated by the example in Figure 18. The plan at the top has two non-primitive actions A and B, with effects as shown, and four IPCs. It is easy to see that the plan has no safe ground linearizations (as there is no ground linearization that is safe with respect to all the IPCs). Suppose there is a way of reducing A to $A_1 \rightarrow A_2$ and B to $B_1 \rightarrow B_2$, where A_1, A_2, B_1 and B_2 are all primitive. In this case, the lower plan, which results after these reductions, is a primitive plan, and is thus a concretization of the top plan. We note that this plan *does* have safe ground linearizations. Thus, we cannot prune the top plan, and the absence of safe ground linearizations of a non-primitive partial plan cannot be taken as an indication of empty candidate set for a non-primitive plan.

A variant of the above point was first noticed by Yang [37], who pointed out that non-primitive partial plans with unresolvable conflicts (which imply no safe ground linearizations), cannot always be pruned without loss of completeness. Yang also points out that one sufficient condition for making such pruning admissible involves redirecting all the IPCs emanating and terminating into a single non-primitive actions to another unique action after the reduction (the *unique main action redirection* discussed in Section A.2). Whether this type of constraint can be satisfied by natural encodings of planning domains is not clear. Clearly, the restriction is not satisfied in the example in Figure 18, since the IPCs incident on and emerging from a single action, A, are redirected to different actions (A_1 and A_2 after reduction).

A.7 Related Work

HTN planning has been considered by many, including some of the recent authors [8] to be a fundamentally different type of planning that should not be combined with "STRIPS" action baed planning. This stand leads to re-invention of many of the plan-space planning ideas. For example, NONLIN [33] allows nonprimitive actions of type $Achieve(c)$, which essentially take the part of condition establishment. Here we took the view that the main idea of HTN planning is to allow non-primitive actions on top of primitive actions, and showed how non-primitive actions can be consistently combined into UCP framework, which already covers the state-space and plan-space approaches. Our view allows for domains that are "partially hierarchicalized" in that reduction structure exists only for part of the domain. It also shows that things like "phantomization" are really an artifact of doing plan-space refinement in the presence of non-primitive actions.

Other research efforts, including IPEM [1] and DPOCL [39], attempted to combine task reduction and plan-space planning. Neither of these approaches however guarantees an integration that preserves both systematicity and user-intent inherent in the specified reduction schemas. For example, DPOCL allows for the establishment of a precondition both with a non-primitive action and a primitive action which can be its descendant. As we argued earlier, this can lead to a loss of systematicity as well as violation of user-intent. Our approach also provides clear semantics of plans containing non-primitive actions, and thus explains the essential differences between the completeness results for primitive and non-primitive partial plans.

Another research effort that allows combination of task reduction and state-space refinements is that of Chien and his co-workers [7]. They argue that the task reduction and plan-space refinements should be kept orthogonal with the former working on activity goals while the latter work on state goals. Domain information pertaining to these techniques is also kept separate. While our approach allows for such a separation, it also allows for task reduction and plan-space refinements to work on the same class of goals in a principled way.