2012

Hierarchical Planning Knowledge for Refining Partial-Order Plans

Stephen Montgomery Lee-Urban
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Hierarchical Planning Knowledge for Refining Partial-Order Plans

by

Stephen Montgomery Lee-Urban

A Dissertation
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Computer Science

Lehigh University
May 2012

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

_____
Date

_____
Chair: Héctor Muñoz-Avila

_____
Brian D. Davison

_____
Jeff Heflin

_____
Dana Nau

Acknowledgements

I am humbled by and grateful for my family and friends, who I love dearly. They have been an inexhaustible source of encouragement, inspiration, support, guidance, and perspective. Expressing my love and appreciation for my incredible parents, for all they have done, is like trying to use words to describe a painting or music; just as poets have interminably struggled to convey the intricacies of love, so too do I accept that no acknowledgement put forth here can convey how profoundly they have enriched this dissertation, and my life.

If I could, I would put the following names on the title sheet of this dissertation, and for more reasons than I list below:

- Kathy (Ed.D., Lehigh University, 1998) and Alfred Lee, my mother and step-father. It takes more than knowing how to walk in order to keep putting one foot in front of the other, and more than goals in order to create opportunities. Very truly, thank you for encouragement, inspiration, editorial perspective… everything both expressible and otherwise.

- James Urban, my father, who always knows where to look when I've lost my sense of humor or smile. Thank you for insightful perspective through laughter, and for always being there for me.

- My brother, Nik, who has always had a knack for being, having, or doing exactly what I need, just when I needed it. The best big brother and champion anyone could ask for.

understanding while I balanced researching for him with wrapping up my degree.

Table of Contents

List of Tables

List of Figures

Abstract

Automated plan synthesis, or "generative planning", is a process whereby a course of action – a plan – is generated to achieve some desired goals. When the planning process uses a previously formulated plan as a starting point for solving a new problem, the process is called "plan adaptation". Over the years there has been a significant research effort on plan adaptation. Part of the reason for this continuing interest in plan adaptation is attributable to studies indicating a wide range of potential applications, which include military planning, computer gaming, narrative computing, manufacturing, route planning, and medicine. Despite this remarkable body of research, existing plan adaptation algorithms do not scale well with problem size; even on medium-size problems, the performance of plan adaptation tends to be rather poor. Furthermore, for some worst-case situations it has been proven that the computational complexity of plan adaptation can be greater than that of generative planning. Although these worst-case scenarios have been shown to be inapplicable to most existing adaptation algorithms, it is nevertheless a fact that the lack of scalability of plan adaptation techniques is a major hurdle preventing their use in real-world applications.

The main goal of this dissertation is to address the problem of how to efficiently represent and apply high-quality, hand-crafted plan refinement and plan adaptation knowledge. To accomplish this goal I studied *domain-configurable plan adaptation*, the results of which are presented in this dissertation. This new problem-solving paradigm uses domain-specific knowledge about plan adaptation to guide a domain-independent

plan adaptation algorithm. This new technique is novel in that existing approaches for plan adaptation modify the input plan by either using domain-independent plan adaptation knowledge in a domain-independent adaptation algorithm or domain-specific plan adaptation knowledge in a domain-specific adaptation algorithm. By focusing on the use of hand-crafted expert partial-order planning control knowledge and its representation, one most notably eases the burden of knowledge acquisition – that is one does not need a complete knowledge base in order to create good plans. This permits the knowledge engineer to focus on encoding strategies most relevant to solving the problems, while leaving the more mundane and human-difficult chores of "plan bookkeeping" to the underlying planner.

1   Introduction

Automated planning has been one of the central topics of Artificial Intelligence (AI) since the inception of the field. One reason for planning's long incumbency is the desire of AI researchers to create systems, or agents, that behave rationally to achieve a desired outcome. It seems reasonable to assume that a key element of rational behavior is acting with deliberated purpose [Wolldridge, 2002].

1.1   Overview of "Planning" and key open challenges

Planning, or automated plan synthesis, therefore represents the reasoning side of taking actions that a rational agent might pursue [Ghallab et al., 2004]. This reasoning involves the selection and ordering of actions based upon their predicted outcomes in order to achieve a pre-stated goal. The purpose of planning is to, given a formal description of the initial and goal states of the world, along with a set of rules for how the world can be systematically altered, generate a plan that specifies how the rules can be applied to transform the initial state into the goal state. Such cognitive behavior is the foundation for critical applications such as: efficient scheduling of industrial equipment, airport traffic control, web service composition, expert-level automated game players, scientific workflow planning, narrative computing (e.g. create a story to help solicit medical information from a sick patient), space exploration (e.g. controlling the mars-lander), and emergency evacuation planning.

While there are a wide variety of subfields within the broad topic of planning, this dissertation focuses on the class of problems called "classical planning" (STRIPS-

family). Classical planning, in the context of AI, is a symbolic approach to problem solving first developed at Stanford Research Institute (the "STRI" in STRIPS) in 1971, and is further described in Section 3.1 of this document. Other classes of planning, such as path and motion planning, perception planning, navigation planning, manipulation planning, and communication planning are outside of the scope of this dissertation.

In spite of its long incumbency within AI and the breadth of application domains in which fielded automated planners can be found, there are several dimensions along which the state of the art in classical planning has enjoyed only limited success. Notably, modern planners continue to seek advances in four important areas, summarized in **Table 1.1**: (1) the repair or adaptation of previously formed plans, (2) the generation of plans for very large problems, (3) the generation of flexible plans, which are less constrained about the order in which the actions are executed, and (4) the efficient representation and reuse of hand-crafted expert domain knowledge and preferences.

While successes have been achieved within and across each of these dimensions, no single framework exists that is capable of addressing all four dimensions simultaneously. Specifically, how to solve the lack of scalability of plan adaptation techniques remains an open question, and is a major hurdle preventing its use in real-world applications. The focus of the research presented in this dissertation is the design and analysis of an automated adaptive planning technique that can succeed in all four of these areas, thereby advancing the state of the art in planning. The key technique, which I call HIEPPR-POP – for HIErarchical Partial Plan Refinements for Partial Order Plans (pronounced "hyper pop") – is a new way of representing and using expert "control-knowledge" to guide a planner in its solution generation process. This new control-knowledge can be used to

Dimension	Definition	Example
Plan adaptation	the repair or adaptation of previously formed plans (solve new problems by reusing solutions)	Prodigy /Analogy
Problem size	the generation of plans for very large problems (those with many goals)	SHOP2
Flexible solutions	the generation of flexible plans (solutions with fewer constraints on action ordering)	VHPOP
Expert knowledge	the efficient representation and reuse of expert domain knowledge and preferences	TLPLAN

Table 1.1 Dimensions of difficulty for modern planning approaches

both solve problems from scratch, and from previous solutions. Furthermore, to ease the notoriously difficult knowledge acquisition problem, the extra knowledge does not have to be complete, which is an uncommon quality for hierarchical approaches. It is in this 4th dimension that I believe HIEPPR-POP makes its most substantial contribution. Each of the dimensions is expanded upon below.

Dimension 1, Adaptation: The first dimension of difficulty for planners is that of the repair or adaptation of previously formed plans. Plan adaptation, as the name implies, extends the generative planning process to include making changes to a pre-existing plan; that is, plan adaptation is the process by which an old solution is modified to solve a new planning problem. These changes include the retraction and modification of past decisions made about actions and their ordering, as well as possibly adding new actions. One classic, motivating example is making changes to a cooking recipe to accommodate a slight variation of available ingredients. So, if margarine is on hand instead of butter it would seem natural, faster, and easier to simply substitute one for the other, rather than re-creating the whole recipe from scratch. Because recipes are sequences of steps, they can be seen as plans. Creating the recipe from scratch can be loosely taken as generative planning whereas modifying an existing recipe for a new problem can be taken as plan

adaptation. Historically, plan adaptation is a subfield that has been slow to see progress and realized successes [Greene et al., 2008], and is an area in which the HIEPPR-POP technique is poised to make a substantial contribution. Section 3.8 summarizes different approaches to plan adaptation.

Over the years there has been a significant research effort on this problem-solving technique. Works include complexity analysis for worst case scenarios [Nebel & Koehler, 1995], search-space analysis [Au et al., 2002; van der Krogt & de Weerdt, 2005; Kuchibatla & Munoz-Avila, 2006], plan merging [Veloso, 1994; Munoz & Weberskirch, 1997; Ram & Francis., 1996; Tonidandel & Rillo, 2005] and plan adaptation algorithms for several planning paradigms including total-order planning [Veloso, 1994], partial-order planning [Ihrig & Kambhampati, 1997; Munoz-Avila & Weberskirch, 1996], HTN planning [Kambhampati, 1994; Warfield et al., 2007], planning graphs [Gereveni & Serenia, 2000], and heuristic planning [van der Krogt & de Weerdt, 2005]. Part of the reason for this continuing interest in plan adaptation is attributable to studies indicating potential applications, which include military planning [Mitchell, 1997; Veloso et al., 1997; Munoz-Avila et al., 1999], computer gaming [Ontañón et al., 2007; Sanchez et al., 2007], manufacturing [Costas & Kashyan, 1993; Munoz-Avila & Weberskirch, 1996; Veerakamolmal & Gupta, 2002], route planning [Haigh et al., 1997], and medicine [Schmidt et al., 2001, 2003; Salem et al., 2003].

Despite this remarkable body of research, existing plan adaptation algorithms do not scale well with problem size. Even on medium-size problems, the performance of plan adaptation tends to be rather poor. Furthermore, for some worst-case situations it has been proven that the computational complexity of plan adaptation can be greater than that

of first-principles planning [Nebel & Koehler, 1995]. First-principles planning, also called STRIPS planning [Fikes & Nilsson, 1971], solves new problems by reasoning on the given action schemas to generate the plans from scratch. Although these worst-case scenarios have been shown to be inapplicable to most existing adaptation algorithms [Au *et al.*, 2002; Kuchibatla & Munoz-Avila, 2006], it is nevertheless a fact that the lack of scalability of plan adaptation techniques is a major hurdle preventing their use in real-world applications. This lack of scalability makes it difficult to consider dropping classical planning assumptions about temporal constraints and non-determinism in the outcome of the actions. While the work presented herein does not address planning with non-determinism and temporal constraints, it takes the important first step of even making this a consideration for problems of the proposed size. The classical planning problem, and hence the classical adaptation problem, assumes that action execution is instantaneous, and that the effects of actions are pre-determined. But in many real-world applications, planning software is needed that can reason with time constraints, and where the outcome of the actions is not pre-determined [Ghallab et al., 2004]. Although planners have been developed capable of dropping these assumptions [e.g., Bacchus & Ady, 2001; Vidal & Geffner, 2006; Onder *et al*., 2006], no plan adaptation algorithm exists capable of dealing with these issues.

Dimension 2, Problem Size: The second dimension of difficulty for planners is that of solving very large problems. The size of a planning problem is related to the number of objects in the problem's initial state and how many goals are to be achieved (problem difficulty increases as the size of the initial and goal states grows). The ability of a planner to solve large problems is the deciding factor for its application to the real-world;

a planner is not very useful in practice if it can only solve small, "toy" problems. It is for this reason that few domain-independent planners are successfully fielded, and instead why domain-configurable or domain-specific planners dominate industrial applications and planning competitions. Section 3.5 summarizes the differences between these approaches. For the purposes of this introduction, it is sufficient to say that the approach presented in this dissertation is a domain-configurable one, given its historical success with large problems.

Dimension 3, Solution Flexibility: The third dimension of difficulty is that of generating flexible plans. A plan is considered flexible (during plan execution) when the sequence of actions representing the problem's solution has a low degree of ordering constraints. That is, the more constraints on the ordering of actions in a planning solution, the more tightly constrained and less flexible the solution plan is for execution; the fewer the constraints on action ordering, the greater the plan's flexibility. Take for example a plan that has a totally-ordered sequence of steps. This total-ordering means that only one step can be executed at a time. Should any one of those totally ordered steps fail during the plan's execution, the remainder of the plan cannot be carried out, and nothing can be done until a new action sequence is generated. In contrast, a plan that only has a partial-ordering on its steps is more tolerant to failure; should one of the steps fail, there may be other steps that can consistently (relative to the ordering constraints) be executed. This flexibility is desirable not only because it allows more time to deliberate over a failure (given that other steps can be executed while the problem is repaired), but also because there are many domains in which parallel action execution is desirable even in the absence of failure.

A simple example of a domain where it is desirable to have actions that can occur simultaneously is that of package delivery. With multiple packages to be delivered to various destinations by multiple vehicles, it is absurd to imagine that each package must be handled one-at-a-time; rather, each vehicle can and should be simultaneously delivering its payload of packages to their destinations. It is for the reason of execution flexibility that the approach presented in this dissertation is built upon a "partial-order plan" representation. Section 3.2 summarizes the differences between partial-order and total-order representations, and how they relate to the planner's search space. It is notable that planners that generate partially-ordered action sequences often do so at the expense of their ability to solve large problems, and therefore the design and analysis of a technique that can succeed in the second and third dimensions of planning difficulty is a compelling scientific goal.

Dimension 4, Expert Knowledge: The fourth and final dimension of difficulty for planners is the efficient representation and reuse of expert domain knowledge and preferences. In many real-world planning domains, there exists a corpus of knowledge about how certain types of problems in those domains are solved, and preferences about solution action sequences that are related to standard operating procedures. For example, an expert in emergency evacuation planning may know that for groups of fewer than four people a helicopter is the best vehicle option, even though other viable means of transport are available. If a planner cannot represent and reuse this knowledge, then the solutions it generates are likely of a lower quality than hand-crafted solutions; furthermore, the planner must spend time searching for solutions that will be immediately rejected. This is another strong motivation for my technique being a domain-configurable one. However,

in designing systems that can reason with this extra knowledge, one introduces a "knowledge-engineering bottleneck" that can make the planner too difficult to use (as acquiring useful knowledge can be too burdensome). Therefore one of the highest-priority design constraints of the technique presented in this dissertation was to support the efficient representation and reuse of this extra planning knowledge. While this design constraint is not new, the approach presented herein is novel in that expert knowledge does not have to be "complete" – that is the planner can find solutions that use as little or as much knowledge as can be obtained.

1.2 Contributions of This Work

The following is a summary of the scientific contributions of this dissertation to the state-of-the-art in planning research:

- **Ability to make partial-order plans for large problems.** Algorithms that use a partial-order plan representation historically cannot generate plans for problems with many goals, which require many actions to solve the problem. I show that in some situations, the HIEPPR-POP algorithm can do so. This is notable in that partial-order solutions can be far more flexible in their execution than totally-order solutions, and also partial-order plans are believed to be a better approach for supporting actions with duration (a condition that exists in many real-world planning problems).

- **Fast, generative domain-configurable partial-order planning.** The main goal of this research was to study scalable (with respect to problem size) and well-founded plan adaptation. However, along the way to achieve this goal the study

included generative domain-configurable partial-order planning. This yielded a new planning algorithm that scales well with problem size, outperforming existing domain-independent partial-order generative planning algorithms in terms of time taken to find a solution (while sacrificing solution quality as measured by plan-length).

- **Well-founded plan adaptation**. HIEPPR-POP has clear semantics specifying the conditions under which soundness (i.e., under which conditions are plans generated guaranteed to be correct) and completeness (i.e., under which conditions are plans guaranteed to be generated when a solvable problem is given) can be guaranteed.

- **The ability to generate adaptation solutions using incomplete adaptation knowledge.** HIEPPR-POP is a domain-configurable adaptation approach that does not require complete adaptation knowledge to generate a solution (because at all times it is refining a partial-order plan, this partial plan can be completed by first-principles whenever there are gaps in the encoded domain-configurable knowledge). This is somewhat novel in modern planning research and eases the knowledge-engineering bottle-neck. A caveat to this contribution is that the ability to extend incomplete solutions generated by HIEPPR-POP into complete solutions (having no flaws) is predicated both on the quality of the first-principles planner used to complete the plans, and the quality of the domain-configurable knowledge used to produce the incomplete solution. In general, even the best implementations of partial-order planning techniques have difficulties finding solutions for even medium-sized problems – therefore, refining incomplete solutions is typically only

possible when the first-principles refinements required need no, or little, backtracking.

- **Ability to study the trade-offs between knowledge given and performance gains**. As a result of the previous bullet, it is possible to investigate the trade-offs between the amount of domain-configurable knowledge given and its result on planning performance, measured in running time and percentage of actions retained.

- **A testbed for other researchers.** HIEPPR-POP is the first adaptive, domain-configurable partial-order planner freely available for the Case-Based reasoning and Planning research communities. This enables others to do focused research on other important areas of plan adaptation, such as the problem of which partial plan should be used at the start of the adaptation process (retrieval), the problems of which steps to remove from the plan to adjust and how to adjust the mapping of objects used in the previous solution with objects in the new problem, without having to first build or re-implement a successful plan-adaptation technique.

- **The capability to retain a significant portion of the plan to be adapted.** The HIEPPR-POP approach should be able to retain a significant portion of the plan when feasible. The amount retained will necessarily depend upon the input plan, the conditions of the new problem, and the domain-specific plan adaptation knowledge provided. This capability is a consequence of the ability of the algorithm to solve large problems, and the use of a partial-order plan representation that captures plan commitments at a finer level of granularity than those used in total-order plan representation (a simple sequence of actions).

- **No tradeoff between time spent to find a previous solution to adapt, versus time spent performing adaptation**. Under some (highly constrained) conditions, the HIEPPR-POP approach (specifically, the DCPOP algorithm) was shown to take roughly the same amount of time to produce a solution to a new problem through adaptation, regardless of the source plan used to make the solution. This is counter to the well-established retrieval-adapt tradeoff commonly considered inescapable in the case-based reasoning literature.

1.3 Dissertation Outline

The remainder of this document proceeds as follows: Chapter 2 informally describes the dissertation topic, summarizing the technique and steps through a simple example. Chapter 3 presents background material and research related to the technique presented in this dissertation. Readers interested in background material related directly to plan adaptation should refer to Chapter 3.8. Chapter 4 presents the core of the technique, and is followed in Chapter 5 by useful extensions that increase the power of the approach while simplifying its use. Next, Chapter 6 shows the results of an empirical evaluation. The dissertation closes with a section offering conclusions and future directions to explore (Chapter 7).

2 Overview of Dissertation

The main goal of this dissertation is to address the problem of using high-quality, but incomplete, expert planning knowledge in a manner that allows for scalable planning that includes adaptation, and to do so in a way that allows for flexible plan execution.

That is, imagine a problem that is solvable by planning. Some examples of "planning problems" are stacking blocks, sequencing moves in the card game Freecell, scheduling the transportation logistics for delivering packages – anything that involves coming up with a sequence of actions to achieve a desired outcome. Suppose also there is some expert knowledge about how to best solve this problem, or a preference about how problems in that "domain" should be solved. For example, in the transportation logistics domain, an expert might say a nearby truck should always pick up a package that is not yet at its final destination. This dissertation presents a theory on how to represent and use this hand-crafted expert knowledge in a new way. The main questions this dissertation addresses are:

(1) *Representation*: how to express this hand-crafted expert knowledge or preference for solution approaches? What constraints does the chosen representation place on the flexibility of solutions created by using this expert knowledge?

(2) *Reuse*: how to use this knowledge to solve other problems in the same problem domain? Is there a way that this knowledge can be used to both solve problems from scratch, and solve problems by reusing other solutions? Does the application of expert knowledge lead to "better" plans?

(3) *Scalability*: How does the use of this knowledge affect the speed of finding a solution? What is the relationship between the amount of expert knowledge, and the size of the problems solvable by the approach?

(4) *Scarcity*: can the new automated problem solving theory work with as little or as much expert knowledge as is available?

(5) *Domain-independence*: can the same representation and mechanism of reasoning be used across different problem domains? What types of problems are solvable with this approach?

The broad title of this dissertation is "Hierarchical Planning Knowledge for Refining Partial-Order Plans". The solution presented achieves this goal in a systematic way that facilitates future extensions to drop classical assumptions, such as instantaneous action execution and deterministic outcome of actions. To date, there exists no domain-independent plan adaptation algorithm capable of dealing with the combined issues of making flexible planning solutions to very large problems by using expert knowledge that can be incompletely specified. To accomplish this goal I designed and investigated a new domain-configurable plan adaptation algorithm. This new problem-solving paradigm uses domain-specific knowledge about planning to guide a domain-independent planning algorithm. Existing approaches for plan adaptation modify the input plan by either using domain-independent plan adaptation knowledge in a domain-independent adaptation algorithm (e.g., [Hanks & Weld, 1995; van der Krogt & Weerdt, 2005; Tonidandel & Rillo, 2005; Kuchibatla & Munoz-Avila, 2006]) or domain-specific plan adaptation knowledge in a domain-specific adaptation algorithm (e.g., [Hammond, 1986; Fagan & Cunningham, 2003; Corchado et al., 2007]).

2.1 Plan Adaptation – The "Grand" Challenge

Despite the successes reported above, existing adaptation procedures are still far from solving what can be called the original plan adaptation challenge. The challenge can be stated as follows: how to perform plan adaptation when there is implicit knowledge in the source plans and, hence, the adaptation process must carefully modify the source plan to preserve such knowledge. The source plans describe high quality procedures and carefully crafted adaptation rules prescribe how these plans can be modified to preserve the quality of the source plans. This problem remains a challenge due to the knowledge engineering bottleneck: encoding *complete* plan adaptation knowledge information in a formal language is unfeasible because of the time it would require to encode such knowledge and because, even if domain experts where to be made available, some of the actual processes followed by the expert are not explicit as the experts are guided by experience rather than a well understood problem solving theory.

A notable approach to solving the plan adaptation with quality measures challenge is the PbR system [Ambite et al., 2005]. Although it is not adapting a solution relative to a new problem, PbR uses rules to modify an existing solution plan into another solution for the same problem but one that has better quality. Unlike the Fox et al. (2006) system, PbR does not need to explicitly represent the plan quality measurements because these are implicitly encoded in the rules, which are carefully crafted by a domain expert. So this system falls in the category of domain-configurable planners; systems in which domain-specific knowledge containers are provided to guide a domain-independent plan generation process. In addition to the term rewriting rules in PbR, other formalisms proposed to represent the domain-specific knowledge includes hierarchical task networks

[Nau *et al.*, 1999; 2005], temporal logic planning [Bacchus & Kabanza, 2000; Kvarnström, & Doherty, 2001; Kvarnström & Magnusson, 2003], and domain-configurable rules [Lee-Urban & Munoz-Avila, 2009]. Regardless of the particular representation formalism, the main drawback of such systems is the potentially large knowledge engineering effort required to encode the domain-specific knowledge. Nevertheless, the approach presented in this dissertation assumes as input the adaptation knowledge. To ease the burden of the knowledge engineering effort, a quality of the adaptation knowledge I seek to use is that it can be incomplete.

The original plan adaptation challenge remains important today. In many real-world domains, carefully crafted plans with implicit quality guarantees are available. Generating plans from scratch with similar qualitative or safety guarantees is not a realistic option. Instead, procedures are needed that adapt these plans to new situations while preserving their quality or safety guarantees. An example is military planning, where rather than explicit measures, semantic categorizations can be used to examine how plans relate to one another [Myers, 2006]; military plans that appear syntactically different in regards to the actions they involve might actually be semantically similar because they execute the same strategy (e.g., a pincer maneuver). Domains where the existence of such plans has been documented include forest fire management [Avesani, et al., 1993], and medicine [Miksch, 1999; Schmidt *et al*., 2001, 2003; Salem *et al*., 2003]. Clinical protocols indicate how to treat a patient with a specific disease. Generally, there is no need for generating new treatments from scratch; rather a large number of treatment protocols have already been acquired. This is the result of a concerted effort by the health care community to improve quality assurance by reducing variance in clinical practice

(Miksch, 1999). The challenge is to adapt these existing treatments to the specific circumstances of the patient. Another example is Emergency Crisis Plans [US Dept. Edu., 2009]. Such plans have been created, or are in the process of being created at every level of our society from small elementary schools to large cities. These plans are pre-conceived activities indicating what individuals or sub organizations must do in case of an emergency and frequently are the result of careful inter-agency negotiations. Hence, sensible modification of these plans to adapt to the changing circumstances is crucial.

2.2 A Motivating Example

In this section, a simple example is given that motivates the problem, and gives insights about the research presented in this dissertation. The purpose is to impart to the reader at a high-level the problem addressed, and a sense of the solution presented. Without such an example, much of the next chapter will lack motivating context. To ground the explanation, I take up the "transportation logistics" example hinted at in the beginning of this chapter. This "domain" is like one that FedEx might use for its business needs – delivering packages from their starting locations to their final destinations.

The example proceeds as follows: first, the general problem of planning is restated. Next, the transportation logistics problem "domain" is explained. Subsequently a means of symbolically representing entities in the world, and relationships between them, is presented. This representation is then used to define, by specifying "initial" and "goal" states, a single problem within the transportation logistics domain. After that, a means of representing actions that can be taken in the domain is provided. Having specified how to represent states and actions, the example moves to a discussion of how they can be used

to make solutions to the example problem, differentiating between "state-space" and "plan-space" search. The example concludes with a discussion of how "expert knowledge" can be used in the solution generation process.

Restating the planning problem: Automated planning is a methodology for creating action sequences. An action sequence is simply a set of steps that when followed in a particular order, transforms the "state" from its initial configuration to a goal configuration. In imprecise terms, a "state" is a symbolic (uses an alphabet and words) representation of what is true in the world (problem at hand). An "action" is akin to a verb in human language– it denotes something that changes what is true in the world. States and actions are inherently tied to the problem area in which a solution is sought. This problem area is referred to as the "domain", which is synonymous with "planning domain" and "problem domain". A "problem" in any planning domain is characterized by the initial and goal states, and the actions available for sequencing. How does the transportation logistics domain define actions and states?

The transportation logistics problem domain: In transportation logistics, there is the notion of the following real-world entities: packages, trucks, airplanes, cities, and locations. A package can be in a truck, in a plane, or at a location. All locations are contained within cities; trucks can move between any locations within a city; airplanes can move between the airports in each city.

Representing the state: Symbolically the real-world entities can be represented as follows. Suppose there is a package called `p-1`, a truck called `t-1`, a city called `Bethlehem`, and two locations within that city, the location `LehighU` and the location `LVI`, which is an airport. The corresponding symbolic representations of these entities

could be: (package p-1), (truck t-1), (city Bethlehem), (location LehighU), (location LVI), and (airport LVI). The parentheses are used to group items. Within a grouping, by "pre-fix" convention, the first word indicates the type of entity, and the second word (sometimes referred to as "argument") indicates the instance of that entity. In general however, what comes within the parentheses can take many forms, and is not limited to any particular order or number of arguments, so long as the representation is well-defined. For example, there are many cities, but representing them symbolically will take the form (city ?x), where ?x is a particular city name. Each unique entity must have a unique representation.

Having represented the entities, it is clear that the relationships between them are missing. For example, how is the fact that Lehigh University is located in Bethlehem captured? One way to do so would be: (in-city LehighU Bethlehem), (in-city LVI Bethlehem). A similar approach can be used to represent "world state" configurations. For example, to assert that the package p-1 is currently at the airport LVI, one could write (at p-1 LVI). To state that the truck in this problem is located at LVI, one could write (at t-1 LVI).

Initial State	Goal State
(package p-1) (truck t-1) (city Bethlehem) (location LehighU) (location LVI) (airport LVI) (in-city LehighU Bethlehem) (in-city LVI Bethlehem) (at p-1 LVI) (at t-1 LVI)	(at p-1 LehighU)

Figure 2.1 Example initial and goal state in transportation logistics domain

20

There is now enough syntax to specify the beginning of a planning problem. A problem is partially defined by its initial and goal states, each of which is a set of symbolic facts. An example problem in this domain is shown in Figure 2.1. It describes a starting configuration of a single package that starts at LVI airport and needs to be delivered to Lehigh University. Note that because the final location of the package is the only requirement of this problem, only this fact need be specified in the goal state. Available in this problem is a single truck, and a single airplane. How can the planning problem be solved? Without any actions available, no sequencing can be made! A planning problem therefore also takes in a representation of actions that can be taken in the problem domain.

Representing the actions: The notion of actions in planning builds upon the syntax used to represent world state. In this planning domain, some of the real-world actions relevant to solving the problem include loading and unloading a truck, and driving between locations. Whatever representation is used for actions must capture the relationship of that action to the symbolic world state. Specifically, anything that can be done to effect change in the real-world has constraints, and this must be modeled in the representation of actions– certain things must be true in order to take an action, and after taking an action certain effects are realized in the world. For example, it is impossible to load a package into a truck if the truck and package are not in the same location; it is impossible to unload a package that is not in a truck; unloading a package removes it from the truck and "delivers" it to the location at which the truck was when the unload action was performed. The means of representing world state is reused in this situation to express preconditions (things that must be true for an action to be usable) and effects (the

21

things that are made true, or untrue) of actions. An example of the load, unload, and drive

"operators" are shown in **Figure 2.2**. The difference between an action and an operator

will be explained shortly.

Load and Unload operators	Drive operator
<pre>(**name:** load-truck
 parameters: ?t ?p ?loc
 preconditions:
  (truck ?t)(package ?p)
  (at ?t ?loc)
  (at ?p ?loc)
 effects:
  (in-truck ?p ?t)
  (not (at ?p ?loc))
)

(**name:** unload-truck
 parameters: ?t ?p ?loc
 preconditions:
  (truck ?t)(package ?p)
  (at ?t ?loc)
  (in-truck ?p ?t)
 effects:
  (not (in-truck ?p ?t))
  (at ?p ?loc)
)</pre> | <pre>(**name:** drive
 parameters: ?t ?from ?to
 preconditions:
  (truck ?t)
  (location ?from)
  (location ?to)
  (same-city ?from ?to)
  (at ?t ?from)
  (at ?p ?loc)
 effects:
  (in-truck ?p ?t)
  (not (at ?p ?loc))
)</pre> |

Figure 2.2 Three actions in the transportation logistics domain

In **Figure 2.2**, the symbolic representation of facts has been extended to include

"variables", which are those elements preceded by a question mark. Without variables,

the specification of actions would be tedious – for every object (each truck, each package,

and so on) in the domain, a single action would be required. By adding variables, actions

are turned into operators. This has the benefit of vastly simplifying the encoding of

permissible actions, while incurring the small inconvenience of having to check the

validity of what is "bound" to the variable (for example, it must be verified that ?t is a

particular truck such as t-1).

The operator `load-truck` requires that the truck and package be at the same location, indicated by the two at preconditions sharing the same location `?loc`. When this operator is valid, meaning its preconditions are met, it can be "applied" to a world state, causing a transformation. Application of the load action transforms the state by adding the fact that the package is in the truck, `(in-truck ?p ?t)`, and removing the fact that the package is at `?loc`. For example, if the world state indicates that `truck-1` is a truck and `p-1` is a package, and furthermore that they are at the same location LVI "`(at t-1 LVI)(at p-1 LVI)`", then the resulting world state from applying the action is "`(in-truck p-1 t-1)`" with the "`at`" pertaining to the package removed.

Solving a planning problem: Now armed with a means of representing facts about the world, and a means of representing actions that transform this state, it is possible to make a solution to the planning problem. Recall that a solution is a sequencing of actions that transform the initial state into the goal state. For this small example, the initial and goal states are as specified in **Figure 2.1**, and the operators are those defined in **Figure 2.2**.

But where should the search process begin? It is possible to begin from the specified initial state, try each action that has its preconditions met, advance the world state one action at a time, and cease the search process when a world state is reached that is consistent with the goal state (that is, a state containing all facts specified, and potentially others not relevant to the solution). Similarly, one can "work backwards" from the goal state by adding an action that adds facts needed in the goal state, and continue to regress the state backwards until a state consistent with the initial state is reached. Both the forward and backward "state-space" approaches can create valid solution sequencing of

23

actions, and indeed can even be used simultaneously, but they do a lot of blind and "fruitless" searching because of needlessly committing to a totally ordered sequence of actions before there is justification to commit to that order (that is, in the forward and backward state-space search, each action has one and only one "correct" place in the generated solution; even two steps that could in theory happen before one another, or simultaneously, would be needlessly committed to a particular order). The benefit of these two approaches is one always has an explicit notion of "world state" – that is the set of facts that are true before and after applying each action is known. However, not only is there much wasted effort, but this total ordering goes against the aim of creating solutions that commit to an ordering only in as much as that ordering is necessary for correctness. At this point, the motivation for a "partial ordering" of actions is clear. By moving the search process from the "state space" to the "plan space", researchers in the field of automated planning opened the possibility of creating flexible solutions. This partial-order approach is also referred to as "least-commitment" planning because constraints are added only when they are absolutely necessary for correctness.

Partial-order planning begins by first adding all actions that have as effects those facts needed in the goal state. These actions are only ordered relative to one another if an incorrect solution would be generated without this ordering. The process is then continued by asserting actions that achieve the unsupported preconditions of actions already added to the plan, and ordering them to preserve correctness. The search process is complete when all facts required in the goal state are supported, and the precondition of every action added to the "partially ordered plan" is supported by the effect of another action or the initial state.

At this point, the search process has produced a set of steps, and a partial ordering across them (for example, two steps that can occur simultaneously will not have an ordering relationship between them). Any sequencing of those steps that respects the ordering across them is guaranteed to be a valid solution to the planning problem. What follows in Table 2.1 is a trace of how the partial order process would proceed given the initial and goal states specified in **Figure 2.1**, and the operators defined in **Figure 2.2**. For clarity, sometimes multiple steps in the process are collapsed into a single step, and also not all details are shown. Furthermore, an explanation of the notation used is omitted, instead relying upon an informal description of each step in the process. Full details of the process, and its notation, are presented in Section 3.2.2. The intent here is to give a "feel" for the underlying planning approach, and the type of knowledge I sought to represent and use with the technique created for and defended in this dissertation. A careful explanation is in Chapter 4.

Partial planning process	Explanation

Steps:
  S0
  SGoal
Ordering:
  S0 < SGoal
Links:
Flaws:
  (at p-1 LehighU)@SGoal
	Add special start and end steps, and an ordering constraint between them. The start step has as effect all facts of the initial state; the goal step has as precondition all facts specified in the goal state. All steps that are added have an implicit ordering constraint forcing it to come between the initial and goal steps.

Add step:
  Su: (unload p-1 t-1 LehighU)
Add link:
  Su →(at p-1 LehighU)@SGoal
Add order:
  S0 < Su
  Su < SGoal
	Add a step that unloads p-1 from t-1 at LehighU. Keep track of the contribution of the step by "linking" it with the precondition of the goal step, and removing the associated flaw. Ensure order consistency by making step Su come between the initial and goal steps.

Flaws:
  (at t-1 LehighU)@Su
  (in-truck p-1 t-1)@Su
	The new step has preconditions that need to be supported. Two important ones are shown, the rest omitted for clarity

Add step:
  Sd: (drive t-1 LVI LehighU)
Add links:
  Sd →(at t-1 LehighU)@Su
  S0 →(at t-1 LVI)@Sd
Add order:
  Sd < Su
	Add a step that drives t-1 to LehighU, and link its effects with the preconditions that need support. Note that if there were many locations, it is completely valid to drive the truck from a different location. An "omniscient" choice was made to use the location at which the truck began.

Flaws: (in-truck p-1 t-1)@Su	The "at" precondition of the unload is removed from flaws; note again that had we not supported the at of Sd using the initial state, we would have to add this as a flaw.

Add step:
  Sl: (load p-1 t-1 LVI)
Add links:
  S0 →(at t-1 LVI)@Sl
  S0 →(at p-1 LVI)@Sl
  Sl →(in-truck p-1 t-1)@Su
Add order:
  Sl < Sd
	Add a step that loads p-1 into t-1 at LVI, and use its effects to support Su. Again, for clarity links supporting Sl from the initial state were added, but these preconditions could have been supported by adding another step (note that this makes the search space infinite). Constrain Sl to come before Sd.

Final partial-order plan:
Steps:
  S0, SGoal, Su, Sd, Sl
Links: (those shown above)
Ordering:
  S0 < SGoal, Sd < Su, Sl < Sd
	The process is complete, because all preconditions are supported, and any sequence consistent with the ordering will transform the initial state into the goal state. While in this case, this produces a total order, one can see that with two trucks and two packages, one will likely exploit the partial-order.

Table 2.1 A trace of a plan-space search process for the transportation logistics domain

Use of expert knowledge: So far, however, the process shown in Table 2.1 has no way to take in to account expert knowledge or preferences on how to find a "good" solution. For instance, when the load step was added, the planner faced two choices on how to support the precondition that the truck t-1 be at location LVI: use the "at" fact listed in

26

the initial state, or add a new action with an effect that supports the precondition. Being an omniscient human (relative to this tiny problem), I chose to use the fact from the initial state, as it would make the example shorter. But how would a planner know to do so? It is precisely this sort of expert knowledge or search preference that the approach presented in this dissertation addresses. To represent this knowledge, the HIEPPR-POP approach presented in this dissertation would express a refinement rule like the following:

If there is a step Sx that requires a truck be at a certain location
And if there is another step Sy that can come before Sx, and provides this fact
Then use the effect of Sy to support Sx

Figure 2.3 Example rule to express the expert knowledge used in **Table 2.1**

It is also worth noting that, while the example in Table 2.1 started from "scratch" (that is, an initial plan containing only the "special" start end goal steps), the HIEPPR-POP approach can start from a previous solution (that is, start with a plan that contains steps, ordering constraints, and links pertaining to a similar problem).

2.3 Research Challenges

The work presented in this dissertation is motivated by existing research in domain-configurable planning. In this related form of planning, domain-specific knowledge enhancing the action schemas is given. This knowledge is used to guide the planning process, which like first-principles planning generates a plan from scratch. Domain-configurable planners have been shown to solve problems more quickly and to scale much better with problem size than first-principles planners, as reported in Section 3.5.

There are four challenges that need to be addressed to make domain-configurable plan adaptation feasible. First, is the need to define a representation formalism capable of

encoding domain-specific plan adaptation knowledge. The representation formalism must be flexible enough to be able to represent the various kinds of transformations that may occur in a plan while having clear syntax and semantics. Furthermore, the representation must allow for rapid plan reuse. There are a number of representation formalisms that have been used successfully for speed-up control including macro-operators (i.e., [Botea et al., 2005]), hierarchical task network methods [Hogg et al., 2007; Koening et al., 2005; Nau et al., 2001], temporal rules based on first-order logic (i.e., [Baccus & Kabanza, 2001] and [Kvarnström & Magnusson, 2003]) and domain-configurable rules [Lee-Urban & Munoz-Avila, 2009]. One important aspect of this challenge is that the representation formalism will fix, or at the very least restrict, the plan generation paradigm on which the plan adaptation will be based. For example, the structure of the macro-operators in Botea et al. (2005) ties it to a total-order planner because these operators modify the current state as maintained by such planners. It is interesting to observe that aside from domain-specific plan adaptation algorithms such as CHEF [Hammond, 1986], most domain-independent plan adaptation algorithms do not represent adaptation knowledge explicitly because, as explained before, the expansion of the baseline plan is done by the first principles planner. Exceptions such as the PbR [Ambite and Knoblock, 2005; 2001] and DCPOP [Lee-Urban & Munoz-Avila, 2009] fall in the category of domain-configurable planners that combine encoding of domain-specific knowledge in a domain-independent setting [Wilkins & desJardins, 2001].

The second challenge is to develop an adaptation algorithm that correctly interprets and efficiently executes the knowledge encoded in these expressions in a way that preserves quality guarantees that are implicitly encoded in the source plans but that may not be

explicitly encoded into the knowledge base used for plan generation/adaptation. Generation of plans that take into account some explicit measure of quality has long been observed to be an important consideration during plan generation [Perez & Carbonell, 1994]. For example, planners like Graphplan [Blum and Furst, 1997] ensure that the length of longest branch of the partial-order representation of the plan generated is the minimum. Hence, assuming all actions have equal execution time, these plans will result in the overall shortest length when each branch of the plan is executed in parallel. Other planners associate costs with each action and take these costs into account to generate plans of some quality as a function of the costs of actions in the plan (e.g., [Marthi et al., 2008; Do and Kambhampati, 2002]).

Benchmarking of plan quality has become an important metric in the International Planning Competition (IPC). A particular challenge arises from the observation that plan adaptation techniques can be detrimental to plan quality if it is not carefully done. For example, if the plan adaptation technique attempts to retain as much of the source plan as possible then it is possible the transformed plans are unnecessarily costly. That is, if the plan adaptation technique commits to be conservative as defined in Nebel & Koehler (1995), then it is conceivable that the solution plan obtained for the target problem may be significantly more costly than one generated from scratch and even much larger that the source plan.[1] Interestingly, Fox et al. (2006) defines a notion of plan stability, loosely related to conservative plan adaptation, that results in solution generation with comparable plan quality versus first-principles planning. A challenge related to solution

---

[1] Taking cost considerations aside, generating a solution plan that retains as many steps as possible from the source plan is computationally harder than planning from scratch (Nebel & Koehler, 1995).

quality in my approach is to consider situations in which these quality measures are not explicit in the domain (i.e., the set of operators) used to generate plans and, instead, the measures are implicitly encoded in the source plans. For example, Myers (2006) points to meta-theoretical measures that describe semantic properties of the domain and use these to identify qualitatively similar plans. For instance, one plan encodes a defensive strategy whereas another one may encode a flanking maneuver.

The third challenge is to test my research hypothesis that domain-configurable plan adaptation algorithm with appropriate knowledge is scalable and retains a significant portion of input plans that are consistent with the conditions of the new problem.

The fourth challenge is to formulate an abstract problem that captures the essence of domain-configurable plan adaptation and to perform complexity analysis on this abstract problem. It is likely that the computational complexity of domain-configurable plan adaptation is greater than that of existing plan adaptation algorithms. This trade-off of increased scalability at the cost of increased computational complexity is reminiscent of how first-principles planning compares to HTN planning. Under some conditions the former can be EXP-SPACE-complete whereas the latter can be undecidable [Ghallab et al., 2004], which is a precise way to say that HTN planning can be incredibly more difficult (unsolvable) than planning from first-principles (solvable). However, this does not mean that the goal of attaining scalable plan adaptation is not attainable. Rather, it means that appropriate domain-specific knowledge must be encoded in order to realize performance gains. As an example, the HTN planner SHOP can be shown to be Turing-complete because it can make calls to external programs to evaluate conditions. Thus, it is possible to express even undecidable problems in the SHOP planning formalism.

30

However, with the right knowledge encoded into it, SHOP has been shown consistently to scale well with problem size [Nau et al., 2005], solving problems several orders of magnitude larger than those solvable by first-principles planners.

The important question of how to learn the plan adaptation knowledge is outside the scope of this dissertation, but is an excellent avenue of future work that would likely broaden the impact of my approach.

3   Related Work – Planning & Other Approaches

In this chapter I discuss related work and introduce the notation of the basic planning elements as presented in the literature.

Planning is among the oldest fields of AI and as such has seen a wide variety of scientific contributions. This section presents a portion of this history in order to better situate my particular contribution to the field. This is not intended to be a comprehensive overview of research on planning, but is instead intended to discuss related research as it pertains to this dissertation (aside from the ones already discussed in previous sections and pointed discussions in subsequent sections).

3.1   Classical Planning and STRIPS Assumptions

The birth of the field of planning, in the context of AI, was the STRIPS system, short for Stanford Research Institute Problem Solver in 1971 [Fikes & Nilsson, 1971]. This system name became synonymous with all "classical" automated planning approaches, referred to interchangeably as planning from first principles, generative planning, STRIPS planning, and classical planning.

Classical automated planning is a special case of the graph search problem. A graph is a structure defined by its nodes and edges: nodes represent a point in the problem space, and edges represent connections between the nodes. For example, one can imagine a railroad system as a graph where the nodes are rail stations, and edges are the railway tracks. In (state-space) classical planning as graph search, the nodes are the world states, and the edges are actions that can be taken from that state (in plan-space classical

planning, nodes are partial plans and edges are plan refinements. This distinction will be clarified later.). A world state is a symbolic description (an alphabet is a set of symbols, like "a", "b", "c"; words are symbols constructed from the alphabet. "Symbolic" is to say a syntax composed of symbols) of facts that are known about the world. For example, a fact about a person named "Steve" at a restaurant called "Tulum" could be represented as "(at Steve Tulum)". An action is an operation that adds and removes facts from a node in a consistent fashion. The planning problem is a symbolic description of the initial and goal states (nodes), and a set of operators (actions/edges) that transform states in a well-defined way. A solution to the planning problem, therefore, is a sequence of actions (a plan) that transforms the initial state into one of a set of goal states. In graph search terms, it is a sequence of edge-node-edge transitions that form a path from the specified start node to the desired end node (eg: Philadelphia train station to the Atlanta train station, with all the stations and paths in between). However, even for very simple problems, it is infeasible to explicitly represent all the nodes and edges of the graph – often, the number of states in the problem outnumbers even the largest estimates of the number of particles in the universe [Ghallab et al., 2004]! Therefore the various techniques researched in automated planning center on implicit, compact representations (logic-based formalisms) of the graph that allow for efficient algorithmic search.

The dominant formalization of the classical planning problem is derived from the STRIPS system [Fikes & Nilsson, 1971] which has had such a far-reaching impact on the field that the terms "classical planning" and "STRIPS planning" are frequently used interchangeably. Mathematically, a planner is a deducing machine. This machine works by applying a set of inference rules to its input. Both the inputs and rules are governed by

a logical system referred to as First Order Logic, or first-order predicate calculus. In this formalization, the domain description taken as input for classical planning is a state-transition system described by the triple $\Sigma = (S, A, \gamma)$, where S is a finite set of states, A is a finite set of actions, and γ is a state-transition function defined by $\gamma : S \times A \rightarrow S$. Each of these elements are summarized in **Table 3.1**, which is condensed from [Ghallab et al., 2004]. In this triple, the set of all possible states S is not represented explicitly. Instead, a state is represented as a finite set of grounded atoms from first-order logic (see Section 3.1.1 for details). Therefore a finite set of predicate symbols and a finite set of constants allow for the enumeration of all possible states, and these two sets are inputs to the planning system and used to calculate S as needed.

Symbol	Name	Meaning
Σ	Sigma, state-transition system	A formalization that models all reachable states and valid actions for a particular planning domain
S	Set of states, S	A finite set of states, each of which is a set of symbolic facts. These fact sets encode that which is currently held as true in the problem.
A	Set of actions, A	A finite set of actions. An action a has the form (*head*, *pre*, *neg*, *pos*), where *head* is a name plus arguments (e.g. (putdown a)), *pre* are the preconditions that govern when the action is "applicable" in a state (e.g. (holding a)), *neg* are the negative effects of the action, which are those facts to be removed from the state to which the action was applied (e.g. (holding a)), and *pos* are the positive effects of the actions, which are those facts to be added to the state on which the action was applied (e.g. (ontable a)).
γ	Gamma, the state-transition function $S \times A \rightarrow S$	A function that defines, for each state S and action A, the state reached by applying A to S

Table 3.1. Elements of the domain description input to classical planners

Similar to S, the set of actions A and the state-transition function γ are not represented explicitly. Instead, inputs to the planning system are "action schemas," or "operators,"

which have the form $o = (o^{head}, o^{pre}, o^{neg}, o^{pos})$. The head of the operator, o^{head}, is a predicate (a name, and zero or more terms, e.g. "(pickup ?a table)" where pickup is the predicate name and there are two terms. The first, "?a" indicates a variable which can be substituted for any constant, and the second term "table" is a grounded constant) and the preconditions o^{pre}, negative effects o^{neg}, and positive effects o^{pos} are conjunctions of atoms whose terms appear in o^{head}. Operators are encoded in this fashion so that a single schema can describe many similar actions by using variables (indicated by being prefaced with a question mark) as the terms to the predicate. An operator is said to be "grounded" when all of the terms of the operator are constants; an operator of this form represents one specific action in the domain. An action a is said to be "applicable" to a state s when s entails all of the atoms appearing in a^{pre}, written as $s \models a^{pre}$. When an applicable action a is applied to a state s, a new state s' is created. This new state s' consists of all the atoms appearing in s, minus all atoms appearing in a^{neg}, and with the addition of all atoms appearing in a^{pos}, written as $s' = (s \setminus a^{neg}) \cup a^{pos}$. If a positive effect is already in s, it is not added; if a negative effect is not in s, it is simply ignored. Any atom not mentioned in the effects is assumed to remain unchanged (called the "STRIPS assumption"). Thus, the state-transition function $\gamma(s, a)$ is s' if action a is applicable in state s, and undefined otherwise. A plan π is a linearly ordered sequence of actions $\pi = \langle a_1, a_2, \ldots, a_k \rangle$; because the head of the action uniquely identifies the operator of which it is as specialization, only the head of action is typically used in the enumeration of the sequence.

Finally, a classical planning problem is a triple $P = (\Sigma, s_0, g)$, where Σ is a classical planning domain, $s_0 \in S$ is the initial state of the problem, and g is a set of goal atoms that must appear in the final state when the actions in a solution plan are applied starting from

state s_0. That is, a solution to problem P is a plan π such that the state $s' = \gamma(\gamma(\ldots, (\gamma(s_0, a_1), a_2), \ldots), a_k)$ satisfies the goals in g. It should be noted that s_0 is specified using the "closed world assumption," which assumes that any atoms not explicitly asserted as true are false. Put succinctly, a solution plan is a linearly ordered collection of actions that when applied, and assuming the actions perform as modeled, achieve the input objective.

In classical planning, there are several other restricting assumptions imposed on the state-transition system, Σ. These are made in order to simplify the problem which, in its classical planning form, is in the P-SPACE complexity class [Ghallab et al., 2004]. These other assumptions state that Σ has the following properties, as summarized in [Ghallab et al., 2004]:

- Σ has a finite set of states.
- Σ is fully observable, meaning the planner has complete knowledge about the state of Σ.
- Σ is deterministic, meaning the application of an applicable action transitions Σ to a single state.
- Σ is static, meaning the system stays in the same state until an action is applied.
- Goals are explicitly specified as a goal state, or set of states, in Σ. A planner's objective is therefore to construct a sequence of state transitions that ends in one of the goal states. This means that goals involving avoiding states, constraints on state trajectories, or utility functions are not handled in classical planners.
- A plan solving a problem for Σ is a linearly ordered, finite sequence of actions.
- Time is implicit in Σ, meaning actions have no duration and cause an instant state transition in Σ.

By representing actions, states, and the transition functions in this way, the planning-as-graph-search problem that could potentially contain millions (or more!) of nodes and edges is transformed into a compact representation where individual actions and states are generated as needed.

The following is a brief, traditional example used to illustrate the planning notation and planning process within a classic domain called "blocks world". Blocks world is the

familiar game of stacking blocks played by children. The world consists of blocks starting in an arbitrary (but specified) configuration. The world can be modified by either picking up a block, or putting the one that is held down. The goal is to arrive at a pre-stated "goal" configuration of blocks.

The process begins by formally defining the problem domain Σ, which is necessary for defining the planning problem *P*. Recall that for convenience, and without loss of generality, the domain is specified by a set of operators. In one representation of blocks world, the following are the two operators used to change the world state:

(:head (unstack ?x ?y) :(pre ((clear ?x)(on ?x ?y)(not (ontable ?x))) :(neg (on ?x ?y)) :(pos (clear ?y)(ontable ?x)))	(:head (stack ?x ?y) :(pre ((clear ?x)(clear ?y)(ontable ?x))) :(neg ((clear ?y)(ontable ?x))) :(pos (on ?x ?y)))

Figure 3.1 One set of operators to define a simple "blocks world" domain

The operator "unstack" has two terms, or parameters, and abstracts the action of taking a clear block (one that has no block on top of it) off of another, and putting it on the table. It is applicable in a state *s* where there is nothing stacked on top of the block indicated by the variable ?x (codified as "(clear ?x)"), block ?x is on the block indicated by variable ?y ("(on ?x ?y)"), and block ?x is not already on the table ("(not (ontable ?x))"). If substitutions can be made for ?x and ?y that unify with the current state (that is, a constant value for ?x and ?y are found such that the positive predicates in the precondition conjunction are in the world state, and the negative predicates – those preceded by "not" – do not appear in the world state), then the operator's negative effects are removed from the state, and the positive effects are added. That is, the fact that ?x is on ?y is removed from the world state, and the facts that ?y is now clear and ?x is on the

37

table are added. The operator "stack" does the reverse of unstack, by placing a block that is on the table onto another block that is clear. Note how the formalization of these two operators are carefully coordinated by the truths that they require, assert, and retract.

Now that the planning domain Σ is defined, one can create a planning problem P within this domain. The problem is specified by augmenting the domain with an initial state s_0 and a goal state g. The following are a valid initial and goal state description:

(:initial (on b c)(on a b)(ontable c))	(:goal (on a b)(on b c))

Figure 3.2 One valid initial and goal state description for the domain in Figure 3.1

Having presented the restricted state-transition system in which classical planner operates, Section 3.2 discusses how this space can be searched for a solution. The next subsection first carefully defines the building-blocks of classical planning.

3.1.1 Definitions for Classical Planning

As is common in the field of automated planning, I adopted a representation that is based on first-order logic. This formal representation allows one to express and assign meaning to strings of characters, which ultimately allows one to model elements of the real world. What follows are formal definitions of the language and concepts described above.

Definition 3-1. A **constant** symbol is syntactically composed from a unique sequence of one or more alpha-numeric characters, which are, by convention, the upper and lowercase versions of the 26 letters in the English alphabet, the digit characters '0' through '9', the dash character '-' and the underscore character '_'), for example 'table' or 'truck-1' (the single quotes are not part of the symbol); a constant is therefore a string semantically used to refer to a specific object in the problem being modeled.

Definition 3-2. A **variable** is a symbol that can be used in the place of a constant, akin to the concept of variables in algebra. As is common in the automated planning field, I follow the syntactic convention that variables begin with a question mark, followed by a sequence of one or more characters. Two examples of variables are '?x' and '?truck' (again, the single quotes are not part of the symbol).

Definition 3-3. A **term** is either a variable or a constant.

Definition 3-4. A particular variable is referred to as **bound** (past tense of 'bind') if and only if there is an assignment of the variable to a term. For example, if the variable '?x' were to be assigned to the constant 'table', then one would say that ?x is bound to table. When a variable is bound to a constant, the variable is said to be **grounded** (a reference to the Earth's surface, I believe).

Definition 3-5. A **predicate name**, or **predicate symbol**, is a character sequence having the same syntax as a constant. However, rather than referring to an *object* in the modeled world, a predicate name is used to semantically refer to a *relation* in that world. Some examples of predicate names are 'on', 'at', and 'in-city'. Predicate symbols also have an **arity**, which indicates the number and names of terms taken as arguments by that predicate. For example on ?x a has a predicate symbol named 'on' of arity 2, the first term is a variable named ?x and the second is a constant symbol a.

Definition 3-6. An **atomic formula**, or **atom**, is a statement of fact about the modeled world. It is syntactically formed by an opening parenthesis symbol '(' followed by a predicate symbol, followed by a space separated list of terms equal in number to the predicate's arity, followed by a closing parenthesis symbol ')'. The space separated list of

terms is referred to as **arguments** or **parameters**. If all arguments are grounded, then the atom is also grounded. The following is a comma separated list of example atoms – the comma is not part of the syntax: (on ?x table), (in package1 truck1), (in-city Lehigh Bethlehem). Each of the predicate symbols on, in, in-city have an arity two. The only argument that is a variable is ?x, the rest are constant symbols.

Definition 3-7. A **substitution** is a collection of variable bindings. When a substitution is **applied** to an atomic formula *a*, a new atom is created by replacing each of the variables in *a* with the term to which it is mapped (if such a mapping exists in the substitution).

Definition 3-8. A **world state** is a finite set of grounded atoms. Semantically, a world state is an assertion about all facts that are true. By convention, any atom not appearing in a world state is assumed to be false (the so-called **closed-world assumption**).

Definition 3-9. A **tuple** is an ordered list of elements; an **n-tuple** is an ordered list of *n* elements, where *n* is a non-negative integer. For example (a, b, c, d, e) is a 5-tuple.

Definition 3-10. An **action** *a* is defined by a 4-tuple (*head*, *pre*, *neg*, *pos*). The first element of the tuple, the action **head**, has syntax similar to that of an atom: an opening parenthesis, an exclamation point followed by one or more characters, followed by a space separated list of constants, ended with a closing parenthesis. An example action head with a single argument is (!putdown block-a). The exclamation point follows convention, and makes it easier to distinguish between atomic formulas and the heads of actions. The remaining elements of the tuple are the action's **preconditions** (*pre*), **negative effects** (*neg*), and **positive effects** (*pos*), each of which are finite sets of ground atoms. Additionally, any parameter appearing in one of these atoms must appear in the

head of the action. The purpose of preconditions and effects are described in the next definition.

Definition 3-11. An action *a* is **applicable** to world state *s* if and only if all atoms in the precondition set of *a* are members of *s*. The applicable action *a* can be **applied** to *s* to create a new world state *s'*. The new state *s'* is a copy of *s* with all atoms appearing in the negative effects of *a* removed, and all positive effects of *a* added. If a positive effect of *a* is already in *s*, it is not added; if a negative effect of *a* is not in *s*, it is simply ignored. Any atom not mentioned in the effects is assumed to remain unchanged (called the "STRIPS assumption").

Definition 3-12. An **operator** has the same syntax and semantics as an action (Definition 3-10), with one difference: the arguments appearing in the head may be variables (and therefore the atoms in the *pre*, *neg*, and *pos* sets may also use variables).

Definition 3-13. A **classical planning domain**, or **classical domain description**, is defined by the 3-tuple $\Delta = (C, P, O)$, where C is a finite set of constants, P is a finite set of predicates, and O is a finite set of operators. A constraint on Δ is that any atom appearing in O must also be a member of P; similarly, any constant appearing in O must be a member of C.

Note that this definition of a classical planning domain is slightly different, but equivalent to, that presented in section 3.1. In that discussion, a domain was defined as a state-transition system described by the triple $\Sigma = (S, A, \gamma)$, where S is a finite set of states, A is a finite set of actions, and γ is a state-transition function defined by $\gamma : S \times A \rightarrow S$. The set of all possible atoms for a given domain can be generated by applying all

41

combinations of constants in C to the predicates in P. The power set of these generated atoms forms the finite set of states S. The set of actions A can be derived from the set of operators O by computing all instantiations. Finally, the state transition function γ can be derived from the definition of applying actions to world state. I adopt the notation used for defining Δ because it is closer to how DCPOP and other planners are implemented (as opposed to the definition of Σ, which is more general than the classical approach).

Definition 3-14. A **classical planning problem** is a triple $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, s_0 is a finite set of ground atoms describing the **initial world state** of the problem, and g is a finite set of atoms that define the problems **goals**. All atoms appearing in s_0 and g must be derivable from C and P. Also, s_0 is specified using the "closed world assumption," which assumes that any atoms not explicitly asserted as true are false.

Definition 3-15. A **plan** $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ is a linearly ordered, finite sequence of actions; because the head of the action uniquely identifies the operator of which it is a specialization, only the head of an action is typically used in the enumeration of the sequence.

Definition 3-16. A plan $\pi = \langle a_1, a_2, \ldots, a_k \rangle$ is a **solution** to a classical planning problem Ψ if and only if each action in π is an instantiation of an operator in O with constants from C, and furthermore only if the result of applying all of the actions in sequence starting from s_0 yields a world state containing at least all those atoms appearing in the problem's goal set g. That is, a solution to the classical planning problem is a sequence of

actions (a plan) that transforms the specified initial state into one of a set of states containing all the specified goals.

World state	A **world state** is a symbolic description of facts that are known about the world.
Action schema, or operator	An action is an operation that adds and removes facts in a consistent fashion.
Operator $o = (o^{head}, o^{pre}, o^{neg}, o^{pos})$	The head of the operator, o^{head}, is a predicate (a name, and zero or more terms, e.g. "(pickup ?a table)" where pickup is the predicate name and there are two terms. The first, "?a" indicates a variable which can be substituted for any constant, and the second term "table" is a grounded constant) and the preconditions o^{pre}, negative effects o^{neg}, and positive effects o^{pos} are conjunctions of atoms whose terms appear in o^{head}
Domain description, "domain"	The **domain description** taken as input for classical planning is a set of operators that transform world states in a well-defined way.
Planning problem	The planning problem is an initial and goal states, along with a domain description
Solution to planning problem	A solution to the planning problem is a sequence of actions (a plan) that transforms the initial state into one of a set of goal states.

3.2   Refining Incomplete Plans – State Space vs. Plan Space Planning

How a planning algorithm represents and explores Σ, the state-transition system in which the solution (plan) is searched, determines whether it is a plan-space or a state-space algorithm. A state-space algorithm searches for a plan in a graph whose nodes consist of world states, and whose edges are applicable actions that transform the world state. World states are descriptions of what is true in the problem, and are represented by first-order logic literals as described in the previous section. A plan-space algorithm searches for a solution in a graph whose nodes consist of partial plans, and whose edges are

applicable refinements to those partial plans. The approach presented in this dissertation makes use of the plan-space representation.

3.2.1   The State Space – Total Order Planning

The simplest algorithm for solving a classical planning problem constructs a search graph where the root, or start node is the initial state. Children, that is nodes that are a single edge away, of the root are generated by applying each applicable action (edges), in order to compute the resulting state (child node). The ***state-space*** algorithm continues expanding the graph until a state (node) is reached that contains all the goals in *g*; the solution plan extracted is the actions that form the edges leading from root to the satisfying node. There exist many search control strategies for guiding which node is next explored and preventing and detecting loops in the search process, such as depth-first, breadth-first, and iterative deepening. Heuristics, which define alternative means of exploring the graph to those mentioned previously and are covered in more detail in the next section, can also be introduced to further constrain the search process, and are responsible for many of the most recent breakthroughs in planning research. Additionally, it is possible to work backwards from the goals in a so-called "means-ends" approach, which can be more efficient, or to combine forward and backward state search strategies. Implicit in state-space approaches is a strict linear ordering of the actions in the plan on account of the graph's structure; hence, state-space algorithms produce totally-ordered, sequential plans.

3.2.2 The Plan-space – Partial-Order Planning (POP)

Plan-space algorithms in contrast, like the ones created for this dissertation, search in a graph where nodes are incomplete (partial) plans and edges are refinements applied to, or removed from, the incomplete plan. That is, instead of searching for a solution in a graph where nodes consist of world states and edges are actions transforming the state, a plan-space approach searches over the set of all possible plans [Penberthy & Weld, 1992]. The solution is not extracted from the sequences of edges in the graph (as done in state-space), but is instead derived from the partial-plan stored within a terminal node of the graph (the plan-space). Each node in the graph represents a "candidate set" of plans, and each edge splits the search space into disjoint candidate sets [Kambhampati, 1997]. Some examples of refinements are the addition of actions to the partial plan, or the addition of ordering constraints between actions. When searching in the space of plans, the ordering of actions is deferred until essential to resolving a flaw (e.g. unsatisfied applicability condition of an action). In least commitment planning, where action ordering is deferred, solution plans are only partially ordered – this means actions can be flexibly interleaved during execution. Any linearization consistent with this ordering (a topological sort) is a solution to the classical planning problem. This particular approach to plan space planning, sometimes referred to as "partial order causal link (POCL) planning", is best described in the works describing the SNLP (McAllester & Rosenblitt, 1991), UCPOP (Penberthy & Weld, 1992), and VHPOP (Younes & Simmons, 2003) planners; unless otherwise stated, when I refer to "partial-order planning" in this dissertation, it is with the POCL formulation of the plan-space problem in mind.

Partial-order planning (POP) was the dominant planning paradigm some 20 years ago because of its ability to flexibly interleave actions, rather than totally order them, while solving problems. POP drops the classical requirement for actions to be totally ordered, which is particularly useful for plan adaptation (e.g., [Ihrig & Kambhampati, 1996], [Muñoz-Avila & Weberskirch, 1997]). However, interest in POP waned when other paradigms such as analysis of planning graphs and more recently planning with heuristics, demonstrated significant gains in planning speed and solvable problem size. More recently, there has been a revival of POP as heuristic methods have been developed that perform comparably to other state-of-the-art first-principles planners. Researchers have pointed out the importance of POP planning for real-world domains because in many real world situations actions can be performed in parallel and the planner should not commit to step orderings, such as the extreme case a total-order sequence, unless necessary (e.g., [Knoblock, 1994], [Paulokat & Wess, 1994], [Nguyen & Kambhampati, 2001], [Vidal & Geffner, 2006]).

Like state-space planning, the partial order planning process requires inputs which are the same as those defined in Section 3.1 (inputs are action schemas and a symbolic initial and goal state specification of the problem). Next an ***initial partial plan*** is created, consisting of two special steps s_0 and s_∞, and an ordering constraint that forces s_0 to come before s_∞ in any solution plan. The step s_0 represents the initial state of the problem; s_0 has no preconditions and its effects are the atoms in the initial state. The step s_∞ represents the problem's goals; s_∞ has no effects and its preconditions are the atoms in the goal state.

Partial-order planning refines this initial partial plan by adding constraints and plan steps, ordered between the two initial steps, until a complete partial-order plan is obtained. A partial-order plan is *complete* if it has no flaws (flaws are defined in the next paragraph). A *partial-order plan* is defined as a 4-tuple *(S, →, →$_{CL}$, B)* of sets of *POP plan elements.* *S* is the set of plan *steps*, which represent the application of actions in the plan. The set → contains the *ordering constraints* between plan steps, which take the form *s* → *s'*, indicating that step *s* must be executed before step *s'*. The set →$_{CL}$ contains the *causal link constraints*, *s* →$_p$ *s'*, indicating that the precondition *p* needed by the action in step *s'* is produced as the effect of the action in step *s*. The set *B* indicates variable *binding constrains*, *?x* ≠ *?y* or *?x* = *?y*, indicating that whenever variable *?x* occurs in the plan it must take a different (respectively the same) value as the variable *?y* (the notation "*?x*" represents that *x* is a variable symbol). Set *B* is empty when planning without variables (i.e. "grounded").

The elements of a partial-order plan are summarized below.

partial-order plan	a 4-tuple *(S, →, →$_{CL}$, B)* of sets of **POP plan elements**
Plan steps: the set *S*	*S* is the set of plan **steps**, which represent the application of actions in the plan.
Ordering constraints: the set →	*The set of **ordering constraints** between plan steps, which take the form s → s', indicating that step s must be executed before step s'.*
Causal link constraints: the set →$_{CL}$	Set →$_{CL}$ contains the **causal link constraints**, *s →$_p$ s'*, indicating that the precondition *p* needed by the action in step *s'* is produced as the effect of the action in step *s*.
Binding constraints: the set **B**	set *B* indicates variable **binding constrains**, *?x ≠ ?y* or *?x = ?y*, indicating that whenever variable *?x* occurs in the plan it must take a different (respectively the same) value as the variable *?y* (the notation "*?x*" represents that *x* is a variable symbol). Set *B* is empty when planning without variables (i.e. "grounded").

Table 3.2 Summary of partial-order plan elements

There are two kinds of **flaws** in POP: open preconditions and causal threats. An **open precondition** occurs when a step *s'* in the plan has a precondition *p*, written *p@s'*, for which no causal link *s →$_p$ s'* exists. A **threat** occurs when a causal link *s →$_p$ s'* and a step *s''* exist such that *s''* has as an effect the negation of *p* (i.e., ¬*p*), written *s'' →$_{¬p}$*, and *s''* can consistently occur between *s* and *s'*, written *s'' || (s →$_p$ s')*, in a linearization of the plan. A **linearization** of a plan is a sequencing of all steps in a manner consistent with the ordering constraints such that, for every two steps *s* and *s'*, *s* will always be listed before *s'* if either *s →$_p$ s'* or *s → s'* hold (e.g. a topological sorting).

There are four possible **POP plan refinements**, each of which adds an element to the 4-tuple *(S, →, →$_{CL}$, B)* that defines the plan: adding steps, adding ordering constraints, adding causal links, and adding binding constraints. Ordering links and binding constraints are added to solve causal threats. A causal link *s →$_p$ s'* is added to satisfy an

open condition $p@s'$ when $s \rightarrow_p$ holds. The step s might be an existing step in the plan or a new one added to satisfy this open condition.

The POP planning process starts from the initial partial plan representing the classical *planning problem*, defined in Section 3.1 as a triple $P = (\Sigma, s_0, g)$, where Σ is a classical planning domain, $s_0 \in S$ is the initial state of the problem, and g is a set of goal atoms. As in classical planning, the solution to the problem is a sequencing of actions that, when applied, transform the initial state into a state that entails the goals (i.e., the state contains all goals). Whenever a new step s is added to the plan the ordering constraints $s_0 \rightarrow s$ and $s \rightarrow s_\infty$ are added to the plan. The objective of the POP planning process is to refine the initial partial-plan into a complete partial-order plan. Any linearization of this complete partial-order plan is a solution to the planning problem. It is notable that there may be many valid linearizations of the partial-ordered plan and thus it represents a set of solutions to the problem (because the actions can be interleaved in any manner consistent with their partial order).

initial partial plan	consists of two special steps s_0 and s_∞, and an ordering constraint that forces s_0 to come before s_∞ in any solution plan
The step s_0	The step s_0 represents the initial state of the problem; s_0 has no preconditions and its effects are the atoms in the initial state.
The step s_∞	The step s_∞ represents the problem's goals; s_∞ has no effects and its preconditions are the atoms in the goal state.
Flaws	Problems in a partial-plan that need resolution before the plan is "complete"; there are two types, open preconditions, and threats
Flaw: open preconditions	*open precondition* occurs when a step *s'* in the plan has a precondition *p*, written *p@s'*, for which no causal link $s \rightarrow_p s'$ exists. In the initial partial plan, all the preconditions of s_∞ are the only flaws in the plan.
Flaw: Threats	A *threat* occurs when a causal link $s \rightarrow_p s'$ and a step *s''* exist such that *s''* has as an effect the negation of *p* (i.e., $\neg p$), written $s'' \rightarrow_{\neg p}$, and *s''* can consistently occur between *s* and *s'*, written *s'' || (s $\rightarrow_p$ s')*, in a linearization of the plan.
POP plan refinements, "flaw resolutions"	There are four possible *POP plan refinements*, each of which adds an element to the 4-tuple *(S, $\rightarrow$, $\rightarrow_{CL}$, B)* that defines the plan: adding steps, adding ordering constraints, adding causal links, and adding binding constraints. Ordering links and binding constraints are added to solve causal threats. A causal link $s \rightarrow_p s'$ is added to satisfy an open condition *p@s'* when $s \rightarrow_p$ holds. The step *s* might be an existing step in the plan or a new one added to satisfy this open condition.
Complete partial-order plan	A partial-order plan is *complete* if it has no flaws
Linearization of a partial-plan	A *linearization* of a plan is a sequencing of all steps in a manner consistent with the ordering constraints such that, for every two steps *s* and *s'*, *s* will always be listed before *s'* if either $s \rightarrow_p s'$ or $s \rightarrow s'$ hold (e.g. a topological sorting).

Table 3.3 Summary of terms used in partial-order planning.

Partial-order planning is an attractive framework because its least commitment property makes it amenable to (1) interleaving planning and execution, (2) performing information gathering, and (3) handing resource and time constraints [Nguyen & Kambhampati, 2001]. At the same time the flexibility provided by its least commitment strategy can be detrimental to its performance. In this regard it is noteworthy that, as far

as the author is aware, until now no scalable domain-configurable partial-order plan adaptation algorithm exists.

3.3   Search Control Knowledge

Search control knowledge can both be built into the planner, and taken as input by the planner. In either case, it is used to guide the search process by determining which edge-node pair is next selected in the graph representing the search space. There are three broad ways of making this selection: domain-independent strategies, domain-specific strategies, and heuristics (which confusingly can also be domain-independent or domain-specific). A planner that takes domain-specific control knowledge as input is termed domain-configurable.

Domain-independent selection strategies are ones that are the same regardless of the domain. An example in the partial-order planning process is to resolve threats before any other flaw type. That is, if there are two edges leaving a node, one denoting a refinement that resolves a threat and the other being a refinement that resolves an open condition, the edge that resolves the threat will be explored first.

Domain-specific selection strategies are tightly coupled to a particular problem domain. An example in a problem domain involving the transportation of packages to destinations would be to always explore an edge that moves a package closer to its final destination.

Heuristics are a means of assigning a mathematical evaluation of costs to the choice points in a search space. For example, rather than always resolving a flaw that is a threat

first to the exclusion of every other edge, a heuristic would estimate the expected utility of following each edge, and select the one that appears to be the "best guess".

3.4 Heuristics in Planning

Many planners use heuristics to estimate how difficult it is to find a solution from a particular node in the search space. Heuristics are one way of tackling the overwhelming size of the problem. A good heuristic is inexpensive to compute and closely approximates the actual cost to the solution, which allows the most promising paths to be explored first. Because of the successes of using heuristics in planning (for example, FastForward and Graphplan are two very successful planners that have dominated the field for some time solely through the use of good heuristics), as well as the success of domain-configurable algorithms (for example the SHOP planner, which is one of the most successful planners used by industry), the HIEPPR-POP approach uses of both techniques to control the adaptation process.

Where does this domain-independent search control come from? There exist many domain-independent strategies to refine partial-order plans. For example, some simple strategies call for solving all open conditions first and then solving the threats. Other, more sophisticated strategies strike a balance between resolving the causal threats and the open conditions [Pollack et al., 1997]. POP planning using these strategies has been empirically shown to perform slower than other more recent first-principles planners such as those that use planning graphs or newer heuristics also based on planning graphs. Nevertheless, some heuristics that were originally conceived for total-order planning have been successfully ported for POP and have resulted in a significant increase of

performance. These heuristics estimate the cost of reaching a complete plan from the current partial plan. Such estimates are used to compare competing refinements with A* cost functions [Hart et al., 1968] or other heuristic cost functions [Younes & Simmons, 2003]. Planning graphs have been used to estimate the cost of refining the current partial plan into a complete plan [Blum & Furst, 1997]. The heuristic estimate is obtained by removing all negative effects from the actions and using planning graphs to obtain a solution for this relaxed problem (e.g., [Hoffmann & Nebel, 2001]). A similar process has been developed for POP planning with adjustments to account for the fact that POP does not maintain an explicit world state, whereas totally-ordered planners (like FastForward) do [Nguyen & Kambhampati, 2001]. An example of this is the heuristic POP planner VHPOP (for Versitaile Heuristic Partial Order Planner)[Younes & Simmons, 2003], which has also been used to build another heuristic POP adaptation planner [van der Krogt & de Weerdt, 2005] and a probabilistic conformant POP planner [Onder et al., 2006].

3.5   Domain Configurable Planning

There are a host of approaches to solving the classical planning problem. The one created for this dissertation focuses on one form in particular: domain-configurable, partial-order (plan space) plan adaptation.

A domain-independent planner is one that takes as input a symbolic description of the initial and goal states, as well as knowledge about the problem domain. This knowledge is encoded as action models (operators) that specify applicability conditions for each action, as well as the effects of the action on the state of the world; that is to say, action models encode how to compute a state transition in Σ. What separates domain-configurable planners from domain-independent planners is that, in addition to receiving

all inputs required by the domain-independent planners, hand-crafted domain-specific control knowledge is also taken as input. This control knowledge is additional information that is tightly coupled to the problem domain and is used by adaptive and generative algorithms to guide the exploration of the search space. For example, in hierarchical task network planning (HTN) the control knowledge is encoded by non-primitive methods, which are used to reason about the search space at a level higher than the action models (this is further explained in the HTN Planning sub-section). The adaptation algorithm presented in this dissertation goes beyond the classical planning problem by taking hand-crafted adaptation control rules as input.

Domain-configurable planning is a form of planning in which domain-specific knowledge enhancing the action schemas is given. This knowledge is used to guide the planning process which, like first-principles planning, generates a plan from scratch. Domain-configurable planners have been shown to solve problems more quickly and to scale much better with problem size than first-principles planners. For example, in most domains used in the International Planning Competition (IPC), a first-principles planner would solve up to a dozen of goals in the same amount of time that a domain-configurable planner would solve several dozens of goals. Furthermore, for these domains, first-principle planners are not able to solve problems with hundreds of goals in any reasonable amount of time, while their domain-configurable counterparts can. Because of their scalability, their increasing number of applications [Nau et al., 2005], and their capability to drop classical planning assumptions, domain-configurable approaches are believed to be closing the gap between academic research in AI planning and real-world applications [Nau, 2007]. One of the most successful domain-configurable

approaches is HTN planning, which uses a knowledge structure called methods to drastically constrain the search process; more details on HTN planning are presented in the next sub-section.

Competing approaches have been developed for domain-configurable planning, including variants of HTN planning [Nau et al., 1999; 2005], temporal logic planning [Bacchus & Kabanza, 2000; Kvarnström, & Doherty, 2001; Kvarnström & Magnusson, 2003] and term-rewriting rules [Ambite et al., 2001; 2005]. These competing approaches have in common the ability to scale performance with increasing problem size and clear semantics [Kvarnström et al., 2000; Bacchus, and Ady, 2001; Kuter & Nau, 2005]. They differ in the presentation of the domain-configurable knowledge and how it is used. Some use HTN constructs to encode domain-specific knowledge and HTN planning to use this knowledge. Others use temporal logic rules and modify a first-principles planning process to prune nodes in the search space as indicated by the given rules. Temporal logic (TL) rules are used in TLPlanner and TALPlanner in a forward state-space search process. These rules use temporal logic modal operators that express relationships between a state and subsequent states, and are used to define a prune function that detects and cuts unpromising nodes (i.e., states) in the state space. For example, a rule might encode the following condition: "Do not move a block if its current position is consistent with the goal configuration". If the planner reaches a state S where one such block has been moved from a previous state in which its location was consistent with the goal configuration, then the rule will trigger and prune S from the search space.

The PbR (Planning by Rewriting) system [Ambite et al., 2001; 2005] uses term rewriting rules to transform partial-order plans solving a problem into other partial-order

plans solving the same problem but with better quality. While PbR does transform partial-order plans, it is not partial-order planning, which is the process of refining partial-order plans to solve POP flaws and that begins from the initial plan and ends in a complete plan. Instead of plan refinement as done in partial-order planning, PbR can be seen as circumscribed to navigate in the subspace of complete partial-order plans.

3.5.1 HTN Planning

In Hierarchical Task Network planning, the planning system formulates a plan by recursively decomposing tasks (symbolic representations of activities to be performed) into smaller subtasks until primitive tasks are reached that can be executed. Primitive tasks represent concrete actions to be undertaken to achieve the high-level tasks. The basic idea was developed in the mid-1970s [Sacerdoti, 1975] and expanded into large-scale systems in the 1980s [Wilkins, 1988]. The formal semantics were developed later [Erol et al., 1994]. A domain description in HTN planning consists of an action model and a task model. The action model consists of a set of planning operators that describe the actions the plan executor can perform directly. Operators have the same form as in the classical planning problem: *(:operator h p e)*, where *h* is the primitive task being performed (called the head of the operator), *p* are the preconditions that determine the applicability of the operator, and *e* are the effects of applying the operator. The task model consists of methods that describe possible ways of decomposing tasks into subtasks. Methods have the form *(:method h p st)*, where *h* is the nonprimitive task being decomposed (called the head of the method), *p* are the preconditions that determine the applicability of the method, and *st* is the collection of subtasks.

The particular variant of HTN planning most closely related to the approach presented in this dissertation is Ordered Task Decomposition (OTD) [Nau et al., 1999]. This is the variant used in SHOP and resulted in significant gains in runtime. In Ordered Task Decomposition, an ordered list of nonprimitive tasks is given as input. The first task in the list is decomposed into subtasks, and the first of these subtasks is further decomposed. The decomposition process recursively continues until a primitive task is obtained. In this situation, an operator applicable to the primitive task is applied. Planning continues with the next task until all nonprimitive tasks are decomposed into primitive tasks, which in turn have been achieved with applicable operators. The solution plan consists of the collection of all primitive tasks in the order they were generated. The task hierarchy linking the tasks in the HTN problem description with the primitive tasks in the solution plan is a hierarchical task network (HTN) and it is said that the HTN entails the plan. For a detailed presentation please refer to [Nau et al., 1999].

3.6 Case-Based Planning

Case-based planning (CBP) is the intersection of planning and case-based reasoning (CBR). In CBR, previous problem-solving episodes are reused to solve new, similar problems. These episodes are stored as cases consisting of a description of the problem, its solution, and additional annotations. A variety of representations are used in the cases, depending upon the class of problems addressed (e.g. classification, configuration, planning). Formally, CBR can be described as a four step process: (1) the retrieval of relevant cases to a new problem, (2) the reuse of a previous solution in a case to the new

problem, possibly through adaptation, (3) the revision of the new solution after simulation or execution, and (4) finally retaining the solution as a new case.

When CBR is used for automated planning, then the technique is CBP. Plan adaptation primarily addresses the *reuse* and *revision* phase of CBP, and is a topic central to CBP research [Cox et al., 2006]. In this dissertation I focus on plan adaptation and will not address the *retrieval* nor *retention* elements of CBP. However, important relationships exist between computing the similarity measures during case retrieval to assure adaptable solutions and performing effective adaptation during case reuse [e.g., Lopez de Mántaras et al., 2006]. Further details of CBP are omitted from this section; for an in-depth review of research in case-based plan adaptation please refer to [Munoz-Avila and Cox, 2008].

3.7   Learning and Planning Control Knowledge

Research on learning and planning has concentrated for the most part on attaining speed-up gains, such as learning macro-operators (e.g., [Mooney, 1988; Botea *et al.* 2005]) and work on learning search control knowledge (e.g., [Mitchell *et al., 1986*; Etzioni, 1993; Katukam & Kambhampati, 1994; Minton, 1988; Fern *et al*., 2004]). Techniques have been proposed for learning action models (e.g., [Martin & Geffner, 2000; Winner & Veloso, 2003]), and learning hierarchical task methods [Hogg and Munoz-Avila, 2007; Choi & Langley, 2005; Reddy and Tadepalli; 1997, Ruby and Kibler, 1991]. Although most of these works fall in the category of speed-up learning; the learned knowledge is used to generate the same plans that could be generated by a first-principles (non-adaptive) planner.

3.8 Plan Adaptation

Plan adaptation has been the subject of ongoing research interest for easily twenty years, thanks to the belief of some in the research community that plan adaptation is easier than planning from scratch in many situations. One of the first systems to gain fame most notably for its approach to the plan adaptation problem (as part of the CBP problem) was CHEF [Hammond, 1986]. This system focused on the Szechwan cooking domain, and used domain-specific plan adaptation rules to repair flaws in cooking recipes, which were the plans in this system. Unlike domain-configurable planners, CHEF was an algorithm hand-crafted to its domain, relying upon the human expert's plan modification rules. Systematicity and completeness were not guaranteed.

Plan adaptation is a problem-solving method in which existing plans are modified consistently with given action schemas to solve new problems. Over the years there has been a significant research effort on this problem-solving technique. Works include complexity analysis for worst case scenarios [Nebel & Koehler, 1995], search-space analysis [Au et al., 2002; van der Krogt & de Weerdt, 2005; Kuchibatla & Munoz-Avila, 2006], plan merging [Veloso, 1994; Munoz & Weberskirch, 1997; Ram & Francis, 1996; Tonidandel & Rillo, 2005] and plan adaptation algorithms for several planning paradigms including total-order planning [Veloso, 1994], partial-order planning [Ihrig & Kambhampati, 1997; Munoz-Avila & Weberskirch, 1996], HTN planning [Kambhampati, 1994], planning graphs [Gereveni & Serenia, 2000], heuristic planning [Gerivini & Serina, 2009; van der Krogt & de Weerdt, 2005; Tonidandel & Rillo 2002], and Ordered Task Decomposition planning [Warfield et al., 2007 & Ayan et al., 2007]. Part of the reason for this continuing interest in plan adaptation is attributable to studies

indicating potential applications, which include military planning [Mitchell, 1997; Veloso et al., 1997; Munoz-Avila et al., 1999], computer gaming [Ontañón et al., 2007; Sanchez et al., 2007], manufacturing [Costas & Kashyap, 1993; Munoz-Avila & Weberskirch, 1996; Veerakamolmal & Gupta, 2002], route planning [Haigh et al., 1997], medicine [Schmidt et al., 2001, 2003; Salem et al., 2003], and bioinformatics [Jin et al., 2009].

A common feature of how these planners operate is that they combine two steps: (1) **baseline plan generation** and (2) **first-principles expansion** of the plan obtained in (1). The first step obtains a plan, called the **baseline plan**, based on the **source plan** (i.e., the plan to be adapted). Multiple techniques have been proposed for the first step including removing elements in the source plan that are not mentioned in the **target problem** (e.g., the adjust-plan step as in [van der Krogt & de Weerdt, 2005]) and replaying the decisions that derived the source plan in the context of the target problem (e.g., derivational replay as in [Veloso, 1994]). The second step simply passes the resulting plan from the first step to a first-principles planner to obtain a solution plan for the target problem or a sub-problem of the target problem. These steps can be performed as a batch process whereby once the baseline plan generation step finishes, the first-principles expansion is performed until a complete solution of the target problem is obtained (as in [Warfield et al., 2007]) or it can be interleaved as in [Gerivini & Serina, 2009] where the steps are performed on subplans of the source plan at different time intervals.

The interaction between these two steps partly explains why plan adaptation algorithms have consistently demonstrated performance improvements over the corresponding first-principles planning on which they are built including total order planning [Veloso, 1994], partial-order planning [Ihrig & Kambhampati, 1997], planning graphs [Gerevini &

Serenia, 2000], and heuristic planning [Gerivini & Serina, 2009; van der Krogt & de Weerdt, 2005; Tonidandel & Rillo, 2002]. Informally, the underlying first-principles planner is still doing the "leg work" in the first-principles expansion and the input plans are giving a "jump start" as a result of the baseline plan generation step.[2] This explanation is formalized in Au et al. (2002) and Kuchibatla & Munoz-Avila (2008).

The first-principles expansion step is largely uninformed about plan quality issues for most existing systems. For example, van der Krogt & de Weerdt (2005)'s system uses heuristics to estimate the number of refinements needed to obtain a solution using a relaxation of the domain obtained by removing all negative effects of the operators (this heuristic was first implemented in [Hoffman & Nebel, 2001]). Therefore, this procedure aims at finding any solution as quickly as possible. More recently, researchers have started looking into plan quality aspects during plan adaptation. For example, Fox et al. (2006) proposes an algorithm that successively improves the quality of the adapted plans while still preserving much of the baseline plan. The plan quality measurements are explicitly encoded in the plan generation knowledge and, hence, first-principles plan generation can produce plans of similar quality albeit requiring more CPU time as demonstrated in their empirical evaluation.

3.8.1 Derivational Analogy and Transformational Analogy

Plan adaptation algorithms can be classified between "derivational analogy" and "transformational analogy" [Carbonell, 1986]. ***Derivational analogy*** systems store the

---

[2] In Veloso (1994) the baseline plan generation and first-principles expansion steps are interleaved. Derivational replay is used to generate the baseline plan. The expansion step can be viewed as "gluing" together baseline plans.

sequence of derivations that led to the source plan rather than the source plan itself. For instance, in planning episodes found in a derivational system, the addition of a particular plan step would be augmented with information about failed alternatives. It was first developed in Prodigy/Analogy [Veloso & Carbonell, 1993] on a total order planner. The system interleaved the baseline plan generation and plan expansion to combine multiple subplans. So the plan expansion steps could be seen as "gluing" together the baselines subplans obtained after replaying the derivations relative to the target problem. Derivational analogy was also developed for partial-order planning in DerSNLP [Ihrig & Kambhampati, 1994] & CAPlan/CbC [Munoz-Avila & Weberskirch, 1996]. An example of derivational analogy in HTN planning would be Repair-SHOP [Warfield et al., 2007].

Transformational analogy systems use the source plan directly during adaptation. In transformational analogy the stored solutions are previously generated plans, nothing more; these plans can be partially or totally-ordered, depending upon the underlying planner used in the system. Transformational systems have been developed for partial-order hierarchical planning [Kambhampati & Hendler, 2002], partial-order planning [Hanks & Weld, 1995], heuristic planning [van der Krogt & de Weerdt, 2005], and ordered task decomposition planning [Warfield & Munoz, 2007; Goldman & Kuter, 2008]. Most of the recent plan adaptation systems fall in the transformational analogy category, including RePlan [Boella & Damiano, 2002], Fox et al, (2006), Serina & Gerivini (2009), and Tonidandel & Rillo (2005). A potential reason for this trend is because of the higher engineering requirements for derivational approaches, as pointed out by Cunningham et al. (1994). However this point is not conclusive because in derivational analogy, a first-principles planner can be used to generate the sequence of

derivations automatically. Nevertheless, transformational systems tend to be considered more practical for industry purposes, in spite of the clear flexibility advantages enjoyed by those that are derivational. The domain-configurable adaptation technique explored in this dissertation does not presume a derivational trace and as such is akin to a transformational approach.

3.8.2 Plan Repair

Some of the works mentioned such as Fox et al. (2006) and Warfield & Munoz (2007) present themselves as plan repair systems. In plan repair, a plan is being executed and during the execution a mismatch occurred between the conditions expected to be true in order to execute the next action and the actual state of the world. At this point the system needs to adapt the remaining actions to be executed to the changing circumstances. In other words, plan adaptation can be viewed as a plan repair problem where conditions changed before the first action of the plan was executed, which is how Fox et al. (2006) tested their system. Analogously, one could say that plan repair is a plan adaptation problem relative to the remainder of the plan not yet executed, which is how Warfield & Munoz (2007) tested their system.

3.8.3 Adaptation Frameworks

One of the most robust frameworks for understanding transformational plan adaptation, SPA (for Systematic Plan Adaptor), was reported in [Hanks & Weld, 1995]. In that work, the problem of systematic plan adaptation was formalized as the process by which refinements made by a classical, generative partial-order planner (e.g., addition of an action, addition of ordering constraint) are *retracted*. This could also be read as a

process by which the refinements made by a first-principle, first-order, non-adaptive, plan-space planner, are the elements that are considered for removal – i.e. removing steps and removing orderings. In the search space of partial plans, the root node is the null plan, and children are all one-step, systematic partial-order plan refinements (step, ordering, or binding constraints). A retraction, therefore, removes such a refinement by adding the current node's parent and sibling nodes to the fringe of the search tree (that is, travel up the edge from child to parent, and enqueue all nodes reachable by every other edge). In this sense, refinements move downward in the search space and retractions move up. To ensure systematic search, nodes added to the fringe are tagged as "up" or "down," indicating that only retractions or refinements, respectively, are applied to the node. **Figure 3.3**, taken from [Hanks & Weld, 1995], illustrates this process. Later, Kuchibatla (2006) presented TransUCP, a framework that extended SPA to work with any classical planner. However, the idea of retraction and refinements remains the same.
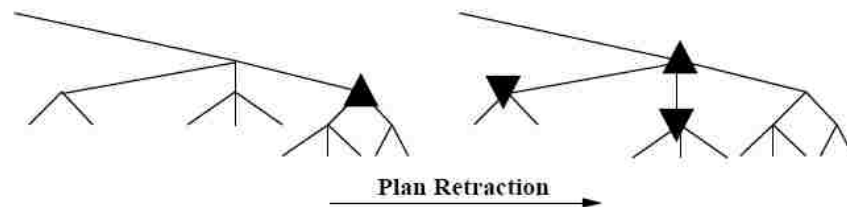
Figure 3.3 Plan retraction as defined (and depicted) in SPA Figure from [Hanks & Weld, 1995], and later in TransUCP [Kuchibatla, 2006] and [van der Krogt & de Weerdt, 2005].

Recent progress in adaptive planning was reported in [van der Krogt & de Weerdt, 2005], where a new process of domain-independent adaptation was shown to beat planning from scratch in each of their benchmarks. At the core of the technique is the computation of "removal trees". Intuitively, these trees modify an input partial-order plan

64

by combining several retractions at once. Instead of performing a single retraction, the causal links involving an action are used to iteratively grow a tree of actions that are removed from the plan. In **Figure 3.4**, an example removal tree (of size three, its height) is depicted; circles indicate the elements of the plan that remain after the tree is removed. A full discussion of how to create these trees is outside of the scope of this document. However, a casual explanation is as follows. They compute the tree step-wise. First they remove a single step. If a correct plan cannot be made from this new plan, they then additionally remove the children of the removed step. If a solution cannot be made from this new plan, the process is repeated, removing the children of the children, until a solution can be produced. Worst case, this results in removing all steps from the plan and generating a solution from scratch.
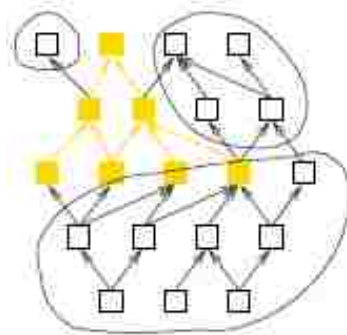
Figure 3.4 A removal tree, as depicted in [van der Krogt & de Weerdt, 2005]. Circles indicate the parts of the partial-order plan that remain after the tree is removed.

The approach described by van der Krogt et al & de Weerdt can be seen as a special form of adaptation as performed by TransUCP. Abstracting away the means by which a retrieved plan is adjusted to a new problem (prior to adaptation), one can depict the two frameworks as shown in **Figure 3.5**. The retraction phase in van der Krogt's work begins with removal trees of size zero (i.e., no retraction). Then, for all partial plans in the new

65

set, a generative planner is called until success or failure is reported. If a new plan could not be found, the size of the removal trees is increased by one and the process repeats. In TransUCP, on the other hand, retraction is unguided, aside from the restriction that removals must occur in the same way that a generative planner might have made the refinements. Also, for any given partial plan in the set to be searched, any number of refinements can be made. The same is true for retractions in TransUCP. In this dissertation, I explore the ability to encode adaptation techniques that guide the "retraction" phase in the right side of Figure 3.5 (e.g. to compute removal trees).
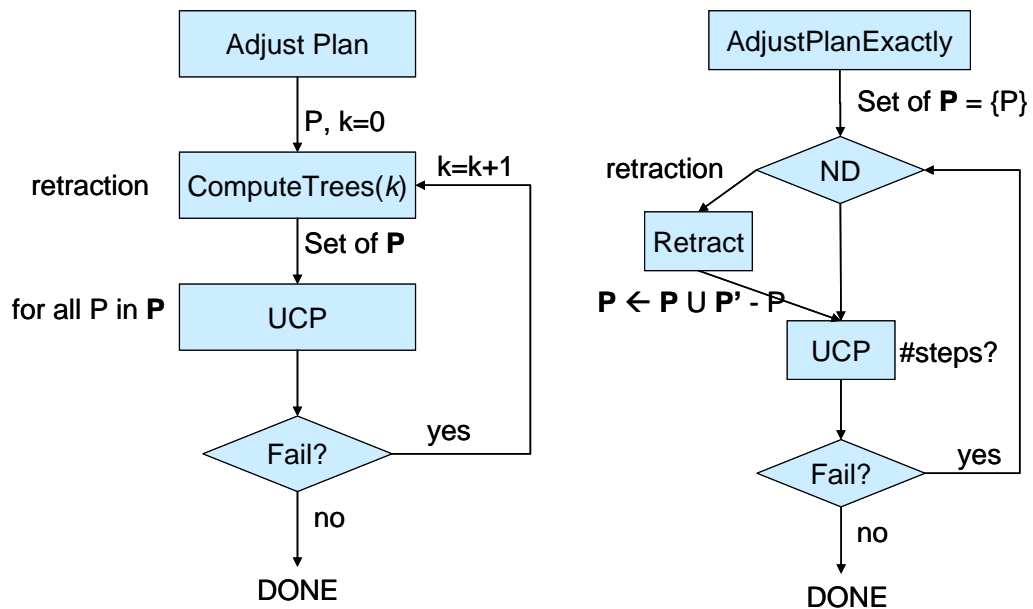
Figure 3.5 Two flow diagrams depicting two forms of domain-independent plan adaptation. On the left is adaptation as performed by [van der Krogt & de Weerdt, 2005]. On the right is adaptation as performed by [Kuchibatla & Munoz-Avila, 2006]. In this figure, UCP stands for any classical, generative planner.

4   Partial Order Planning With Refinement Rules

The main goal of this dissertation is to address the problem of capturing high-quality, hand-crafted plan refinement and plan adaptation knowledge. This chapter presents the core of the algorithm I created to address this problem, and an analysis of its properties. The material presented in this chapter relies heavily on the algorithms and notation presented in Chapter 3, and in particular the formalizations of classical planning described in Section 3.1.1, and the formalization of partial-order planning explained in Section 3.2.2.

Partial-order planning is an attractive framework because its least commitment property makes it amenable to (1) interleaving planning and execution, (2) performing information gathering, and (3) handling resource and time constraints [Nguyen & Kambhampati, 2001]. At the same time the flexibility provided by its least commitment strategy can be detrimental to its performance. In this regard it is noteworthy that, as far as I am aware, until now no scalable domain-configurable partial-order plan adaptation algorithm exists.

The remainder of this chapter is organized as follows: the theory of partial order planning used by the approach presented in this dissertation is formally defined (the previous explanation was informal and general). After that, I introduce the main knowledge representation formalism I created and used for my approach – "refinement rules". Next, the motivating example from Chapter 2 is revisited more formally in order to re-situate the refinement rules. Following that, I describe the algorithm for capturing

and using the rules. Finally, the motivating example is reworked in order to show the theory of refinement rules in action.

4.1 Partial-order Planning Definitions

This chapter relies upon the theory of classical planning, and partial-order planning in particular. The definitions and notation essential to these theories, which are strongly influenced by [Ghallab et al., 2004, and Weld 1994], are presented in the next two subsections (please refer to the previous chapter for a less formal, intuitive presentation of each). This particular approach to plan space planning, sometimes referred to as "partial order causal link (POCL) planning", is best described in the works describing the SNLP (McAllester & Rosenblitt, 1991), UCPOP (Penberthy & Weld, 1992), and VHPOP (Younes & Simmons, 2003) planners; unless otherwise stated, when I refer to "partial-order planning" in this dissertation, it is with the POCL formulation of the plan-space problem in mind.

4.1.1 Partial-order planning

This dissertation adopts a specific approach to plan space planning, sometimes referred to as "partial order causal link (POCL) planning", that is best described in the works describing the SNLP (McAllester & Rosenblitt, 1991), UCPOP (Penberthy & Weld, 1992), and VHPOP (Younes & Simmons, 2003) planners; unless otherwise stated, when I refer to "partial-order planning" in this dissertation, it is with the POCL formulation of the plan-space problem in mind.

In state-space search for a solution plan, where the nodes in the graph explored are world states and edges between nodes are applicable actions that transform the origin

node to the destination node, there are both explicit and implicit commitments being made by the planner. The explicit commitments made are (1) the addition of an action to the solution plan, and (2) the restriction of precisely when the action is to be executed relative to all the other actions in the plan. What is being implicitly committed to are (1) substitutions for operator parameters that yield the ground action added to the solution, and (2) the justification for how the preconditions of an action are supported – that is how each atom in the action's precondition set is guaranteed to be an element of the state immediately preceding the application of the action. Partial-order planning search algorithms refine the state-space algorithms by making each of the state-space decisions explicit, and furthermore, by only committing to a decision if it is *absolutely necessary* to ensure the correctness of the returned solution (the so-called "least-commitment" property).

The key insight for understanding partial-order planning is the idea of plan "flaws", which are derived from making all state-space search decisions explicit. Consider the classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, s_0 is a finite set of ground atoms describing the initial world state of the problem, and g is a finite set of atoms that define the problems goals. As before, the solution to this problem is a sequence of steps that, when applied starting from the specified initial world state, and assuming actions perform as modeled, result in a terminal state containing at least all the goals specified in g. Rather than performing the search for a solution by reasoning on the applicability of actions to world states (state-space search), the search is performed by reasoning on the applicability of resolution strategies to errors (flaws) in candidate partial-order plans (plan-space search). These flaws guarantee the candidate is

not (yet) a solution to the problem. Search terminates when a candidate is generated with no flaws remaining, resulting in a complete partial-order plan, any linearization of which is ensured to be a solution to the classical planning problem.

What are the errors in a partial-plan that prohibit it from being a solution? The answer, somewhat surprisingly, amounts to only two flaw types that invalidate the guarantee that a step's precondition will exist in the state immediately preceding the execution of that step: this first flaw type centers on *which* step is to provide (assert) the needed precondition, and the second flaw type centers on *when* the step must occur.

This first flaw type is referred to as an 'open condition' or 'unsupported precondition'. Intuitively, this reflects the situation where a step s has a precondition p that has no specified provider. Resolving a flaw of this type amounts to committing to the provider of p by either using the initial state (if p is an element), reusing a step in the plan that asserts p as an effect, or adding a new step to the plan having p as an effect and ordering it to occur before s. This resolution approach respects the 'least commitment property' in that, by definition, a step is only executable if all of its preconditions appear in the state immediately preceding its execution. Hence, at a minimum, any search process to generate a solution to a planning problem must guarantee that all preconditions of every step in a candidate plan be established at *some point* in the plan (that is, *which* step provides the necessary condition).

The second flaw type also has to do with guaranteeing that a step's precondition will exist in the state immediately preceding the execution of that step. However, rather than having to do with committing to a provider of a precondition (as is the case with an open condition flaw, which ensures that a precondition is provided as an effect by some step in

70

the plan), this second flaw type, called a 'threat' has to do with committing to an ordering of steps (that is, *when* the step should occur). Why is handling open condition flaws insufficient to ensure the correctness of a candidate solution plan?

Consider the following: two steps, unordered relative to one another, have been added to a partial-plan, in order to resolve two open conditions. The first step s_1 asserts p for s_x, and the second step s_2 negates p for some other step. Without any constraints on the ordering of s_1 and s_2 relative to each other, it would be possible to attempt to execute s_1 (assert p) followed by s_2 (delete p), followed by s_x (which requires p). This would have the unintended and harmful result of breaking the commitment that s_1 provide p for s_x, and yield an incorrect (not executable) plan, because s_2 has removed p from the world state prior to the execution of s_x. This is why s_2 is said to 'threaten' the effect provided by s_1 for s_x. Resolving a flaw of type threat amounts to making a (now necessary) commitment to the ordering of s_1, s_2, and s_x (by either 'promoting' s_2 by forcing it to occur after s_x, or by 'demoting' s_2 by forcing it to occur prior to s_1), or making a (now necessary) commitment to the binding of variables such that the effects of s1 and s2 are not the same (a 'separation').

Resolving these two flaw types are a necessary and sufficient condition to guarantee the correctness of the generated partial-order plan. Because all flaws must be resolved, and once a flaw has been repaired it never occurs again, least-commitment partial-order planning amounts to making commitments on *how* to resolve flaws. The selection of *which* flaw to repair has a profound effect on the time taken to find a solution, but is not a backtrack point, so long as all resolvers to a repaired flaw are added to the search frontier and the repaired plan is removed from it. This approach of flaw resolution not only

guarantees that all solutions returned by the search process are correct (so-called 'soundness'), but also that all solutions appear as leaves in the search graph (so-called 'completeness') and that no plan is ever considered twice in the search process (so-called 'systematicity').

The definitions of the above terms, and a presentation of the least-commitment partial-order planning algorithm appear below.

Definition 4-1. A plan **step** is an instantiation of an operator (each parameter of the operator has a substitution to a term) from the input classical planning domain (Definition 3-13), or the special partial-order planning "initial step", or the special partial-order planning "goal step" (see the next two definitions).

Definition 4-2. The **initial step**, often written as s_0, is a special non-executable step that all partial-order plans contain; it is used for representing the initial state of a given classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, s_0 is a finite set of ground atoms describing the initial world state of the problem, and g is a finite set of atoms that define the goals of the problem. It is constructed as the instantiation of a "dummy" operator having no preconditions, and a set of positive effects that contains all the atoms appearing in the problem's initial world state, and a set of negative effects that, while typically left empty for computational time and space efficiency, is semantically understood to contain all atoms in the domain that are not elements of the set of positive effects. While the initial step is never to be executed, it is always constrained to be the very first step of a partial-order plan.

Definition 4-3. The **goal step**, often written as s_∞, is a special non-executable step that all partial-order plans contain; it is used for representing the goal state of a given classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, s_0 is a finite set of ground atoms describing the initial world state of the problem, and g is a finite set of atoms that define the problems goals. It is constructed as the instantiation of a "dummy" operator having no positive nor negative effects, and a precondition set that contains all the goal atoms appearing in g. While the goal step is never to be executed, it is always constrained to be the very last step of a partial-order plan.

Definition 4-4. An **ordering constraint** is a relation between two steps in a plan p. It takes the form $s_i < s_j$ where s_i and s_j are steps, and it semantically means that s_i must appear before s_j in any linearization of p.

Definition 4-5. A **binding constraint** is a relation between a variable v and term t. It can take two forms: (1) $v = t$, which semantically means that v is assigned to be equal t (also called a **codesignation constraint**), or (2) $v \mathrel{!=} t$, which semantically means that v is constrained to never take the value t (also called a **non-codesignation constraint**).

Definition 4-6. A **causal link** is a relation between an atom a_i appearing in either the positive or negative effects set of a step s_i and an atom a_j appearing in the precondition set of a step s_j. The atoms a_i and a_j must be equivalent, and s_i must be constrained to come before s_j. Syntactically, a causal link is written $s_i \rightarrow_a s_j$. Semantically, a causal link reflects that the effect a_i of s_i is being used to **support**, or **establish**, the precondition a_j of s_j. One can read a causal link as "s_i causes a to become true for s_j"

Definition 4-7. A **partial-order plan** is a 4-tuple $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$ of sets of POP plan elements (Definition 4-14), where: S is a set of steps, $\rightarrow$ is a set of ordering constraints involving only elements appearing in S, $\rightarrow_{CL}$ is a set of causal links involving only elements appearing in S, and B is a set of binding constraints (empty when planning without the use of variables).

I do not list flaws (Definition 4-9) as part of the tuple defining a partial-order plan because flaws are derived from the elements 4-tuple; however flaws are often included in the plan data structure for speed efficiency reasons, and I also refer to flaws as 'plan elements' (Definition 4-14) when not ambiguous.

Definition 4-8. A **linearization** of a partial-order plan $(S, \rightarrow, \rightarrow_{CL}, B)$ is a totally-ordered sequence of the steps contained in S that is consistent with the ordering constraints contained in $\rightarrow$. This is equivalent to a topological sort of the ordering constraints.

Definition 4-9. A partial-order plan may contain **flaws** of only two types: open preconditions (Definition 4-10), and threats (Definition 4-11).

Definition 4-10. A flaw of type **open precondition** occurs when a step s_j in a partial-order plan has a precondition p, written $p@s_j$, for which no causal link $s \rightarrow_p s_j$ exists. In the initial partial-order plan, all the preconditions of s_∞ are the only flaws in the plan.

Definition 4-11. A flaw of type **threat** occurs when a causal link $s_i \rightarrow_p s_j$ and a step s' exist in a partial-order plan such that s' has as an effect the negation of p (i.e., $\neg p$), written $s' \rightarrow_{\neg p}$, and furthermore s' can consistently, relative to the ordering constraints, occur between s_i and s_j, written $s' \, || \, (s_i \rightarrow_p s_j)$, in a linearization of the plan. This is a flaw because, without resolution, it would be possible to create a linearization in which one or

more preconditions needed for a step are not available in the world state immediately preceding the application of the threatened step.

Definition 4-12. There are two ways to **resolve an open precondition flaw** $p@s_j$ in a partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$: (1) *Operator instantiation:* add to S a new step s_r that has an effect that unifies with p, add the ordering constraint $s_r < s_j$ to the set $\rightarrow$, and finally add the causal link $s_r \rightarrow_p s_j$ to the set $\rightarrow_{CL}$. Alternatively, (2) *Step reuse:* non-deterministically select a step s_r from S such that s_r has an effect that unifies with p, and s_r can, relative to the ordering constraints, be consistently ordered to occur before s_j and; given this s_r, add the ordering constraint $s_r < s_j$ to the set $\rightarrow$, add finally add the causal link $s_r \rightarrow_p s_j$ to the set $\rightarrow_{CL}$.

Definition 4-13. There are three ways to **resolve a flaw of type threat** $s' \,||\, (s_i \rightarrow_p s_j)$: **promotion**, **demotion**, and **separation**. Promotion resolves the threat by adding $s_j < s'$ to the set of ordering constraints, if doing so does not violate the consistency of the ordering constraints (does not introduce a cycle). Demotion resolves the threat by adding the ordering $s' < s_i$, with the same restriction about not violating the consistency of the ordering constraints. Separation resolves the threat by adding a binding constraint b, where b does not violate the consistency of the set of binding constraints B, that prevents any effect of s' from unifying with $\neg p$ to the set of binding constraints (this is only possible when planning with variables), and adding two ordering constraints that force s' to come between s_i and s_j (for systematicity). For each of the three flaw resolutions, a resolution is only applicable if its modification to the partial-plan does not violate the consistency of the ordering constraints, nor the binding constraints.

Definition 4-14. Given a partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$, and the set of flaws F extant in ρ, a **POP plan element** refers to a member of any one of the set of steps, set of ordering constraints, set of causal links, set of binding constraints, or flaws (unsupported preconditions and threats) extant in ρ. Plan elements also refer to any of the parameters, preconditions or effects of a step, any action provided in the domain.

Definition 4-15. A **refinement** to an incomplete partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$ is the resolution, according to Definition 4-12 and Definition 4-13, of any single flaw in ρ. A plan without any flaws (a complete plan) cannot be refined. Partial order causal link (pocl) planning does not add elements to $S, \rightarrow, \rightarrow_{CL}, B$ except for the express purpose of resolving a flaw, although in general, the plan space may be explored without this restriction (with consequences to algorithmic systematicity).

There are four refinement types, **add-step**, **add-order**, **add-link**, **add-binding**, each of which refers to adding an element to the correspondingly named set. Partial-order planners use these refinements in order to address flaws by progressively adding elements to the null plan until a complete plan is found. In order to ensure the correctness of solutions returned, one is not permitted to add elements directly to the set of flaws (this can only be done indirectly, by adding to the other four sets defining the partial-order plan).

Definition 4-16. The **initial partial-order plan**, also called the **null plan**, given a classical planning problem $\varPsi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, consists of the initial step $\mathbf{s_0}$ and goal step $\mathbf{s_\infty}$, and an ordering constraint that forces s_0 to come before s_∞ in any solution.

Definition 4-17. A partial-order plan ρ is **complete** if and only if it contains no flaws, and the sets of ordering and binding constraints are consistent. A linearization of a complete partial-order plan is guaranteed to be a solution to the classical planning problem for which it was generated, due to the definition of plan refinements.

The algorithm for heuristic partial-order planning is shown below, followed by a detailed explanation.

Procedure HPOP(Ψ)
Input: a classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, s_0 is a finite set of ground atoms describing the initial world state of the problem, and g is a finite set of atoms that define the problems goals.
Output: a complete partial-order plan for (Ψ) or fail

1. frontier $\leftarrow$ { nullPlan(Ψ) }
2. **while** frontier is not empty **do**
3. $\pi \leftarrow$ heuristicSelectPlan(frontier, Δ)   //backtrack point (AND?)
4.   remove π from frontier
5.   **if** π has no flaws **then**
6.   **return** π
7.   **else**
8.   flaw $\leftarrow$ heuristicSelectFlaw(π, Δ)   //NOT backtrackable (OR?)
9.   add all resolutions of flaw to frontier
10. **return** fail

Figure 4.1 Pseudo-code of HPOP

Figure 4.1 presents the pseudo-code for POP planning with heuristics (HPOP), which is presented in a fashion similar to that of Williamson and Hanks 1996, (for an excellent discussion of the importance of domain-independent heuristics for both plan and flaw selection POCL planning, and an empirical evaluation of several techniques for doing so, see Younes & Simmons 2003), with the exception of notation changes that I made to be consistent with the definitions already presented. It receives as input a classical planning problem Ψ (Definition 3-14).

It outputs a complete, partial-order plan (Definition 4-17) for Ψ or fail if none is found. HPOP maintains a list of partial plans *frontier*, which at the beginning contains the initial plan (line 1, Definition 4-16). Line 2 begins an iteration that will continue while there are partial plans in *frontier* and the solution has not been found (line 6). Line 3 selects a plan π in *frontier* based on a heuristic selection, such as selecting the plan in *frontier* that has the fewest open condition flaws (Definition 4-10). If this plan has no flaws (Definition 4-9), it is returned and the process is terminated (line 5-6). Otherwise, all possible refinements (Definition 4-15) of flaw *flaw* (selected heuristically) in plan π are computed and added to *frontier* (line 8-9); an alternative to adding all refinements of flaw *flaw* in plan π to the search frontier of plans *frontier* is to make a backtrackable selection of a single flaw resolution strategy of *flaw* to perform, so long as each alternative flaw resolution strategy is guaranteed to be considered once and only once upon backtracking. Note that line 8, the selection of flaw, is not a backtrack point given that, by definition, all flaws in a partial-order plan must be resolved before a solution can be obtained; the selection of which flaw to resolve can have a profound effect on the efficiency of the search process, but the order of doing so neither effects the algorithm's systematicity nor completeness. Furthermore, the selection of which partial plan to refine (line 3) amounts to committing to one particular resolution added to *frontier* (line 9) during a previously resolved flaw (line 8). The now-refined plan π is removed from *frontier* in line 4. When *frontier* contains no plans, a failure is returned (line 9). For speed efficiency reasons, the set of flaws is usually incrementally maintained in a data structure for each partial-order plan, rather than doing the computationally expensive task of re-computing the flaws whenever they are being checked.

Many domain-independent strategies exist to guide refining partial-order plans; these strategies are used to guide the heuristic selection of the next plan to refine (line 3), and to guide the heuristic selection of the next flaw to resolve (line 8). For example, some simple strategies call for solving all open conditions first and then solving the threats (ie select open-condition flaws until only threats remain). Other, more sophisticated strategies balance between resolving the causal threats and the open conditions. Partial-order planning using these strategies has been empirically shown to perform slower than other more recent first-principles planners such as those that use planning graphs or heuristic planners. Recently, heuristics have been devised for POP that have resulted in comparable performance results with other state-of-the-art planners. These heuristics estimate the cost of reaching a complete plan from the current partial plan. Such estimates are used to compare competing refinements with A* cost functions (Hart *et al.*, 1968) or other heuristic cost functions (Younes & Simmons, 2003). Planning graphs have been used to estimate the cost of refining the current partial plan into a complete plan (Blum & Furst, 1997). The heuristic estimate is obtained by removing all negative effects from the actions and using planning graphs to obtain a solution for this relaxed problem (e.g., (Hoffmann & Nebel, 2001)). A similar process has been developed for POP planning with adjustments to account for the fact that POP does not maintain an explicit world state, whereas totally-ordered planners (like FF) do (Nguyen & Kambhampati, 2001). For this work I unsuccessfully attempted to use the heuristic POP planner VHPOP (Younes & Simmons, 2003), which is freely available for download and has been used to build a heuristic adaptation POP planner (van der Krogt & Weerdt, 2005) and a probabilistic conformant POP planner (Onder *et al.*, 2006). VHPOP is both well-

conceived and well-implemented – it earned the title "Best Newcomer" in the third International Planning Competition (held at AIPS-2002). The following is a quote from Hakan L. S. Younes (Younes, 2006), which sums up nicely the unique position his system holds in the modern history of partial-order planning:

> Entering a partial order planner into the planning competition was a gamble. I almost felt that the fate of this once dominating planning paradigm was resting on my sholders [sic]. Had VHPOP performed poorly, it might have thrust partial order planning back into darkness. Instead it earned me recognition as Best Newcomer and put partial order planners in the spotlight once again, an achievement that crowned my two-year effort with the planner.

I found it difficult to modify the codebase of VHPOP to suit my research needs for this dissertation, but was enlightened by the POP planning subtleties revealed in Youne's implementation. I instead wrote DCPOP and HIEPPR-POP in java, making some sacrifice in speed in exchange for portability and impact. A big thanks is owed also to the creators of JavaFF (Coles et al., 2008) for their PDDL parser and STRIPS data structures – these core classes only required minimal changes for my needs and, as advertised, I found their implementation an excellent basis for extension.

4.2   Domain Configurable Planning With Refinement Rules

This section presents the core of the algorithm I developed for this dissertation.

4.2.1   Partial-Order Plan Refinement Rules

The plan adaptation approach I present in this dissertation draws much inspiration from existing research in domain-configurable planning. In this form of planning, domain-specific knowledge enhancing the action schemas is given. This knowledge is used to guide the planning process which, like first-principles planning, generates a plan

from scratch. Examples of state-space domain-configurable knowledge structures include HTN methods in SHOP (Nau et al., 1999) and temporal logic rules in TLPLan and TALPlan (Bacchus & Kabanza, 2000; Kvarnström, & Doherty, 2001). Domain-configurable planners have been shown to solve problems more quickly and to scale much better with problem size than first-principles planners. Because of their scalability, their increasing number of applications, and their ability to drop classical planning assumptions, domain-configurable approaches are believed to be closing the gap between academic research in AI planning and real-world applications [Nau et al., 2005]. Consequently, the technique I developed is akin to domain-configurable planning in that it uses domain-specific knowledge to guide the planning process, but is unlike most domain-configurable planning in that it can reuse previous planning solutions in addition to planning from scratch. One of the crucial research challenges of this work is how to represent the domain-configurable plan adaptation knowledge. Before presenting my formalism, I first discuss two notable domain-configurable knowledge formalisms, HTN methods and temporal logic rules.

HTN methods are used to sketch the underpinning of the solutions by indicating how and when high-level tasks are decomposed into simpler tasks. Methods use applicability conditions that are evaluated against the current state of the world, which is maintained at all times by the forward hierarchical planning process followed by the SHOP planner. Compound tasks are further decomposed until primitive tasks, which are accomplished by actions, are generated. Applying the actions transforms the current state. The SHOP planner maintains a totally-ordered plan that indicates how to transform the initial state into the current state. Hence, SHOP methods can be seen as domain-configurable

81

knowledge on top of a forward state-space search process. For example, in the transportation logistics domain, a HTN method might check if the location of a package in the current world state (not plan state) does not match the desired goal location and if so call some suitable subtasks to move this package.

Temporal logic rules are used in TLPlanner and TALPlanner in a forward state-space search process. These rules use temporal logic modal operators that express relationships between a state and subsequent states, and are used to define a prune function that detects and cuts unpromising nodes (i.e., states) in the state space. For example, a rule might encode the following condition: "Do not move a block if its current position is consistent with the goal configuration". If the planner reaches a state S where one such block has been moved from a previous state in which its location was consistent with the goal configuration, then the rule will trigger and prune S from the search space.

Applicability conditions in these systems refer to the state of the world and the consequence of domain knowledge can be either to prune out world states (e.g. TLPlan) or to constrain the subsequent applicable HTN methods (e.g. SHOP). For the purposes of the refinement rules presented in this dissertation, because HIEPPR-POP uses a partial-order plan representation, the algorithm needed knowledge whose applicability conditions refer to the state of the plan rather than the state of the world, and therefore whose consequence prune out incomplete partial-order plans. This is an important distinction because given a partial-order plan, computing the valid conditions for any point in the plan is an intractable problem (Chapman, 1987). Furthermore, I would like to be able to explicitly tell the partial-order planner which refinements to take to modify the

current plan. Therefore, the algorithm presented herein uses domain-configurable partial-order knowledge encoded and interpreted according to the following definitions.

Definition 4-18. A **DCPOP refinement rule precondition** *rrp* = (+|−)<POP plan element>, given a classical planning domain Δ = (*C*, *P*, *O*), *rrp* is syntactically a plus or minus, followed by any POP plan element (Definition 4-14) that might exist in any partial-order plan generated by a classical partial-order planning process operating on Δ. The semantics for plan elements remains unchanged.

The semantics of a refinement rule precondition are defined relative to a partial-order plan ρ = (*S*, →, →*CL*, *B*), and the set of flaws *F* extant in ρ as follows: the precondition *rrp* is true when preceded by a plus, relative to ρ, if and only if the POP plan element in *rrp* is an element of ρ or *F* and false otherwise; the precondition *rrp* is true when preceded by a minus, relative to ρ, if and only if the POP plan element in *rrp* is not an element of ρ nor *F* and false otherwise.

Because of the verbosity of the parsable form of plan elements, I will continue to use the shorthand representations used up to this point for the remainder of the dissertation. However, in any implementation of an algorithm using rules, there must be a representation for each plan element. The following is the syntax I found useful:

- an open condition: (openCondition <atom> <step>) and (openCondition <variable> s[g]),

- a threat: (threat <link> <step>)

- a precondition that may or may not be supported by the effect of another step: (precondition <atom> <step>) and (precondition <variable> <step>)

- an effect: (effect <step> <atom>) and (effect <step> <variable>)

- an ordering constraint: (order <step> <step>)

- a step: (step <step> <operator head>) or (step <step> <variable>)

- a link: (link <step> <atom> <step>) or (link <step> <variable> <step>) or (
  link <effect> <open condition>)

Definition 4-19. A **DCPOP refinement rule effect** rre = (do:|undo:) <POP plan refinement>, given a classical planning domain $\Delta = (C, P, O)$, rre is syntactically a 'do:' or 'undo:', followed by any POP plan refinement (Definition 4-15) that might be applied to a partial-order plan generated by a classical partial-order planning process operating on Δ. The syntax and semantics for plan refinements remains unchanged.

The semantics of a refinement rule effect are defined relative to a partial-order plan ρ = $(S, \rightarrow, \rightarrow_{CL}, B)$, and the set of flaws F extant in plan ρ as follows: the effect rre is **applied** to ρ by performing the plan refinement in rre when preceded by a 'do:', or removing the plan refinement from ρ when preceded by an 'undo:'. When step s is deleted (i.e., by an undo) from ρ, any ordering constraint or causal link connecting to/from s is also removed. When an ordering constraint, a causal link, or a binding constraint is deleted from ρ, the sets of plan elements may need to be updated. The choice of how (for example, computing the transitive closure of the ordering constraints, or removing all instances of a deleted variable from the binding constraints) and when (for example, after each effect, or after applying all effects belonging to a single applicable rule) to update the elements of ρ and F is left as implementation detail, but can have a profound effect on the time taken to apply refinement rule effects.

Definition 4-20. A **DCPOP refinement rule** *drr* = if *rrp* [, *rrp*]* then *rre* [, *rre*]*, is syntactically an 'if', followed by one or more refinement rule preconditions, followed by a 'then', followed by one or more refinement rule effects.

The semantics of a refinement rule are defined relative to a classical planning domain $\Delta = (C, P, O)$, a partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$ derivable in a classical partial-order planning process operating on that domain, and the set of flaws F extant in ρ as follows: the rule is **applicable** to ρ if and only if every *rrp* appearing in the 'if' part is true in ρ, and **inapplicable** otherwise. There may be multiple ways to make the preconditions of a rule *drr* true, and to ensure systematic search this is handled as a backtrack point. An applicable rule *drr* can be **applied** to a partial-order plan ρ by applying all of the effects of *drr* to ρ in the order that they appear in the rule. Like the preconditions, there may be multiple ways to apply the effects of a rule, and this is also a backtrack point in the search space.

Definition 4-21. A **DCPOP planning domain** $DPD = (\Delta, DRR)$ is a tuple containing a classical planning domain $\Delta = (C, P, O)$, along with a list *DRR* of one or more DCPOP refinement rules *drr* written for Δ.

Definition 4-22. A **DCPOP planning problem** $DPP = (\Psi, DPD)$ is tuple containing a classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, along with a DCPOP planning domain $DPD = (\Delta, DRR)$.

The conditional, "if", part of a rule is a conjunction of one or more POP plan elements (Definition 4-14: steps, links, bindings, ordering constraints, and flaws). These POP plan elements are preceded by either a plus or a minus. The consequent part, "then", is a

sequence of POP plan refinements (Definition 4-15: add step, add link, add order, add binding), preceded by do or undo symbols. The semantics of a rule are as follows. The rule is satisfied if each of the POP plan elements preceded by a plus sign occurs in the partial-plan being refined, and none of the POP plan elements proceeded by the minus sign occur in the partial plan. The consequent part indicates each of the POP plan refinements to add, if it is preceded by a "do", or to retract, if it is preceded by an "undo". The POP domain-configurable (refinement) rules are a natural extension of POP refinements and in fact all POP refinements can be expressed using these rules. For illustration purposes the rules in **Table 4.1** encode the POP plan refinement strategies that solve a flaw of type threat:

if $+(s_t \parallel (s_p \rightarrow_p s_c))$, $\neg(s_t < s_c)$ **then do:** $s_c \rightarrow s_t$	PROMOTE
if $+(s_t \parallel (s_p \rightarrow_p s_c))$, $\neg(s_p < s_t)$ **then do:** $s_t \rightarrow s_p$	DEMOTE
if $+(s_t \rightarrow_r \parallel (s_p \rightarrow_p s_c))$, $\neg(p = r)$ **then do:** SEPARATION	SEPARATE

Table 4.1 DCPOP rules encoding the three ways to resolve flaws of type threat.

The condition $(s_t \parallel (s_p \rightarrow_p s_c))$ is not directly a POP plan element but can be derived from plan elements by using Horn clauses. I used horn clauses to simplify conditions in the conditional part of the rule. This has been shown to be quite useful to simplify domain descriptions in domain-configurable planners such as SHOP as well as for first-principles planners. Among the conditions that can be defined by Horn clauses are *same(?x,?y)* and *different(?x,?y)* indicating that two variables take the same (or different) value. In this dissertation I instead write *?x = ?y* (or *?x ≠ ?y*) for readability. I also use Horn clauses to compute the relation *s < s'* between any two steps to indicate that for any linearization of the plan, step *s* will occur before step *s'*.

It is similarly possible to encode the POP plan refinement strategies that solve a flaw of type open precondition as refinement rules:

if $+ p@s_c, + s_p \rightarrow_p, \neg(s_c < s_p), s_p \neq s_c$ **then do:** $s_p \rightarrow_p s_c, s_p < s_c$	REUSE//binding made explicit?
if $+ p@s_c, + O_p \rightarrow_p$ **then do:** $s_p = O_p, s_p \rightarrow_p s_c, s_p < s_c$	INSTANTIATE

Table 4.2 DCPOP refinement rules encoding the two ways to fix open conditions.

For the work in this dissertation, the goal is not to use POP rules to express *domain-independent* refinements such as the ones shown in **Table 4.1** and **Table 4.2**, but rather *domain-specific* refinements and retractions modifying an existing plan. That is, rather than leaving the selection of flaw (line 8 in **Figure 4.1**) and selection of which flaw resolver's children to explore (line 3 in **Figure 4.1**) to a non-deterministic or domain-independent heuristic choice, I seek to capture domain-specific rules that govern both which flaw (or flaws) to operate upon, and how specifically to address them. These domain-specific rules augment the input to the domain-independent search process, yielding the *domain-configurable* planner I sought to create.

There are two classes of DCPOP refinement rules: progressive rules and regressive rules. Distinguishing between these two classes of rules aids the systematic search of the algorithm.

Definition 4-23. A DCPOP refinement rule *drr* = if *rrp* [, *rrp*]* then *rre* [, *rre*]* (Definition 4-20), where *rre* is a refinement rule effect (Definition 4-19), *drr* is called a **regressive rule** when every effect in *drr* is prefixed with an *undo*; thus rules of this type only modify a given plan by retracting POP plan refinements (Definition 4-15).

Definition 4-24. A DCPOP refinement rule *drr* = if *rrp* [, *rrp*]* then *rre* [, *rre*]* (Definition 4-20), where *rre* is a refinement rule effect (Definition 4-19), *drr* is called a

progressive rule when every effect in *drr* is prefixed with a *do*; thus rules of this type only modify a given plan by adding POP plan refinements (Definition 4-15).

(1)	(2)
if $+$ (at ?pac ?loc) @ s_g $\quad + s_0 \rightarrow$ (package ?pac) $\quad + s_p \rightarrow$ (at ?pac ?loc) $\quad \neg\, s_p \rightarrow$ (at ?pac ?loc) s_x $\quad \neg\, s_g < s_p$ **then do:** $\quad s_p \rightarrow$ (at ?pac ?loc) s_g	**if** $+$ (at ?pac ?dest) @ s_g $\quad + s_0 \rightarrow$ (package ?pac) $\quad + s_0 \rightarrow$ (in-city ?dest ?city) $\quad + s_p \rightarrow$ (at ?pac ?start) $\quad + s_0 \rightarrow$ (in-city ?start ?city) $\quad \neg\, s_p \rightarrow$ (at ?pac ?start) s_x $\quad \neg\, s_g < s_p$ $\quad + s_t \rightarrow$ (at ?truck ?start) $\quad + s_0 \rightarrow$ (truck ?truck) $\quad \neg\, s_t \rightarrow$ (at ?pac ?loc) s_x **then do:** $\quad s_{load}$: (load ?pac ?truck ?start) $\quad s_{drive}$: (drive ?truck ?start ?dest) $\quad s_{unload}$: (unload ?pac ?truck ?dest) $\quad s_{load} \rightarrow$ (in ?pac ?truck) s_{unload} $\quad s_{drive} \rightarrow$ (at ?truck ?dest) s_{unload} $\quad s_{unload} \rightarrow$ (at ?pac ?dest) s_g $\quad s_p \rightarrow$ (at ?pac ?start) s_{load} $\quad s_t \rightarrow$ (at ?truck ?start) s_{load} $\quad s_t \rightarrow$ (at ?truck ?start) s_{drive}

Table 4.3 Two progressive refinement rules for the transportation logistics domain

Table 4.3 shows an example of two plausible POP progressive refinement rules in the transportation logistics domain. These POP rules partially encode the strategy that has been used in the examples up to this point, which is to try to reuse steps that provide needed effects rather than add new steps to provide those effects. The first refinement rule fixes an open condition flaw by reusing a step that places a package where it needs to be. The first two conditions of the rule check if a package ?pac is required to be at location ?loc for some step s_g (possibly the goal step) in the plan. The third condition finds a step s_p (possible the initial step) that establishes the package at that required location; note that once a variable is bound to a value, the matching process no longer treats it as a variable that needs filling. The fourth condition verifies that there is no other

step using the required effect of s_p, in order to avoid creating threats that need to be resolved later in the planning process. The final condition ensures that s_g is not already constrained to come before step s_p, which would make using s_p for s_g impossible. This rule makes a single refinement: it adds a causal link (and ordering constraint) between s_p and s_g to resolve the open precondition on s_g.

The second POP rule is much lengthier, but reveals a few subtleties of the representation formalism. In plain words, the rule expresses the following: a package is not at its destination, but is at a spot where an unused truck is; load the package in the unused truck, drive to the destination and unload it. The first two conditions find a step with an open precondition indicating a package has not been delivered. The next three conditions verify that the package's starting location and final destination are in the same city (if this is not true, then the package would have to be flown first). The next condition, a "not", ensures that there is no other step in the plan that uses the effect establishing the package's start location. Without this guard, it would be possible that refinements added are either redundant, or worse, introduce a threat that is difficult or impossible to resolve. The next condition, also a "not", verifies that the step to provide the needed effect is not already committed (ordered) to occur after the step that needs to consume the effect. The final three conditions find a truck that is "available", meaning that its location is not being used for any other step. Note also that the final condition reuses the step named s_x; it is always the case that a non-grounded variable appearing in a "not" condition can be reused later in the conditional list, because it is guaranteed that no successful binding could be made for it.

Rule two makes nine refinements to a plan that matches the conditionals. The first three refinements are the addition of a load step, drive step, and unload step. The remaining 6 refinements all add causal links, and the associated ordering constraints to back them. The first causal link refinement is interesting in that it links an effect that does not appear in the conditional section of the rule. Because unload actions require that the package be in the truck being unloaded, and also having just added a load/unload pair, the newly added effect is immediately linked with the newly added open condition. Similarly, the next causal link refinement in the rule eagerly supports the `(at ?truck ?dest)` precondition of the new unload step. The next refinement, $s_{unload} \rightarrow_{(at\ ?pac\ ?dest)} s_g$ supports the open condition that triggered this rule in the first place. The final three refinements reuse the effects found in the conditional section of the rule to support the flaws introduced by the addition of the three new actions.

Clearly, the two refinement rules presented above are insufficient for making a complete plan (having no flaws). This is a highly desirable property as in some domains it might be difficult to obtain a collection of refinement rules that produce a complete plan. Consequently, rules can be given for the more computationally complicated details (e.g., how to achieve the goals), leaving the rest to HPOP, the underlying first-principles planner. Ideally, the intermediate plan produced from adaptation will be easier to complete than the initial plan. However, if so desired, a complete set of rules can be fully encoded to ensure that the resulting plans are complete.

4.2.2 The DCPOP Algorithm

Procedure DCPOP(DPP, π_{old})
Input: A DCPOP Planning problem $DPP = (\Psi, DPD)$, with classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain; a DCPOP planning domain $DPD = (\Delta, DRR)$, where DRR is a list of one or more DCPOP refinement rules drr written for Δ; a partial-order plan π_{old}
Output: a complete plan for Ψ or fail

1. $\pi_{adj} \leftarrow$ adjust-plan(Ψ, π_{old})
2. `frontier` $\leftarrow$ { doAllRegressiveRules(DPP, π_{adj}) }
3. **while** `frontier` is not empty **do**
4. $\pi \leftarrow$ heuristicSelectPlan(Ψ, P)   //backtrack point
5.   remove π from `frontier`
6.   **if** π has no flaws **then**
7.   **return** π
8.   **else**
9.   R $\leftarrow$ { (drr, σ) : drr is an instance of a refinement rule in DRR,
                        σ is a substitution causing the preconditions of drr to be true
                        relative to π, and σ is as general as possible) } //backtrackable
10.   **if** R is empty **then**   //do classical POP planning
11.   `flaw` $\leftarrow$ heuristicSelectFlaw(π, Δ)   //NOT backtrackable
12.   add all resolutions of `flaw` to `frontier`
13.   **else**   //use DCPOP rule
14. $\delta \leftarrow$ nondeterministically choose pair (drr, σ) from R
15. $\pi' \leftarrow$ apply all refinement rule effects of δ to π
16.   add π' to `frontier`
17. **return** fail

Figure 4.2 Pseudo-code of DCPOP

Figure 4.2 presents the pseudocode of my domain-configurable plan adaptation algorithm on top of HPOP. As with HPOP (**Figure 4.1**) it receives as input a classical planning problem Ψ. Unlike HPOP, it also receives the plan to be adapted, π_{old}, and the DCPOP planning domain DPD. The output is a complete plan solving Ψ or fail if none is found. DCPOP begins by adjusting π_{old} relative to (Ψ) (line 1). Adjust plan works by repeatedly (1) removing a step s that mentions objects in the input plan that are not mapped into objects in the new problem, and (2) removing any ordering or causal link constraint connecting to/from s. This is a common step for adaptation in first-principles

POP planning (e.g., (Hanks & Weld, 1995; van der Krogt & Weerdt, 2005; Kuchibalta & Munoz-Avila, 2006)). Then, a set of plans is found by repeatedly applying the regression rules in *DPD* until none is applicable (line 2). These plans are added to `frontier` (line 2), the list of current candidate partial plans to be refined. The next part of the pseudocode continues iterating while there is at least one candidate plan to be refined and no solution has been found (lines 3-16). When the list of candidate plans is empty, a failure is returned (line 17). At each iteration, a candidate plan π is selected using the HPOP heuristics and is removed from `frontier` (lines 4 and 5). If this candidate plan has no flaws, it is returned (lines 6 and 7). Otherwise a new partial-plan π' is computed by applying an applicable refinement rule to π, and π' is added to `frontier` (lines 9, and 14-16). If no refinement rules are applicable to π (line 10), then standard classical POP refinements are added to `frontier` (lines 11 and 12).

To simplify the presentation and analysis of DCPOP, the choice of which refinement rule to apply (line 14) is specified as a nondeterministic step. Any implementation of the algorithm must necessarily be deterministic, and the correct implementation of the nondeterministic step is as follows:

- If in the course of iteration a plan π is refined with a chosen rule δ from its set R (line 9) to make a new partial-plan π', and in a subsequent iteration, π' is found to have neither an applicable refinement rule nor applicable classical refinement, then another iteration is performed on π using an untried element of its set R.

- If in the course of iteration a plan π has unsuccessfully had all elements of its set R applied, and furthermore all classical refinements to its heuristically

selected flaw have also been explored, then iteration of the algorithm resumes at (backtracks to) a previous choice point (selection of an untried δ from set R).

There are a few notable elements of the DCPOP algorithm, as compared to the classical HPOP algorithm presented in **Figure 4.1**. First, HPOP always begins search with the initial (or null) plan, whereas DCPOP is designed to begin search with a previously formed partial-order plan, which places DCPOP in the class of plan adaptation algorithms. Recall that the motivation for doing so is to reduce the size of the search space to be explored by exploiting the theory that similar problems have similar solutions (and therefore it is easier to refine a near-complete solution than generating a new solution from scratch). Doing so in this context amounts to adjusting the input plan (line 1) such that it contains no elements that would cause forward refinement (least-commitment) search to fail. Furthermore, DCPOP goes beyond domain-independent approaches to adjusting the input plan by incorporating domain-specific control strategies for adjusting the input plan (line 2). For example, a domain expert in transportation logistics may know that it is best to remove all commitments having to do with planes. In DCPOP, it is possible to encode rules that would retract all plan elements even tangentially related to plane scheduling, whereas in domain-independent plan adaptation frameworks, doing so would be impossible.

The second notable feature of the DCPOP algorithm is that, once an adjusted plan is created, the search algorithm is very nearly identical to HPOP. In fact, because DCPOP searches in the plan space (in contrast to, for example, task reduction in state-based HTN planning), even when there exist no applicable rules (line 10), it is still possible to further refine the (hopefully) much progressed, candidate partial-plans by applying traditional

domain-independent heuristic partial-order plan refinement techniques. This is significant for two reasons.

First, it allows DCPOP to find solutions to problems even when the set of input rules is insufficient to do so without applying any other search techniques. That is, the set of refinement rules **need not be complete** in order to find a solution. Rather than having domain engineers laboriously consider ways of fixing all problems that might exist in an incomplete partial-plan and authoring a correspondingly complete set of rules, a sparse set of rules can be authored in order to 'sketch' how to solve the hard (search space expensive) parts of finding a solution in a particular domain, while deferring to brute-force domain-independent search strategies to solve the other flaws. Even when search must resort to domain-independent approaches, domain-specific search can resume as soon as more rules become applicable. This eases the notorious knowledge acquisition bottleneck. For example, running DCPOP without any rules is equivalent to running HPOP, whereas running DCPOP with a single rule allows for making 'leaps' in the search space whenever that rule becomes applicable. Second, and once again because DCPOP operates in the plan space, I hypothesize that the incomplete plans may **more easily be extended into solutions** than incomplete plans generated by a totally-ordered, domain-configurable adaptation strategy. This equates to fewer 'dead-ends' in the search space because there are fewer commitments to break than in totally-ordered approaches (which are far more constrained).

Readers that are familiar with 'plan critics' may note the similarity of DCPOP refinement rules to the knowledge constructs used in such seminal planners as SNLP and O-Plan. Indeed, the notion of critics was once a popular way of expressing search control

strategies for how to resolve particular types of errors in incomplete plans. These critics paved the way for understanding what came to be known as the two fundamental flaws in partial-order planning (unsupported conditions, and threats), and the minimal and sufficient ways to resolve them. Because of the attractiveness of the properties of sound, complete, and systematic search algorithms (notably, getting your planning algorithm published), these flexible critics were in effect discarded in exchange for the domain-independent search strategy that is captured by HPOP (on account of the minimal, necessary, and sufficient nature of how to resolve those fundamental flaws). The fundamental argument of this dissertation is that, for the most part, planning approaches have unnecessarily and deleteriously discarded the flexibility and efficiency of applying 'unsystematic' critics to quickly traverse portions of a planning search space (and thereby making hard problems solvable), in exchange for critics that provably force an expensive (yet complete) exploration of the search space that is only useful for the smallest of planning problems. I argue that combining necessary and sufficient refinements (classical HPOP) with 'rule-of-thumb' knowledge (DCPOP refinement rules) yields an approach that exploits the strengths of both while ameliorating their respective weaknesses: HPOP is strong in its systematic search, and weak in solving large problems; unstructured critic evaluation and application is strong in its ability to solve large problems (with complete hand crafted rules), but weak in systematically exploring ways of fixing all types of problems (when the rules are incomplete).

4.3 Properties of the DCPOP Algorithm

In this section, I present criteria that determines the soundness (i.e., that all solutions generated are correct), correctness (the ability of the algorithm to correctly execute refinement rules according to their semantics), completeness (whether or not at least one solution will be found whenever a solvable problem is given), and complexity (an abstract formulation of worst-case running time for the DCPOP algorithm to solve a problem, relative to input expert knowledge).

4.3.1 Soundness

An algorithm that is sound is one that always generates correct solutions (relative the definition of the problem the algorithm solves) – that is, the output of a sound algorithm must always solve the input problem, or report the failure to do so.

Definition 4-25. Given a classical planning problem Ψ (Definition 3-14), and a collection of refinement rules *DRR* (Definition 4-20), a planning algorithm is **sound** if and only if, all answers returned for Ψ by the algorithm using *DRR* are guaranteed to be solutions to Ψ, according to Definition 3-16.

The previous definition not only states that soundness means never returning a wrong answer (and reporting a failure to find a solution when one exists is ok), it also basically states that solutions generated with the refinement rules and the operators could also be generated without the refinement rules (i.e., by using the operators only).

Theorem 4-1. The DCPOP algorithm is sound, according to Definition 4-25.

Proof. The DCPOP algorithm (**Figure 4.2**) returns a candidate solution plan in line 7. It returns a candidate only if the candidate has no flaws. Because a partial-order plan having no flaws (and consistent sets of ordering and binding constraints) is a solution to the classical planning problem (Definition 3-16), then the algorithm is sound.

∎

4.3.2 Completeness

A planning algorithm is "complete" if, whenever given a solvable problem, that algorithm will find at least one solution to that problem; if a complete algorithm returns a failure to find a solution for a given problem, then this problem has no solution. This notion can be tricky for least-commitment plan-space planning, given that the search space of the algorithm is infinite. Infinite partial plans can be generated because of the fact that the planning algorithm can always resolve a flaw of type open precondition by supporting it with a new step that has a matching effect. Therefore, when analyzing an algorithm that searches in an infinite space of solutions, the algorithm is considered to be complete if and only if the solution is guaranteed to be generated (as a leaf node) in the search process when computational space and processing time are also treated as infinite resources. As put by Weld 1994, "if a plan exists, does a sequence of non-deterministic choices [through the algorithm] exist that will find it?"

Definition 4-26. Given a classical planning problem Ψ (Definition 3-14), and a collection of refinement rules *DRR* (Definition 4-20), a planning algorithm using *DRR* to generate plans is **complete** if and only if, whenever Ψ is solvable, the algorithm generates a solution.

97

Theorem 4-2. The DCPOP algorithm is ***not*** complete, according to Definition 4-26.

Proof. In order to prove the completeness of the DCPOP algorithm, it must be shown that, for an input solvable classical planning problem Ψ, at least one of DCPOP's execution traces returns a solution. Additionally, there are several dimensions along which to consider the completeness of DCPOP, and all possible combinations of values for each dimension must be provably complete in order for DCPOP to be provably complete in general. The dimensions, and the values they can take, are as follows:

- Whether or not the plan to be adapted, π_{old} is the null plan for Ψ

- Whether or not the retraction rules in the set of DCPOP rules *DRR* can generate the null plan for Ψ

- Whether the refinement rules in *DRR* are always applicable, sometimes applicable, or never applicable

Case 0: First, we assume that the input plan to be adapted, π_{old}, is equivalent to the null plan for Ψ. That is, no plan adaptation is to occur.

In this situation, no retraction rules will be applicable because retraction rules can only remove plan refinements, and the null plan contains no refinements that can be removed (by definition); thus, whether or not the retraction rules present in *DRR* are able to make the null plan for Ψ has no effect on the completeness of this case.

That leaves a final consideration for Case 0, which is the effect on the search of the refinement rules in *DRR*.

- In the event that there are no refinement rules in *DRR*, or in the event that none of the refinement rules are ever applicable, line 10 of DCPOP always evaluates to true. Given that in Case 0 search begins with the null plan and neither

retraction nor refinement rules are ever applicable, search under these conditions is equivalent to HPOP (that is, the while loop is equivalent to that of **Figure 4.1**). Because HPOP is complete, DCPOP is also complete under these restrictions.

- In the event that there are refinement rules in *DRR* that are applicable, recall that after all applicable refinement rules (under all valid substitutions) are applied to a candidate partial-order plan in the frontier, the algorithm backtracks to classic HPOP flaw resolution strategies for that partial-order plan. Thus, either the application of refinement rules alone will generate the solution node in the search space, or the application of classical HPOP flaw refinement alone will generate the solution, or an interleaving of rule application and HPOP flaw refinements will do so. Therefore, DCPOP is complete under these considerations.

Because we have considered all variations of inputs under the restriction that the input plan to be adapted is equivalent to the null plan, we can now assert the following: DCPOP is complete if the input plan to be adapted is equivalent to the null plan (Definition 4-16) for the given classical planning problem.

Case 1: Having proven all completeness results under the assumption that the plan to be adapted, π_{old}, is equivalent to the null plan for the input classical problem Ψ, the next case to consider is when π_{old} is *not* equivalent to the null plan for Ψ. That is, DCPOP is asked to perform plan adaptation. For Case 1, we also make the assumption that the retraction rules, when applied (line 2), result in the null plan for Ψ.

In this situation, application of the retraction rules will result in the null plan for Ψ. This has the effect of discarding plan adaptation for the first-principles HPOP approach. Because in Case 0, it was proven that DCPOP can always generate a solution when starting from the null plan for Ψ, this case is also complete.

Case 2: The final case to consider is when π_{old} is *not* equivalent to the null plan for Ψ (plan adaptation occurs), and, unlike Case 1, the retraction rules when applied (line 2) do *not* result in the null plan for Ψ (non-trivial plan adaptation).

In this case, DCPOP might not generate solutions for problems where a first-principles POP plan adaptation algorithm will generate one. Because DCPOP will only perform retraction as indicated by the POP rules (line 2), and for this case it is assumed that the partial-plan resulting from applying the retraction rules is not the null plan, the frontier at the first iteration will contain a single partial-plan π' containing commitments on some or all of the elements of its steps, ordering constraints, causal links, or binding constraints. It is trivial to construct a situation where any one of these constraints results in a flaw that has no resolution (for example, a threat that cannot be resolved consistently via promotion, demotion, or separation). Therefore, DCPOP is *not* guaranteed to be complete under the restrictions imposed in this final case.

Because Case 3 shows a situation where DCPOP is not provably complete, DCPOP is not complete in the general case.

∎

A systematic plan adaptation planner (e.g., Hanks & Weld (1995)'s or van der Krogt & Weerdt (2005)'s algorithms) could theoretically generate one as they will

systematically retract steps from the adjusted input plan, eventually retracting all constraints until the null plan is reached (if necessary).

The following lemmas summarize the completeness results from the proof of Theorem 4-2.

Lemma 4-1. DCPOP is complete if the input plan to be adapted is equivalent to the null plan (Definition 4-16) for the given classical planning problem.

Proof. See Case 0 in the proof of Theorem 4-2. ∎

Lemma 4-2. When given a solvable problem and a set of retraction rules in *DRR* that, when applied, result in the null plan for the input classical problem Ψ, DCPOP is complete.

Proof. See Case 1 in the proof of Theorem 4-2. ∎

Lemma 4-3. When given a solvable problem, DCPOP is not guaranteed to be complete when the input plan to be adapted is not the null plan, and the retraction rules do not cause the search frontier to be initialized with the null plan, regardless of whatever refinement rules *DRR* may contain.

Proof. See Case 3 in the proof of Theorem 4-2. ∎

4.4 Discussion

Other researchers have proposed plan adaptation algorithms based on heuristic planning (Koenig *et al*., 2002; Boella, & Damiano, 2002; van der Krogt & Weerdt, 2005). However, unlike DCPOP, these algorithms modify the input plan by first removing steps,

followed by a call to a heuristic planning process to generate a complete plan from the modified plan. This form of adaptation has shown improved performance over that of state-of-the-art first-principles planners but is still far from the performance of domain-configurable first-principle planners. It is relevant to this discussion to point out that, as of today, the best performing POP planners use heuristics, and that *no domain-configurable POP planner exists*. This is in itself significant because reasoning with partial-order plans is considered crucial in many real-world situations (Ghallab *et al*., 2004).

At this point a clarification is needed. There is a variant of SHOP, called SHOP2, which allows defining a partial order between the tasks to achieve (Nau *et al*., 2001). The way SHOP2 operates is to select the next task to achieve that is consistent with the partial order. The selected task is decomposed all the way to primitive tasks, which are satisfied by actions. These actions are used to *advance forward the current world state*. The process repeats itself by selecting the next task. At all times, the partial solution constructed is a totally-ordered plan consisting of all primitive tasks in the order they were generated, as is the case with SHOP. Although this allows for interleaving actions achieving different tasks, unlike POP, actions in SHOP2 appear in the plan in the order they were generated. This is analogous to the way non-linear total-order planners such as Prodigy operate; resulting plans can interleave actions achieving different goals but the planner commits to the order of the action at the point that the actions are generated (Veloso *et al*., 1995). The crucial characteristic of POP is not the fact that actions can be interleaved to achieve different goals but that POP does not commit to an action ordering unless it is necessary (Kambhampati *et al*., 1996).

Another clarification should be made about HTN planners such as UMCP (Erol *et al*., 1994b) which pre-dated SHOP. These planners decompose a task into a plot, which indicates the resulting subtasks and their constraints including ordering constraints and causal links. Like POP, UMCP can generate partially ordered plans. But unlike POP, UMCP does so in a hierarchical fashion by reasoning on high-level tasks. UMCP and other HTN planners such as O-PLAN (Currie & Tate, 1991) and SIPE (Wilkins, 1988) were also in part motivated by what later came to be known as domain-configurable planning; knowledge, in the form of HTNs, was provided to guide the planner. However, the performance of general HTN planning can be very slow because of the multiple interactions between the tasks at different levels in the hierarchy. Also, this knowledge is crucially different from that of DCPOP in that rules in DCPOP can make any kind of refinement, whereas the control knowledge in the other systems is focused on step addition and ordering only (see Section 5.4 for more details about how my algorithms relate to UMCP). Currently, research on the general form of HTN planning is almost non-existent. It is conceivable that the techniques that I developed for this dissertation could be used to improve the performance of general HTN planning, and would be an interesting avenue to explore as future work.

There are a number of research challenges that still need to be addressed. First, the challenge of selecting among alternative POP rules must be discussed. This refers to line 9 of DCPOP. The pseudocode applies all applicable rules and collects all resulting plans. The other two alternatives, selecting the first applicable rule or using other more sophisticated criterion to select one rule, will need to be investigated to formulate possible trade-offs between these alternatives.

Challenges involving the use of POP rules. First, I studied ways to detect applicable POP rules and to apply them to modify a partial-order plan. This required developing methods to quickly evaluate the conditional part of POP rules. In my work with HTN plan adaptation Repair-SHOP [Warfield et al., 2007], we developed a structure to quickly identify the first inconsistent action of the plan, which was much faster than checking every action beginning from the first until the inconsistency is found. DCPOP has several analogous data structures that help evaluate POP rules in the current plan. Second, multiple POP rules might be applicable to the same plan. There are three possible approaches when this happens. One is to apply POP rules simultaneously to obtain alternative plans. Many planners such as VHPOP follow a similar strategy in that they compute all possible one-step POP refinements that can be made to the current plan (Younes & Simmons, 2003). This works surprisingly well provided that adequate heuristics are defined to select the next plan to refine among several candidates. The second alternative is to leave to the domain expert to encode the rules so that such conflicts are minimized and list the order in which these rules are to be evaluated; that is, during problem solving the first applicable rule is selected, and applied. This is the approach followed by the SHOP HTN planner. A third alternative approach is to use rule de-conflicting techniques to select appropriate rules (e.g., Chapter 9 of Russell & Norvig (2002)).]

5   Hierarchical partial-order plan refinement

The refinement rules presented in Chapter 4 form the basis for controlling, in a manner consistent with the instructions encoded by a domain expert, the refinements that a partial-order planner should make under certain conditions. I've shown that this domain-configurable control knowledge is unique in how it controls the underlying planning algorithm DCPOP, and furthermore, that the expert domain-configurable control knowledge need not be complete for the algorithm to find a solution. Additionally, results of experiments described in the next chapter reveal that refinement rules can be used to adapt plans without a tradeoff that has long been considered "inescapable" by case-based reasoning researchers.

But in what situations does this encoding of rules, and their use in the DCPOP algorithm, fall short of ideal, and how can these limitations be addressed? That is, how effectively can rules model domains (how little or how much knowledge is needed to have a useful impact on plan generation, in terms of the ratio of the number planning decisions guided by rules to the number of planning decisions made by first-principles, domain-independent planning operations), how efficient is the planning algorithm while using rules (is the process faster or slower than first-principles, and how does it compare to other domain-configurable planning techniques), and what is the quality of the plans produced?

There are a few notable limitations in how this rule knowledge is encoded and used in DCPOP; this chapter addresses how to extend the representation and algorithm to eliminate these limitations. To do so, I rely upon the notational and computational power

of hierarchies, and introduce a new algorithm called HIEPPR-POP – for HIErarchical Partial Plan Refinements for Partial-Order Plans (pronounced "hyper pop"), which subsumes the DCPOP approach (all domains that can be encoded for DCPOP can be encoded in HIEPPR-POP, but not the other way around).

The two main limitations addressed in this chapter arise from refinement rules that share a prefix of refinement rule preconditions, and rules that share common refinement strategies. As presented, the refinement rules of the previous chapter require the following inefficient, and inconvenient limitations:

(1) If two or more rules differ only in a single precondition, each of the common preconditions must be restated and re-evaluated for each rule. This not only makes the knowledge encoding process more error prone and tedious, but it also wastes computation.

(2) If the refinements required by a rule are those that are defined by another rule, those refinements cannot be reused (ignoring "copy/paste") – that is, there is no way for one rule to "refer" to another. Without such a mechanism, the process of encoding rules is more difficult and error prone than it need be.

Example 5-1. Illustrating first limitation, shared preconditions.

The following is an abstract example of the first limitation, namely the situation where two or more rules differ in a single precondition.

Suppose there are two refinement rules, R_a and R_b, each of which has 11 preconditions. The first 10 preconditions of R_a and R_b are exactly the same; only the 11th precondition differs. Suppose also that neither rule is applicable, because the 11th precondition is not satisfied in the current plan by either rule. The DCPOP algorithm

would have to, for each of the rules, evaluate the first 10 preconditions in order to find

that the 11$^{\text{th}}$ was not satisfied. This situation is only worsened if there are additional rules

sharing the common precondition prefix, or the precondition list is long. In practice, I

found that refinement rules frequently share common preconditions, often being

differentiated by only a few discriminating preconditions. What is needed is a way to

"group" these common preconditions, both for authorial convenience, and algorithm

efficiency.

Example 5-2. Second limitation is the inability to refer to other rules.

The second limitation, having to do with the need to refer to other rules, follows the

motivation of Example 5-1. Suppose that R_a and R_b, from that example, each refine the

plan by adding a new step followed by a number of refinements r_n that are the same

between Ra and Rb. As written, the DCPOP algorithm would require that the two rules

replicate the list r_n; while this does not introduce a computational inefficiency, it is

certainly an authorial limitation that makes maintaining and debugging the knowledge-

base more error prone. Furthermore, many problems are solvable by combining solutions.

With simple refinement rules, there is no way to explicitly reuse a rule to solve a

subproblem. For example, one can imagine that R_a is a rule to deliver a package within a

city, and R_b delivers packages between cities. To deliver a package between cities, one

first must deliver the package within a city to get the package to an airport, fly the

package to its destination city, and finally deliver the package to its goal location within

the destination city. Naturally, one might attempt to encode the refinements of R_b as first

using R_a (to deliver the package to an airport), then adding a step that flies the package to

its destination city, and finally using R_a once more to deliver the package from the airport

to its final destination. This chapter presents a way to encode and reason on rules of this "hierarchical" form, a methodology that addresses both types of limitations in a single framework.

Example 5-3. Inefficient use of rules from redundant computation.

Another way to motivate the need for hierarchies is to observe that the refinement rules of the previous chapter are naïve – each iteration of the search process must re-evaluate what is true in the plan, regardless of which rules were just applied, and what the application of those rules implies about the state of the current partial-order plan. For example, suppose half of the rules of a given domain exist for the sole purpose of removing unnecessary steps in a plan to be adapted, and half of the rules are for refining that partial plan after doing all removals. Further assume that all of the "retraction" rules have been applied to a particular partial-order plan, and only refinements need be made to complete the partial-plan. The DCPOP algorithm would not be able to skip the evaluation of the rules that remove steps, instead wasting the computation power to recheck conditions that the domain engineer knows are no longer relevant (this same limitation is manifested in Example 5-1 and Example 5-2). What is needed is a mechanism that allows for more informed search, namely the ability to precisely control which rules are to be evaluated at varying points in the search process. In this example, a domain engineer would like to specify that the latter half of the rules should only be applied after all of the first half of the rules are exhaustively applied (and furthermore that once the first-half of the rules have been applied, they should never be re-evaluated).

5.1   HIEPPR-POP methods

This chapter extends the rule formalism and DCPOP algorithm to support reduced precondition evaluation and to introduce the ability of rules to refer to one-another, and in doing so, addresses the limitations shown in the above examples. This new form of domain-configurable knowledge, which I call "HIEPPR-POP methods" (hereafter referred to simply as "methods" where the meaning is unambiguous), is used by my HIEPPR-POP algorithm, which is presented in the next section of this chapter. Because of the similarities in structure of HIEPPR-POP methods to methods as used in HTN planning [Erol et al., 1994], I borrow some of the terminology used to describe HTNs such as "task name", "method", "subtasks", "task reduction", "primitive task", and "precondition-subtask pairs". In spite of the naming overlap, the HIEPPR-POP algorithm does not perform HTN planning (see Section 5.4 for a discussion of how HIEPPR-POP compares to UMCP).

Recall that applicability conditions for domain-configurable planners such as TLPlan or SHOP refer to the state of the world and the consequence of the hand crafted domain knowledge can be either to prune out states (e.g. TLPlan) or to constrain the subsequent applicable HTN methods (e.g. SHOP). For partial-order plans, on the other hand, the expert knowledge should evaluate applicability conditions that refer to the state of the plan rather than the state of the world [Ambite et al., 2001; 2005]. This is an important distinction because given a partial-order plan, computing the valid conditions for any point in the partial plan (that is, all predicates that may be true in the world state immediately preceding the application of an action, relative to all linearizations of the partial-order plan consistent with the ordering constraints) is an intractable problem

[Chapman, 1987]. The DCPOP rules presented in the previous chapter provide exactly this functionality: they allow a domain expert to explicitly tell the underlying partial-order planning algorithm which refinements are applicable to a partial-order plan. HIEPPR-POP methods therefore also use partial-order plan elements (in the form of method precondition evaluation) to constrain their applicability and plan modification. Like DCPOP rules, the effects of HIEPPR-POP methods can modify the elements of the sets of steps, ordering constraints, causal links, and binding constraints representing partial-order plans. Unlike DCPOP rules, methods can constrain which knowledge constructs (methods) are considered in subsequent applications of the expert knowledge (in a manner akin to how SHOP methods constrain the evaluation, or applicability, of subsequent HTN methods in hierarchical task-network planning).

This section formally defines HIEPPR-POP methods; the next section presents an algorithm that uses the methods to make modifications to partial-order plans. In order to provide guiding context for the definitions to follow, I first informally describe HIEPPR-POP methods, deferring the formal presentation of the definitions of each of the parts of the knowledge structure until the essence of a method is conveyed.

```
1. (method task: <task>
2.        preconditions:
3.                [(+/–)<POP plan element>]*
4.        subtasks:
5.                (<subtask>*)
6.        [preconditions:
7.                [(+/–)<POP plan element>]*
8.        subtasks:
9.                (<subtask>*)] )
```

Figure 5.1 The syntax of a HIEPPR-POP method.

A HIEPPR-POP method *m* is written and interpreted as follows: *m* (task *t*, one or more lists of [preconditions *p*, subtasks *st*]). Where *t* is a non-primitive task name used in line

1, the preconditions *p* (lines 2 and 3) are a conjunction of one or more POP plan elements as defined in Section 3.2.2, and *st* (lines 4 and 5) is a list of mixed primitive and nonprimitive "subtasks" (the primitive/non-primitive distinction is clarified below). That is a task *t* is decomposable by a method *m* if and only if all of the following are true: (1) the taskname of *t* and *m* are equivalent; (2) it is possible to unify the arguments of *t* with the parameter templates in *m*; and (3) *m*'s preconditions are true. A task having name *t* may have multiple methods with the same name, each differentiated by its parameters, and list of precondition-subtask pairs.

The <task> *t* can be decomposed, by *m*, into the list of subtasks *st* [<subtask>]* if *t* and *m* share the same name and number of arguments, and if each of the POP plan elements appearing in *p* that are preceded by a plus sign occurs in the current plan and none of the POP plan elements proceeded by the minus sign occur in the current plan. That is, a method that realizes a task is only applicable if, in its precondition section, all the elements with a plus are present in the current partial plan, and all those preconditions with a minus are not. An empty subtask list is evaluated as true, meaning the methods refinements are trivially realized. For methods that have multiple precondition-subtask lists, each precondition list is evaluated, top-to-bottom, until one having all preconditions satisfied is found (at which point the subtasks are applied).

Central to the discussion of HIEPPR-POP methods is the notion of "tasks", which come in two forms: "primitive" and "non-primitive". The semantics of each of these is carefully explained below. First, what is a task, and how does it relate to refinement rules?

Whereas a constant symbol (Definition 3-1) is used to refer to an *object* in the modeled world (e.g. "truck"), and a predicate (Definition 3-5) is used to refer to a *relation* in that world (e.g "at-location"), a taskname is used to identify and refer to a domain-configurable *strategy* (method) for making changes to a plan (e.g. "deliver-package"). That is, a task specifies which strategy is to be used, while a method specifies how the strategy may be employed. In the previous chapter, refinement rules had no need for names, as it was impossible to "invoke" one rule from another. In this and the next chapter, I make the case that the addition of tasknames (and algorithmic changes to support them) is one way to overcome the limitations of DCPOP introduced at the beginning of this chapter (Example 5-1 through Example 5-3).

The reader may wish to think of a HIEPPR-POP method as a DCPOP rule with a name – the similarities between the two formalisms will make for an easier understanding of the definitions to follow, while their differences will lend insight into how the representation will effect an algorithm using HIEPPR-POP methods.

Definition 5-1. A HIEPPR-POP **atom**, extends the definition of the classical planning atom (Definition 3-6) to not only include statements of fact about the modeled world (e.g. '(at truck1 ?loc)', a predicate symbol with arguments matching its arity (Definition 3-5)) but to also include statements of fact about the properties of a partial-order plan derivable in that domain – plan elements as in Definition 4-7 (e.g. 'Sx < Sy', where Sx and Sy are steps). As with classical atoms, if all arguments are grounded, then the atom is also grounded. See the discussion following Definition 4-18 on representing plan elements.

Definition 5-2. A HIEPPR-POP **term** is a HIEPPR-POP variable symbol, HIEPPR-POP constant symbol, atom (Definition 5-1), or assignment expression (Definition 5-8).

Definition 5-3. A HIEPPR-POP **task-symbol** *s* is a constant symbol (Definition 3-1), and is used to unify tasks (Definition 5-4) with methods (Definition 5-7) that accomplish them. The term **task-name** may be used interchangeably. Symbol *s* can be a **primitive task-symbol** (Definition 5-5) or **non-primitive task-symbol** (Definition 5-6).

An example task-symbol is "`deliver-all-packages`".

Definition 5-4. A HIEPPR-POP **task-atom** *t* is a character sequence having the form (*s t1 t2 ... tn*) where *s* is a task-symbol (Definition 5-3), and the arguments *t1 t2 ... tn* are HIEPPR-POP terms. The task-atom is **primitive** if *s* is a primitive task-symbol, and non-primitive if *s* is a non-primitive task-symbol.

An example task-atom with an arity of two is "`(deliver-package-for-step p1 ?drivestep)`"; note that the task-head `deliver-package-for-step` is not preceded by an exclamation point symbol as is done with the heads of actions.

Definition 5-5. A HIEPPR-POP **primitive task** *t* is syntactically a '!do' or '!undo', followed by any POP plan refinement (Definition 4-15) that might be applied to a partial-order plan generated by a classical partial-order planning process, namely any planning activity that modifies the elements defining a partial-order plan: the addition or removal of a plan step, the addition or removal of a causal link, the addition or removal of an ordering constraint, and the addition or removal of a variable binding constraint.

Continuing the analogy of HIEPPR-POP methods being akin to named DCPOP rules, a HIEPPR-POP primitive task is akin to a DCPOP refinement rule effect (Definition 4-19).

The syntax (with the minor change of prefixing the task-head with an exclamation point symbol rather than appending a colon to it) and semantics are as indicated in Definition 4-19 and the discussion that follows it.

Borrowing the same convention as used in SHOP, a primitive task is made more easily identifiable by preceding the task-head of t with an exclamation mark. Because a primitive task refers to an irreducible, fundamental aspect of partial-order planning operations, primitive tasks require no domain expert to define their behavior and as such are available in any HIEPPR-POP planning domain.

The following are the six primitive tasks allowed in HIEPPR-POP; only three of six are shown, given that the other three (noted by "!undo" rather than "!do") are opposite in function:

- (!do add_step <s> <o>), adds a new step <s> to the plan that is an instantiation of the operator <o> relative to the parameters specified.

- (!do add_link <s1> →<c> <s2>), adds the causal link indicated by the parameters.

- (!do add_order <s1> → <s2>), adds the ordering constraint indicated by the parameters.

For example the primitive task (!do add_step s_1 (load ?p ?t ?l)) adds a step s_1 corresponding to an instantiation of operator (load ?p ?t ?l) in the transportation logistics domain. As before, and by convention, a name preceded by a question mark indicates a variable (for example, *?p* denotes the variable *p*). The term $s_{<string>}$ denotes a variable for steps. I adopted this convention to improve readability but it is still strictly expressible in first-order logic. So in the example s_1 denotes the variable assigned to the step added for

114

the action *(load ?p ?t ?l)*. The scope of a variable is limited to the method in which it occurs.

A non-prim task is a task for which there exists a method specification in domain with an equivalent task-head.

Definition 5-6. A HIEPPR-POP **non-primitive task** is a task (Definition 5-4) for which there exists a unifiable HIEPPR-POP method definition in the HIEPPR-POP domain file. That is, whereas primitive tasks (Definition 5-5) are "built-in" to any underlying POP planning algorithm, non-primitive tasks require a correspondingly named method defined in the input domain file.

Definition 5-7. A HIEPPR-POP **method** m is a triple *HPM* = (*taskname*, *parameter-templates*, *precondition-subtask-pairs*) in which the **task-head** *taskname* (Definition 5-3) is a constant symbol (Definition 3-1) used for identification and reference purposes, *parameter-templates* is a list of parameter templates (Definition 5-8) that allow for the passing of information between methods, and *precondition-subtask-pairs* is a list of pairs *psp* = (*mp*, *st*) where *mp* is a list of method preconditions (Definition 5-9) and *st* is a list of subtasks (Definition 5-10).

The pairs of *psp* work together to constrain when and which methods get evaluated, and how a plan is to be modified. Figure 5.1 presents a schema for methods which may be a useful guide to the coming definitions.

Definition 5-8. A HIEPPR-POP **assignment-expression** *a* is syntactically a variable symbol followed by the symbol '=', followed by a HIEPPR-POP atom (Definition 5-1).

115

Definition 5-9. A HIEPPR-POP method **precondition** *hmp* is syntactically a DCPOP refinement rule precondition (Definition 4-18) *rrp* = (+|−)<POP plan element>, the constant "`true`", the constant "`false`", or an expression term that is evaluable to the values true or false.

Semantically, a method precondition *hmp* can be either logically true or logically false, relative to a partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$, and the set of flaws *F* extant in ρ. The precondition *hmp* evaluates to logically true in the following situations, and false otherwise: (1) *hmp* is a DCPOP refinement rule precondition *rrp* that evaluates to true − that is, in this case, the syntax and semantics are the same as in Definition 4-18 (the plan element must be an element of ρ or *F* if preceded by a plus, and must not be present in ρ nor *F* if preceded by a minus); (2) *hmp* is the constant `true`. The constant `false` always evaluates to false.

Applicability conditions. HIEPPR-POP provides a number of built-in applicability conditions, that is, the precondition elements of *p*, to facilitate encoding domains; these conditions are the same as defined in the previous chapter. The core of the theory is that reasoning on the essential logical elements of set membership of a partial plan is sufficient. As before, the precondition *(parallel s'' (link s' s p))* indicates that *s''* can be consistently ordered between *s'* and *s* given the plan's ordering constraints. While this is not directly a member of the essential sets required for capturing POP plan elements it can be derived from the members of those sets (plan elements); HIEPPR-POP provides built-in procedures to quickly check this, and other conditions that can be derived from the essential sets. Another precondition for which I have built specialized checking

116

procedures include *(equal ?x ?y)* (or *(different ?x ?y))* indicating that two variables must have the same (or different) value.

The most notable of the preconditions with specialized checking procedures is the relation *s < s'* between any two steps, which indicates that for any linearization of the plan, step *s* will occur before step *s'*. HIEPPR-POP uses a data structure to quickly compute transitive closure of the orderings. HIEPPR-POP also has a primitive condition *(threat C s)* which checks if a step *s* is a threat to the causal link *C*. This special treatment is justified by the observation that checking step ordering is a common operation to do on a partial plan with many pending refinements.

Definition 5-10. A HIEPPR-POP method **subtask** *hms* is syntactically a DCPOP refinement rule effect (Definition 4-19) *rre* = (do:|undo:) <POP plan refinement>, or a HIEPPR-POP task-atom.

The semantics of a method subtask are defined relative to a partial-order plan $\rho = (S, \rightarrow, \rightarrow_{CL}, B)$, and the set of flaws F extant in plan ρ as follows: (1) if subtask *hms* is a DCPOP refinement rule effect *rre*, then it is **applied** to ρ – that is, in this case, the syntax and semantics are the same as in Definition 4-19 (briefly restated, the plan refinement is performed on ρ when preceded by a 'do:', and is retracted from ρ when preceded by an 'undo:'); (2) if subtask *hms* is a HIEPPR-POP method taskname

Definition 5-11. A HIEPPR-POP **planning problem** $HPP = (\Psi, HPD)$ is tuple containing a classical planning problem $\Psi = (\Delta, s_0, g)$, where $\Delta = (C, P, O)$ is a classical planning domain, along with a HIEPPR-POP planning domain $HPD = (\Delta, HPM)$.

5.2   The HIEPPR-POP Algorithm

Figure 5.2 presents the pseudocode of the *HIEPPR-POP* algorithm. It begins by constructing an initial partial plan for the input classical planning problem (S,G) if π_{old} is null (generative planning), otherwise it assigns to π the input partial plan (Line 1 and 2). It then calls the HTNDecompose procedure (Line 3). If this procedure returns a partial plan with no flaws then this plan is returned (Lines 4-5). Otherwise the partial plan is completed by first principles (Line 6).

Procedure HIEPPR-POP (S, G, T, A, M, π_{old})
//**input**: state S, goals G, task list T, actions A, HIEPPR-POP methods M, partial-plan π_{old}
//**output**: a complete plan π
1. **if** π_{old} is null **then** $\pi \leftarrow$ initialplan(S,G)
2. **else** $\pi \leftarrow \pi_{old}$
3. $\pi \leftarrow$ HTNDecompose(π, T, A, M)
4. **if** flaws(π) $= \varnothing$ **then**
5.   **return** π
6. **else return** first-principlesPOP(π)

Procedure HTNDecompose(π, T, A, M)
1. **if** T $= \varnothing$ **then return** π
2. $\tau \leftarrow$ firstTask(T); T' $\leftarrow$ reminderTasks(T)
3. **if** τ is primitive **then**
4.   **If** refinement τ is applicable **then**
5.   perform-refinement(τ,A) on π
6.   **return** HTNDecompose(π, T', A, M)
7.   **else return** π
8. **else**
9.   find applicable method m in M for τ wrt π
10.   **if** there is no applicable method **then**
11.   **return** π
12.   **else**
13.   obtain subtasks τ' of m wrt π and τ
14. $\pi' \leftarrow$ HTNDecompose(π,τ', A, M)
15.   **if** $\pi' = \pi$ **then return** π
16.   **else return** HTNDecompose(π',T', A, M)

Figure 5.2 Pseudo-code of HIEPPR-POP

118

The *HTNDecompose* procedure checks if *T* is empty, in which case the input plan is returned (Line 1). It then gets the first task τ and the remaining tasks *T'* of *T* (Line 2). If the first task is primitive and its refinement is applicable to the plan (Lines 3-4), then the refinement is applied to the plan and a recursive call is made with the remaining tasks and the refined partial plan (Lines 5-6). If the refinement is not applicable then the input partial plan is returned unmodified (Line 7). If the first task is compound, then applicable methods are sought to decompose it (lines 8-10). If no method is applicable the partial plan is returned unmodified (Line 11). Otherwise, the task is decomposed and a recursive call is made with the resulting tasks τ' (Lines 13-14). If the recursive call returns the same partial plan, the process is terminated and the partial plan is returned unmodified (Line 15). Otherwise, a recursive call is made with the refined partial plan obtained from Line 14 and the remaining tasks.

Plans generated by HIEPPR-POP correctly solve the problem (S,G). The reason is that each of the primitive tasks performs valid POP refinements (that is, HIEPPR-POP is sound). However, no guarantees can be given that HIEPPR-POP will terminate. This depends on the POP methods provided (e.g., using the POP methods may result in an infinite loop). A similar observation can be made for any domain-configurable planner.

Similarities and differences versus HTN planning. The task decomposition process of HIEPPR-POP is very similar to the one from an HTN planner such as SHOP. This is the result of HIEPPR-POP methods having similar semantics to SHOP methods; the latter determines the preconditions' applicability by checking them in the state of the world whereas the former checks them for existence in the current partial-order plan. Another

similarity is that primitive tasks indicate transformations to the state of the world in the former and to the partial-plan state in the latter. Their main difference is that whereas in SHOP the primitive tasks denote domain actions, in HIEPPR-POP primitive tasks denote partial plan refinements, some of which are the result of the domain actions (e.g., adding a new step in the partial plan) while others are the result of the mechanics of POP planning (e.g., adding an ordering constraint). Semantically, SHOP and HIEPPR-POP are quite different; determining if a plan π generated by SHOP is a solution for a problem is a function of whether π is entailed by the HTN generated by SHOP. In contrast, determining if the partial plan π is a solution for a problem is independent of the HTN used by HIEPPR-POP to generate it. Like any POP plan, π is a solution if it is complete. This allows HIEPPR-POP to continue doing first-principles planning when an incomplete partial plan is returned by the task decomposition process (Line 6 of HIEPPR-POP). This is not possible with a plan generated by SHOP; the HTN methods in SHOP are required to fully model the domain.[3]

5.3   An Example

Figure 5.3 shows an example of two methods for generative planning in HIEPPR-POP for the blocks world domain. This particular encoding unstacks all blocks to the table and then proceeds to stack them. The first method decomposes the task *(unstack-stack)*, has no preconditions, and generates two subtasks *(unstackAll)* and *(stackAll)*.

---

[3] A way around this limitation of SHOP is explained in Alford et al. (2009). It translates SHOP's methods into PDDL and uses a planner based on PDDL to obtain solutions.

```
(1)
(:method  (unstack-stack)
   :preconditions ()
   :subtask  ( (unstackAll) (stackAll) ) )


(2)
(:method (unstackAll)
   :preconditions
       ( (s₀→(on ?a ?b) )
         ( ?effClearA = (s₀ →(clear ?a) ) )
         ( not (sᵢ →(ontable ?a) )
         ( ?effHE = s_handEmpty  →(handempty) )
         (not (link s_handEmpty →(handempty)  s_other ) ) )
   :subtasks
       ( (!add_step s_unstackAB (unstack ?a ?b) )
         (!add_link s₀→ (on ?a ?b)  s_unstackAB )
         (!add_link ?effClearA s_unstackAB )
         (!add_link ? effHE s_unstackAB )
         (!add_step s_putDownA (put-down ?a) )
         (!add_link s_unstackAB →(holding ?a) s_putDownA)
         ( unstackAll_startingFrom ?b s_unstackAB )
         ( unstackAll ) )

   :preconditions ()
   :subtask () )
```

Figure 5.3 Sample methods in Blocks World

The second method has two precondition-subtask pairs, which can be thought of as an

if-then/else-if. The first if-then pair operates as follows: the effects of the initial step are

queried for an effect matching *(on ?a ?b)*; the second precondition ensures *?a* is the top-

most block on a stack in s_0, and the matching effect is bound to the *?effClearA* variable.

The next precondition makes sure there is no step s_i placing *?a* on the table, as a means of

avoiding the addition of steps that would threaten one-another. In order to move *?a* to the

table, the hand must be empty, so *?effHE* is bound to a step producing this effect. The

next condition ensures the effect *?effHE* is not used to support a precondition of another

step in the plan, which could be the case if there are multiple stacks in the problem's initial state.

With the preconditions of the first if satisfied, it's time for reducing the sub-tasks in the "then" part. The first six subtasks are primitive. The first four subtasks modify the current partial-order plan by adding an *unstack* step, with appropriate links supporting its preconditions. The variables bound in the if-part (e.g. *?a*, *?effClearA*, and *?effHE*) are used for this purpose. The next two primitive tasks add a step that puts *?a* on the table and a link supporting its *(holding ?a)* precondition. The compound task *(unstackAll_startingFrom ?b $s_{unstackAB}$)* has a matching method, not shown in this example, which intuitively resumes the unstacking process starting from block *?b*, which was just made clear by $s_{unstackAB}$. After this subtask is decomposed, the task *(unstackAll)* is then recursively invoked, in order to accommodate problems having multiple stacks of blocks in the initial state.

When it is the case that there remain no more stacks to unpile, the preconditions of the first if-then cannot be satisfied; at this time the second preconditions-subtasks pair is used. Because an empty precondition list and an empty subtask list is evaluated as true, the task *(unstackAll)* matched to this method is accomplished and the state of the partial-order plan is such that all blocks are on the table, and the hand is empty. At this point the second subtask, *(stackAll)*, of the *unstack-stack* method is used to satisfy the open-conditions of the goal state.

5.4   Comparison of HIEPPR-POP to UMCP

Having presented the way in with HIEPPR-POP uses hierarchies to search for solutions, one might wonder how the algorithm is related to UMCP (Erol *et al.*, 1994b). The UMCP algorithm, which stands for Universal Method Composition Planner, is a sound and complete HTN planning algorithm. It is a seminal contribution in that it was the first formulation of clear and concise syntax and semantics for HTN planning. Given that HIEPPR-POP performs a type of task reduction planning, it is natural to wonder its similarities to and differences from UMCP.

UMCP decomposes a task into a "plot", which indicates the resulting subtasks and their constraints including ordering constraints and causal links. Like POP, UMCP can generate partially ordered plans. In UMCP, there are three types of tasks: (1) goal tasks, which are propositions to make true in the world [e.g. (at pac1 loc1)], (2) primitive tasks, which correspond to the addition of the domain action to the plan [e.g. (!drive-truck t1 loc1 loc2)], and (3) compound tasks, which are to be satisfied through the accomplishment of any combination of goal, primitive, and compound tasks [e.g. (deliver-all-packages)]. The means of accomplishing a compound task, like HIEPPR-POP, is specified by defining a correspondingly named method or methods. It is in the specification of methods, and their evaluation, that the two algorithms differ.

```
(declare-method task (arg1 arg2 ..)
        :expansion ( (label1 task1 arg11 arg12 ..)
                     (label2 task2 arg21 arg22 ..) ..)
        :formula Constraint-Formula)
```

Figure 5.4 Syntax for defining methods in UMCP

In UMCP, a method is defined as shown in **Figure 5.4**. The "expansion" of a method corresponds to how the task should be accomplished via any combination of goal,

primitive or compound tasks. Thus, in HIEPPR-POP parlance, the expansion label defines the subtasks of the method being defined. The "formula" section of a method definition in UMCP imposes constraints on the successful (correct) use of the method to accomplish the task, and is specified through a "constraint-formula"; in HIEPPR-POP, the preconditions of a method are used to impose constraints on when it is valid to use a method to accomplish a task. The constraints on the valid application of a method to accomplish a task is where UMCP and HIEPPR-POP differ.

A constraint-formula in UMCP is evaluated to be logically true or false, and is either the symbol indicating trivial satisfaction (the Boolean value "true"), a single constraint, or constraints connected by the Boolean operators AND, OR, and NOT (with the typical interpretation of these terms). A single constraint must always evaluate to logically true or false, and can be any of the following: (1) "(veq v1 v2)", or "(veq v1 c1)" – the existence of a binding constraint indicating a variable is equal to another variable or constant symbol defined in the domain, (2) "(ord n1 n2)" – the existence of an ordering constraint placing one task (primitive or otherwise) before another, (3) "(initially p)"– the existence of a predicate in the initial state of the problem to solve, (4) "(before p n1)" or "(after p n1)" – the assertion that a predicate exists in the world state immediately before (or after) the indicated task is ordered to occur in any valid linearization of the plan, (5) "(between p n1 n2)" – the assertion that, if task n1 comes before task n2, the indicated predicate exists in every world state between those two tasks in every valid linearization of the plan, and (6) "(protect p n1 n2)" – the assertion that no task occurring between n1 and n2 in a valid linearization of the plan have as effect the predicate p.

In UMCP, in order to correctly use a method to accomplish a correspondingly named task, the modification of the plan as specified by the expansion must conform to the constraint formula of the method being applied. That is, after decomposing the task using the specified expansions, the constraint formula must evaluate to true against the newly refined plan (as must all task expansions applied along the way). This differs in several important ways from how task reduction is performed in HIEPPR-POP.

The most important difference in task reduction between the two algorithms is in how the conditions governing the correct application of a method to a task are evaluated. In UMCP, the validity conditions are used to ensure that whatever modifications have been made to the plan being refined must conform to the specified formula. That is, one can see the conditions of a UMCP method as "filters", whereby refinements are first made to a plan according to the expansions, and then any plans that do not meet the specified conditions are discarded. In this respect, the validity conditions of UMCP may be read as "post-conditions" (things to be true *after* performing task reduction), whereas in HIEPPR-POP the validity conditions governing successful application of a method are "pre-conditions" (things that must be true *before* performing the task reduction). The other differences in validity conditions between UMCP and HIEPPR-POP have to do with the types of formulas that can evaluate to true. The following is a list of similarities and differences between the conditions evaluated in the two algorithms:

- Variable equality (and inequality) are the same in the two algorithms.
- In UMCP, the ordering constraint manager not only tracks constraints on primitive tasks (domain actions) but also constraints that have been added with respect to non-primitive actions (tasks); in HIEPPR-POP, ordering constraints

125

only exist for domain actions. Therefore, in UMCP, the ordering constraint manager has more information to evaluate against than in HIEPPR-POP.

- In UMCP, while it is possible to query whether a predicate is asserted by the initial state, it is not possible to find any other step in the plan that asserts that predicate (as is possible in HIEPPR-POP). Specifically, while it is possible to query whether a predicate is true in the world state immediately before or after a task in UMCP (this is also possible in HIEPPR-POP by using a combination of ordering constraint conditions and step add conditions), it is not possible to determine which step is the provider in UMCP, if the provider is not the initial step.

This chapter has shown how the addition of hierarchies to the refinement knowledge addresses some notable limitations of the DCPOP algorithm, by creating a new algorithm called HIEPPR-POP – for HIErarchical Partial Plan Refinements for Partial-Order Plans, which subsumes the DCPOP approach (all domains that can be encoded for DCPOP can be encoded in HIEPPR-POP, but not the other way around). The two main limitations addressed by this chapter were the encoding and use of refinement rules that share a prefix of refinement rule preconditions, and rules that share common refinement strategies. The next chapter presents and empirical evaluation of both algorithms.

6   Empirical Evaluation

This chapter presents three sets of experiments designed to evaluate the efficacy of the domain-configurable algorithms and knowledge structures presented in the previous two chapters.

The first subsection presents an evaluation of DCPOP, the non-hierarchical form of the domain configurable partial-order plan refinement algorithm. Because the "grand challenge" of this dissertation was to try and solve the plan adaptation problem, the empirical evaluation focuses on how DCPOP performs at this task. While the results of the first evaluation reveal an interesting contribution of the work (namely, the retrieval-adaptation tradeoff held as unshakable in the case-based reasoning community did not apply to DCPOP), they are nevertheless indicative of yet another attempt at plan adaptation that met only limited success.

The second subsection presents the results of taking the lessons learned from designing, analyzing and evaluating DCPOP, in order to make the hierarchical form of the domain configurable partial-order plan refinement algorithm HIEPPR-POP and attendant knowledge structures. Once again, like many researchers before me, I was unable to find success at the plan adaptation problem. However, a startling result is that the algorithm, when operating with complete and carefully crafted expert knowledge, is able to perform comparably to SHOP1, in terms of time taken to find a solution. An important caveat to this result is that while the solutions produced by SHOP1 are near optimal in blocks-world (in terms of plan-length), I made no attempt in any of the three

analyzed domains to produce short plans by HIEPPR-POP. In general, the expert knowledge I encoded in order to achieve the fast planning performance reported does not follow the same criterion for solution generation as a typical planner, which work hard (through plan selection and flaw selection heuristics) to keep solution plans short. For more details, see subsection two.

The final subsection of this chapter presents the results of empirically evaluating how HIEPPR-POP performs without complete knowledge. The ability to specify only a "sketch" of how to solve a problem, in the form of incomplete HIEPPR-POP methods, is an attractive goal. This would permit the domain engineer to encode only the most vital details of how experts solve problems (or exhibit preferences for solution types), leaving the more mundane "bookkeeping" tasks to the underlying planner. Unfortunately, like others who have attempted to use partial-order planning to solve planning problems of even moderate size, I found the approach to be completely unsuccessful. Nevertheless, the state of the art in planning continues to find improvements to plan ranking and flaw selection heuristics, and as these techniques advance, the goal of supporting "sketching" solutions becomes ever more realizable.

6.1 DCPOP: Adaptive Planning (small problems)

I first used the non-hierarchical version of HIEPPR-POP (named DCPOP) to not only evaluate empirically the feasibility of my domain-configurable adaptation approach, but to also revisit the trade-off between adaptation and retrieval effort traditionally held as a principle in case-based reasoning. This principle states that the time needed for adaptation reduces with the time spent searching for an adequate case to be retrieved. In

particular, if very little time is spent in retrieval, the adaptation effort will be high. Correspondingly, if the retrieval effort is high, the adaption effort is low. I analyzed this principle in two boundary conditions: (1) when very bad and (2) when highly capable adaptation procedures are used. The results showed that in the first boundary condition the adaptation-retrieval trade-off does not necessarily exist. I also claimed that the second does not hold for a class of planning domains frequently used in the planning literature. To validate this claim, I performed experiments on two domains of this type. I used DCPOP in order to investigate the adaptation-retrieval trade-off in a system capable of performing "omniscient" search with ideal inputs, thereby providing a suitable framework in which to re-evaluate the adaptation-retrieval trade-off.

For the purposes of this section, I refer to HIEPPR-POP methods as "POP Rules". This is to highlight the fact that no non-primitive tasks were used in the methods input to the system. I defined two classes of POP rules: regressive rules and progressive rules. **Regressive rules** indicate POP plan elements that must be removed from the plan. As a result, they always have the *undo:* label in the consequent part of the rule. ***Progressive rules*** indicate POP plan elements that must be added to the plan. As a result, they always have the *do:* label in the consequent part of the rule. This distinction facilitates the systematic search performed by the adaptation algorithm.

Recall that DCPOP receives as input the initial state, goal state, and actions. It also receives the plan to be adapted, π_{old}, and the POP rules R. The output is a complete partial plan solving (S,G,A) or fail if none is found. DCPOP begins by adjusting π_{old} relative to (S,G). Adjust plan works by repeatedly (1) removing a step s that mentions objects in the retrieved plan that are not mapped into objects in the new problem, and (2) removing any

ordering constraint or causal link connecting to/from s. This is a common step for adaptation in first-principles POP planning (e.g., [Hanks & Weld, 1995], [van der Krogt & de Weerdt, 2005], [Kuchibatla & Munoz-Avila, 2006]). Then, a set of plans is found by repeatedly applying regression rules in R until none is applicable. These plans are added to P, the list of current candidate plans to be refined. When the list of candidate plans is empty, a failure is returned. While there is at least one candidate plan to be refined and no solution has been found, the following computation occurs: at each iteration, a candidate partial plan π is selected using the heuristics and is removed from P. If this candidate partial plan has no flaws, it is returned. Otherwise each partial plan computed by applying an applicable POP rule to π is added to P. If no domain configurable refinements are found, standard POP refinements are added to P (lines 11 and 12). In principle, DCPOP-A could use any relevant partial plan and flaw selection heuristics described in [Younes & Simmons, 2003] for lines 4 and 11; however my implementation uses last-in-first-out selection (a stack)for both plans and flaws. This proved sufficient for resolving the flaws remaining in the plans produced after applying all domain-configurable solving knowledge, however it is exceptionally unlikely for all but toy domains that a domain engineer would be able to manually author rules that leave flaws whose resolution require no backtracking.

(1)	(3)
if $+ s_0 \rightarrow$ (on ?x ?y)	if $+ s$: (move ?x ?y table)
$+ s_0 \rightarrow$ (block ?y)	$+ s_0 \rightarrow$ (block ?y)
$- s \rightarrow$ (on ?x table)	$+ s_0 \rightarrow$ (on ?x ?y)
then do: s': (move ?x ?y table)	$- s_0 \rightarrow$ (on ?x ?y) s
do: $s_0 \rightarrow$ (on ?x ?y) s'	then do: $s_0 \rightarrow$ (on ?x ?y) s
(2)	(4)
if $+ s$: (move ?x ?y table)	if $+ s$: (move ?x table ?y)
$+ s'$: (move ?z table ?w)	$-$ ((on ?x ?y) @ s_∞)
$- s \rightarrow s'$	then undo: s:(move ?x table ?y)
then do: $s \rightarrow s'$	

Figure 6.1 POP rules partially encoding the unstack-stack strategy

Figure 6.1 shows an example of POP rules in the Blocks World domain. The blocks world is a puzzle-like domain in which piles of blocks on a table must be reconfigured into a target configuration. The basic restriction is that blocks can only be moved either from the top of a pile to the top of another pile or to the table. Clearly, there is a high degree of goal interaction – achieving one block on another may require undoing a stack that successfully achieved some other set of goals. This problem is easy for humans, solvable by babies, but can be hard for planners. These POP rules I created for this domain encode the common domain-configurable strategy, called ***unstack-stack***. This strategy first unstacks all blocks to the table and then stacks them in the required configuration (yielding an "omniscient" plan adaptation algorithm).

The first POP rule unstacks block *?x* to the table. The first two conditions check if block *?x* is on top of another block *?y* in the initial state. The third condition checks that no existing step unstacks *?x* to the table. This rule makes two refinements: it adds a step *s'* unstacking *?x* to the table and adds a causal link connecting the step s_0 to achieve a precondition of *s'*. The second POP rule ensures that unstacking steps (e.g., step *s*) are done before stacking steps (e.g., step *s'*). The third POP rule is intended as a refinement

131

of an input partial plan so that it commits to the encoded strategy. It checks if a block (*?x*) that is unstacked by a step *s* is linked to the condition (*on ?x ?y*) in the initial state. If it is not, it adds a causal link connecting the condition and *s*. This rule can be triggered in situations where in the initial state of the retrieved partial plan, block *?x* was on top of a block *?y* and later in that partial plan *?x* was unstacked to the table by a step *s*. This plan would not have been generated by the strategy encoded in **Figure 6.1**. The fourth POP rule is a regression rule. It removes any stacking step from the table that does not achieve a goal.

Any step removed by the fourth rule does not need to be added back because in the stack-unstack strategy, blocks are stacked only to achieve goals. After all these steps are removed, the four POP refinement rules of **Figure 6.1** will produce incomplete partial plans that can be further refined by a first-principles process without backtracking on any of the refinements made by applying these rules. Note that this means that, for any given flaw remaining in the plan, any resolution to that flaw need not be backtracked over. This is a highly desirable property as in some domains it might be difficult to obtain a collection of POP rules that produce a complete plan (for instance, by anticipating and resolving threats before they arise). Consequently, rules can be given for the more computationally complicated details (e.g., how to achieve the goals), leaving the rest to standard partial-order planning. Ideally, the intermediate partial plan produced from adaptation will be easier to complete than the initial partial plan. The unstack-stack strategy, partially encoded in **Figure 6.1**, can be fully encoded to ensure that the resulting partial plans are complete. Furthermore, no backtracking will be needed during the plan adaptation process. Hence, when used in DCPOP, these rules resulted in an omniscient

plan adaptation algorithm. This was confirmed in the experimental evaluation – no backtracking occurred in any of the plan adaptation instances. Not all of the POP rules are presented, for the sake of document readability.

In addition to performing experiments in the blocks-world domain, I also evaluated the DCPOP algorithm by encoding rules for the logistics transportation domain. In the logistics transportation domain, packages must be relocated into target locations. While the goal interaction is limited in this domain (unlike block world, one can safely focus on achieving the goal one package at a time), the nature of the problem makes knowing the world state very useful. There are two transportation means: trucks, which can be used to relocate packages within locations in the same city, and airplanes, which can be used to relocate packages that are in different cities.

For each domain I constructed a case base of 100 cases and a testing set of 10 problems. All problems have the same goals but their initial state is randomly generated. For the blocks world the goal is to achieve a 5-block pile and for logistics a particular configuration of 4 packages required to be at 4 different locations. The initial state for the blocks world is a configuration of the 5 blocks. So the total number of problems that can be generated is 501. The initial state for the transportation domain is a configuration of 3 cities, each having 3 locations (including 1 airport), each city has 1 truck and there are 2 airplanes. So the total number of problems that can be generated, given that the packages can start in any of the 9 locations and that the start locations of the trucks and airplanes are fixed, is 6561. For each problem p in the testing set I adapt each of the cases c stored in the case base. I ran each problem-case pair (p,c) 30 times and averaged the results. So the total runs for each domain was $10 * 100 * 30 = 30,000$ runs. Even though these

problems are comparatively "small", with respect to the size of problems solvable by domain configurable *generative* planners, these problems are nevertheless present sufficient complexity to the state of the art in plan *adaptation*.

Fixing the goals is a simplifying way to simulate how retrieval algorithms would work with an omniscient adaptation algorithm. Namely, any case is retrieved that achieves the same goals regardless of the similarity. In experiments reported in [Munoz-Avila & Hullen, 1996], it is shown that modifying features in the initial state can result in a significant change in the adaptation process on top of a partial order planner. For historical context, in Prodigy/Analogy [Veloso, 1994] retrieval occurs by iterating over two steps. At the first step the system uses a hash table to identify if there are cases stored achieving the same number goals and then, in the second step, computes similarity based on the initial state. If a sufficiently similar case is found (e.g., the similarity of the initial states is greater than a pre-defined threshold) then the case is retrieved. Otherwise it repeats the two steps by removing one goal. With an omniscient adaptation algorithm the second step would be unnecessary. A similar process to Prodigy/Analogy is performed in CAPlan/CbC and derSNLP.

Figure 6.2 shows the run-time results for the blocks world (left) and the logistics transportation domain (right) respectively. The x axis corresponds to the 100 cases * 10 problems and the y axis correspond to the average time in seconds (not log scale) over the 30 runs for each (case, problem) adaptation process. The x-axis is sorted so the first 100 averaged data points are shown with the first given problem, then again the next 100 points with the second given problem, and so forth. Thus, the vertical bars in the graphs separate data for each of the 10 problems; between those bars (i.e., for a given problem),

the data points show the averaged times to adapt each of the cases in the CB into a solution for the problem.

In the blocks world domain, I observed that the running times for adapting each case to a given problem is clustered around the same time intervals. For example, for the 4th problem the average time to adapt all cases is 0.155 seconds with a standard deviation of 0.012 seconds. Similar results were observed across all other problems.

In the logistics domain, there is no significant time difference between solving times across all given problems; the average problem solving time, across all pairs (case, problem), is 9 seconds with a standard deviation of 0.5 seconds. The results for both domains support the hypothesis that regardless of which any two cases are retrieved for a given problem, their adaptation times will be roughly the same, regardless of the individual cases similarity to the new problem, and given that the problems solved by the cases and the new problem are of the same size.
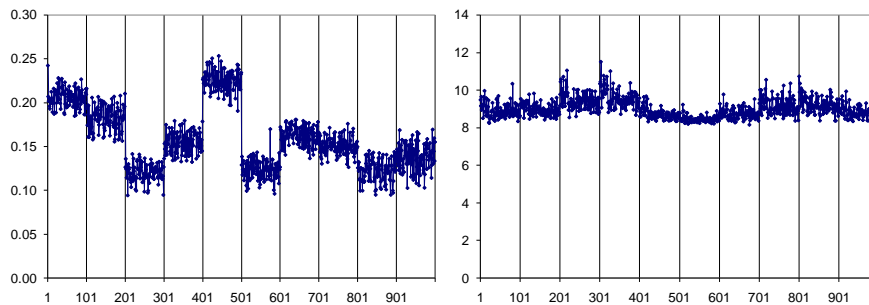
Figure 6.2 Adaptation times for blocks world (left) and logistics (right)

In spite of showing that the DCPOP algorithm can overturn the long held adaptation-retrieval tradeoff that has been accepted as unshakable in the case-based reasoning literature, the results are nevertheless underwhelming, with respect to the "grand challenge" of plan adaptation (Section 2.1) – the size of problems solvable by DCPOP is

135

small, and the time to adapt them quite high. What was gained from these experiments was an understanding of the need for hierarchies, which led to the extension of the DCPOP algorithm into HIEPPR-POP.

6.2 HIEPPR-POP: Generative Planning, Complete Knowledge

In order to evaluate how the addition of task names effects the run-time performance of the algorithm and its ability to solve large problems, I performed generative planning experiments with HIEPPR-POP on three STRIPS domains, each of which were used in the AIPS-2000 planning completion: the logistics transportation domain, the blocks world domain, and the scheduling domain. The blocks world and logistics domains are well known, and are the same as presented in the previous section. The scheduling domain involves the machining of parts to achieve various shape and surface conditions; parts can be made cylindrical, be painted various colors, and be finished "rough", "smooth," or "glossy". The goals are mostly non-interacting, however actions compete for machine availability. Also, for a single part, some goals "clobber" one-another (for example, making the part cylindrical will undo other properties like a glossy finish).

As a benchmark for HIEPPR-POP, I used the results for the SHOP system reported for the IPC-2000 held at AIPS-2000 (URL: http://www.cs.toronto.edu/aips2000/). The website hosts the problem descriptions, the actions used, and the time results obtained by SHOP running on a 500Mhz Pentium III processor with 1GB of RAM. I ran HIEPPR-POP on a single core, 2.8Ghz Pentium 4 with 1.5 GB of RAM. Hence all CPU time measurements for HIEPPR-POP are multiplied by a factor of 5.6. I acknowledge that this may not be the most ideal evaluation, however it is a useful baseline that ensures no

136

accidental nor intentional experimenter bias (for example, mistakes made in an attempt to re-encode the domains, or to run the SHOP system without any performance tweaks that may have been made for the competition).

For blocks world I encoded the following strategy: all blocks are first unstacked to the table and then stacked back into the desired configuration (**Figure 6.3** shows two of the methods encoding this strategy; observe how the ability to use tasks profoundly changes the encoding from what was shown in **Figure 6.1**). Note that in my unstack-stack solution, even if there exists in the initial state a configuration of blocks (some, or all of them) that matches exactly the configuration required in the goal state, they are unstacked to the table anyway. This is done to sacrifice solution quality (measured by plan-length) for speed to find solution. Solutions produced by SHOP1 are near optimal in blocks-world (in terms of plan-length), which imposes an additional computational cost that HIEPPR-POP does not have to pay; SHOP1 solutions for the other two domains are similarly of high-quality. In general, the expert knowledge I encoded in order to achieve the fast planning performance reported does not follow the same criterion for solution generation as a typical planner (which work hard, through plan selection and flaw selection heuristics) to keep solution plans short. The entire blocks world domain required 1 method for the high-level strategy, 2 methods to perform unstacking of blocks, and 2 methods for stacking. The problem solving times across problems of increasing difficulty (size) are shown in **Figure 6.4**

```
(:method simpleStrategy "Unstack all to table, then build goal stacks."
      :parameters ()
      :preconditions ()
      :subtasks ( ( unstackAll ) ( stackAll ) ) )

(:method unstackAll "Unstack all first. "
      :parameters ()
      ;;unstack, starting from top of a pile
      :preconditions
      (        ( effect s[INIT] (on ?a ?b) ) ;;find ?a on ?b in init
              ( ?effClearA = ( effect s[INIT] (clear ?a) ) ) ;;?a is on top
              ( not ( effect s[i] (ontable ?a) ) ) ;;never made to table
              ( ?effHandEmpty = ( effect s[handEmpty] (handempty) ) )
              ( not ( link s[handEmpty] (handempty) s[other] ) )   )
      :subtasks
      (        ( !add_step s[unstackAB] (unstack ?a ?b) ) ;;?a now in hand
              ( add_link ?lTemp00 s[INIT] (on ?a ?b) s[unstackAB] )
              ( add_link ?lTemp01 ?effClearA s[unstackAB] )
              ( add_link ?lTemp02 ?effHandEmpty s[unstackAB] )
              ( !add_step s[putDownA] (put-down ?a) )
              ( add_link ?lTemp10 s[unstackAB] (holding ?a) s[putDownA] )
              ( unstackAll_startingFrom ?b s[unstackAB] )
              ( unstackAll )  ;;recurse for other stacks        )
      :preconditions ()        ;;we're done unstacking. this method is done.
      :subtasks () )
```
Figure 6.3 Two HIEPPR-POP Methods Encoding Unstack-Stack
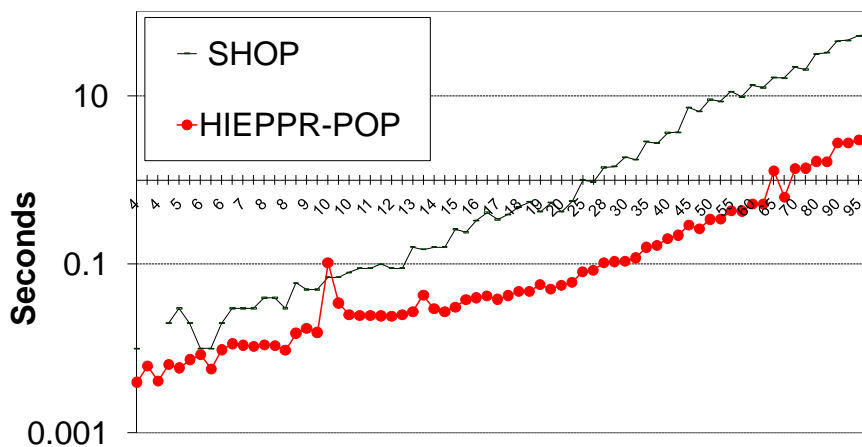
Figure 6.4 Blocks World Time Comparison (log scale)

138

For the logistics transportation domain I created a simple strategy whereby each package is transported to its destination independently of the other packages. To make task reduction "chain" more effectively, which made it far easier to author the domain (at the expense of plan length), I always add an additional step after flying or driving to ensure there is at all times an unused 'at-location' for the vehicle in question. This was by far the most difficult domain to encode a complete set of domain knowledge, even after adding the restrictions of handling each package independently of the other packages, and always inserting an extra drive or fly; my encoding required 10 methods total, 1 for the high level strategy, 1 for determining the object type of the 'at' (ie, package, truck, or plane), 6 methods (each with precondition/subtask pairs) to establish the 'at' of packages, 1 method for moving trucks, and 1 method for moving planes. The problem solving times across problems of increasing difficulty (and size) are shown in **Figure 6.5**. It is an interesting implementation aside that this experiment instigated my move to support typing, because of the heavy bookkeeping required in an untyped domain (for example, there are 36 primitive subtasks in one of the methods, many of which for supporting types).
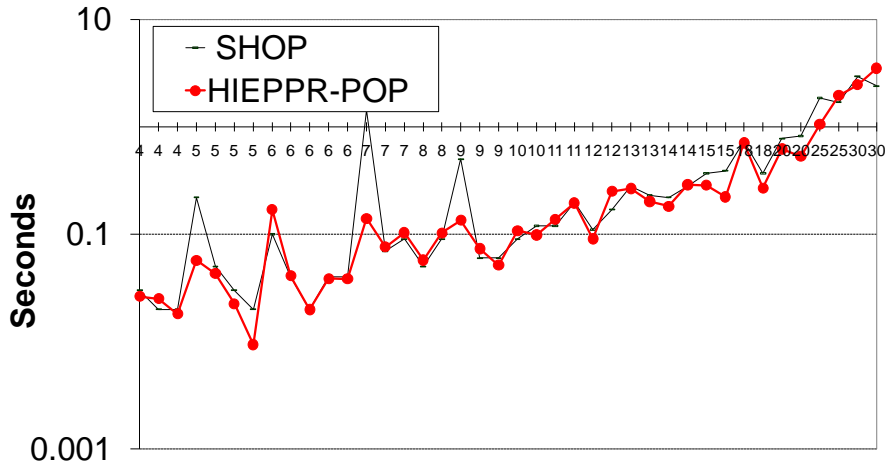
Figure 6.5 Logistics Time Comparison (log scale)

For the scheduling domain I first achieve all cylindrical goals, then surface condition goals, and finally paint the parts appropriately. Doing so required 1 method for the high-level strategy, 2 methods for making parts cylindrical, 5 methods to achieve the surface condition, and 2 methods for painting. It is notable that the partially-ordered solution plans produced in the final domain allowed for multiple linearizations when parts didn't require the same machines. So for instance, the lathing and polishing of two different parts could occur simultaneously if neither part required the machine used to satisfy the other part's goal. The comparative planning times for the scheduling domain are shown in **Figure 6.6**
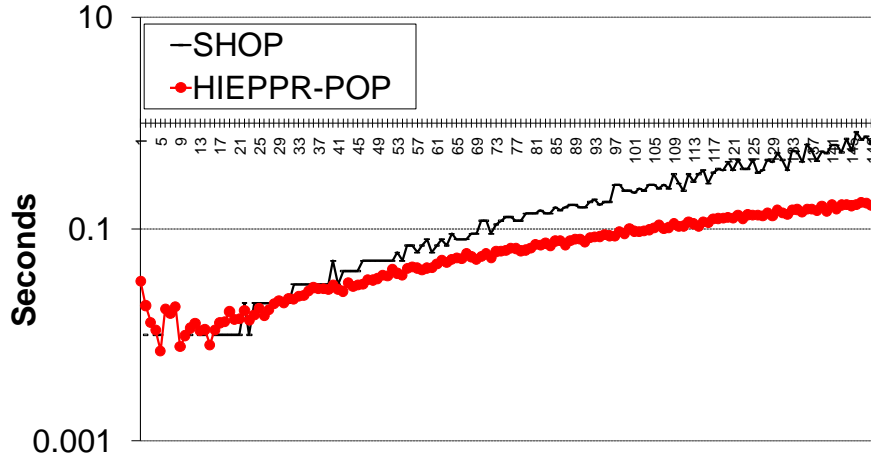
Figure 6.6 Schedule Time Comparison (log scale)

In this domain, the HIEPPR-POP methods frequently had negated preconditions, which take the longest to compute (intuitively, finding that there is no link matching a given criterion means checking that all of them do not match). Nevertheless, good planning times were achieved because subtasks typically added two steps (solving a substantial goal clobbering problem) and 12 links. Most first-principles planners were unable to solve all but the smallest problems in the domain, if any (the domain was later eliminated from the planning competitions). An exception to this is FF, which (impressively) was the only domain-independent planner able to solve beyond problem instance 10. For context, FF worst-case solving time was 27.79 seconds, whereas the worst case of SHOP1 was 0.82 seconds and HIEPPR-POPs worst-case was 0.17 seconds.

Although all these strategies are admittedly simple, the goal with these experiments was to evaluate how the addition of tasks to the DCPOP algorithm (to yield HIEPPR-POP) would affect the size of problems solvable, as well as the speed to solve those problems. In the face of finding out that all problems were unsolvable unless the entire planning process was guided by complete domain-configurable knowledge, I encoded for

each of the three domains a complete collection of methods. That is, by the end of the HTN_Decompose step of the HIEPPR-POP algorithm, a check for the existence of flaws and ordering loops returned null (ie a complete plan). What was completely unanticipated is that the addition of tasks enabled an implementation of the algorithm to achieve comparable CPU performance to that of the well established and most widely used domain-configurable generative planner – SHOP. Once again, this is with the caveat that solutions produced by SHOP1, and typical first principles planners, work hard to keep solution plans short (or of low cost), which imposes an additional computational cost that HIEPPR-POP does not have to pay.

HIEPPR-POP is implemented in Java. I used basic data structures such a two-dimensional reachability matrix to maintain and compute transitive closure of the ordering constraints. The complexity of the update process is on the order of $O(|V||E|)$, where $|V|$ is the number of vertices, which in this case are the steps in the plan, and $|E|$ are the edges, which in this case are the ordering and causal links. The matrix allows for constant time query. I intend to release HIEPPR-POP as free software under the Global Free Licence(sic). This being the first version of HIEPPR-POP, its inputs are purely STRIPS constructs without PDDL extensions such as conditional effects, type information, and quantifiers; therefore I restrict the first evaluation to the simplest benchmark domains from IPC-2000 as subsequent competitions used increasingly complex PDDL extensions.

In each of the graphs, the y-axis is seconds to solve the problem (log scale), and the x-axis is problem instance (the larger x, the larger the planning problem). For the blocks world and schedule domains, HIEPPR-POP performs better than SHOP across most

problems, and appears exponentially faster in blocks-world (caveat same concerns about plan quality). For the logistics domain they have almost the same performance. More interesting than "who-beats-who" is the fact that, unlike what might be expected for generative domain-configurable POP planning, HIEPPR-POP curves have a similar pattern to the ones in SHOP. This shows that generative POP can perform reasonably well, at least in time taken to find a solution, if adequate domain-configurable knowledge is provided.

Currently, the implementation of HIEPPR-POP doesn't support primitive tasks that retract plan refinements (!remove_step, !remove_link, and !remove_order) and only supports primitive tasks that add refinements. Furthermore, lifted planning in HIEPPR-POP is buggy.

6.3 HIEPPR-POP: Generative Planning with Incomplete Knowledge

Having achieved success with HIEPPR-POP on large problems with complete domain-configurable knowledge, it was natural to next evaluate the algorithm on large problems with incomplete knowledge. This would be one step closer to supporting fast plan adaptation (on account of being able to solve problems that were in configurations not anticipated by the domain engineer), as well as supporting "plan sketching" (giving only a few pieces of critical domain-configurable knowledge, and leaving the rest to the underlying first-principles partial-order planner. As hinted at in the beginning of this chapter, this experiment was a resounding failure.

Partial-order planning has a reputation for being slow, though its history is more subtle than that. The following is a paraphrasing of the history of POP planning, as told by

Subbarao Kambhampati in a talk about trying to "revive" partial-order planning (for his planner RePOP): From the early days of planning research (1970) up until 1995, partial-order planning was "king", culminating in the advent of the UCPOP algorithm. If a 6-block problem could be solved, the POP planner was considered very good. In 1995 came the advent of planning as constraint-satisfaction, with the seminal planners Graphplan (Blum & Furst) and SATPLAN (Kautz & Selman). Since 1997, the planning community has focused much of its energies on developing effective domain-independent heuristics (thanks in large part to Bonet & Geffner's notion of a relaxed plan), but these have been almost entirely for state-spaced planners. Jorg Hoffman's FF planner, which operates in the state-space, enjoyed years of success (even to this day) starting in 2000. It seems that Subbarao Kambhampati is one of the few who still believe there is hope for partial-order planning (and he has done much work to show that the state-space heuristics can be adapted and applied to partial-order planning with success).

Another "believer" in partial-order causal link planning is Hakan Younes and Reid Simmons, who were the creators of the VHPOP planner discussed in detail in the related work section (I was originally going to build my approach on top of their planner, which is considered a gold-standard implementation of POCL planners, and, as its name implies, not only supports many heuristics for plan and flaw selection but can interleave their use). Awarded "best newcomer" at the 3rd international planning competition (2002), the success ration of VHPOP was 54% -- far lower than that of FF (85%), LPG (87%), SHOP2 (99%), TALPlanner (100%) and TLPlan (100%). This is surprising, given the extensive optimizations implemented by VHPOP, and thorough evaluation of the most effective heuristic techniques to apply in partial order planning.

144

While HIEPPR-POP with complete knowledge can be very fast, my implementation of first-principles, generative POP planning is not. Even for very small problems, (eg 3 blocks), the search process suffers from poor flaw selection and plan ranking heuristics, leading the planner to explore many hundreds (or thousands, in some cases) of plans. Given that solving even toy problems proved infeasible, it should come as no surprise that HIEPPR-POP without complete knowledge could not make any solutions whatsoever. Currently, the underlying POP planner is only useful for resolving trivial flaws (ie, few resolvers, and no backtracking required) that remain in the plan after applying all expert knowledge. Due in large part to the inefficacy of HIEPPR-POP's underlying POP planner, I humbly join the ranks of the many who have tried this problem before me without success. HIEPPR-POP, in spite of the strengths of the algorithm (as evidenced by the speed of generative planning with complete domain-configurable knowledge, and the ability, under certain conditions, to solve adaptation problems without the traditional retrieval-time/adaptation-time tradeoff) is currently incapable of solving the grand plan adaptation problem as laid out in Section 2.1.

7 Conclusions and Future Work

The main goal of this dissertation was to address the problem of using high-quality, but incomplete, expert planning knowledge in a manner that allows for scalable planning that includes adaptation, and to do so in a way that allows for flexible plan execution. To accomplish this goal I created and studied a new form of expressing and using domain-configurable hierarchical planning knowledge for refining partial-order plans.

Milestone zero was the investigation of using existing plan adaptation algorithms (domain-independent and domain-configurable) in state-of-the-art partial order planners, such as VHPOP [Younes & Simmons, 2003]. This was completed between 2007-8, and resulted in the decision to create a new planning system from scratch. While implementing novel domain-independent adaptation algorithms in existing systems proved feasible, it proved forbiddingly cumbersome to integrate novel domain-configurable approaches. Furthermore the choice to develop a system in-house provided more liberty with respect to future licensing and distribution concerns, and has also been an enlightening pedagogical exercise.

Having decided to develop a new algorithm, the first milestone was the definition of a representation formalism capable of encoding domain-specific partial-order plan adaptation knowledge. The representation formalism had to be flexible enough to be able to represent the various kinds of transformations that may occur in a partial-plan while having clear syntax and semantics. Furthermore, the representation had to allow for rapid

plan reuse. The design of the representation formalism is inherently tied to the underlying partial-order plan generation paradigm on which the plan adaptation is based.

I reached this first milestone in 2008 with the development of the DCPOP-A algorithm, and presented an empirical and theoretical evaluation of it in (Lee-Urban & Munoz-Avila, 2009). This first implementation was non-hierarchical and had no notion of tasks, but demonstrated the feasibility of the approach. The run-time performance of the system was quite slow, but was notable in that it consistently demonstrated near-constant time adaptation in two benchmark domains of arbitrary source plans into solutions to 10 randomly picked problems. Another limitation of this first approach was that it did not scale well; DCPOP-A could only find solutions to small problems.

The second milestone was to investigate ways of extending the formalism to (1) better achieve the goal of rapid plan reuse (rather than slow, but constant time reuse), to (2) make encoding the domain-configurable knowledge easier, and to (3) find a way to make the approach scale to large problems. To that end, I developed the theory in 2008-9 to allow for hierarchical knowledge structures, resulting in HIEPPR-POP. These changes successfully made encoding the knowledge less error-prone, reduced the required number of preconditions and effects of the control rules, decreased planning time considerably (speedups of roughly 1000x over the non-hierarchical approach), and allowed the process to scale to very large problems. Preliminary results indicated that with carefully constructed control rules, HIEPPR-POP was at least competitive with the original version of SHOP and could solve generative problems of equivalent size. However, the status of the new implementation at the time did not yet allow the retraction of plan refinements

(and is therefore only capable of generation, not adaptation). The next step was to finish the retraction-related elements of the implementation.

The third milestone was to undertake a rigorous empirical and theoretical evaluation of HIEPPR-POP, and prepare submissions for the planning and case-based reasoning conferences. This included the analysis of the complexity of the algorithm for performing generation as well as adaptation, the examination of conditions for its soundness and completeness, and the study of how flexible (in execution) solution plans are. I evaluated HIEPPR-POP by testing against standard domains used in the International Planning Competition (IPC).

For generating adaptation problems there were two problem generation schemas. In the first schema I constructed a program, called *Modifier*, which added a parameterized number of features in the initial state and related goals; the selection of features is specialized for each domain. I then: (1) used the random problem generator to generate problems. (2) solved these problems using HIEPPR-POP to obtain plans from scratch. (3) used the modifier program to obtain variants of the problems which were then solved by taking the solution obtained in (2) as the source plan input for HIEPPR-POP plan adaptation. By creating modified partial plans in this fashion, I increased the likelihood that the source plan can be extended to solve the variants of the problems. For the second problem generation schema, I used the same Steps (1)-(3) as in the first schema but the modifier program would be extended to also remove features from the initial and goal states in addition to adding new features to those states (the selection of the features was also domain-specific). This increased the likelihood that the target problem could not be solved by simple extension of the source plan.

The final milestone was to formulate an abstract problem that captured the essence of domain-configurable plan refinement knowledge for partial-order planning, and to perform complexity analysis on this abstract problem.

7.1.1   Scientific Contributions

Throughout this document, several scientific contributions of thisdissertation to the state-of-the-art in planning research have been identified. For convenience, they are summarized and extend below. This work contributes (or is poised to contribute) the following:

- **Ability to make partial-order plans for large problems.** Algorithms that use a partial-order plan representation historically cannot generate plans for problems with many goals, which require many actions to solve the problem. I have shown that in some situations, the HIEPPR-POP algorithm can do so. This is notable in that partial-order solutions can be far more flexible in their execution than totally-order solutions, and also partial-order plans are believed to be a better approach for supporting actions with duration (a condition that exists in many real-world planning problems).

- **Fast, generative domain-configurable partial-order planning.** The main goal of this research was to study scalable (with respect to problem size) and well-founded plan adaptation. However, along the way to achieve this goal the study included generative domain-configurable partial-order planning. This yielded a new planning algorithm that scales well with problem size, outperforming existing domain-independent partial-order generative planning algorithms in terms of time

taken to find a solution (while sacrificing solution quality as measured by plan-length).

- **Well-founded plan adaptation**. HIEPPR-POP has clear semantics specifying the conditions under which soundness (i.e., under which conditions are plans generated guaranteed to be correct) and completeness (i.e., under which conditions are plans guaranteed to be generated when a solvable problem is given) can be guaranteed.

- **The ability to generate adaptation solutions using incomplete adaptation knowledge.** HIEPPR-POP is a domain-configurable adaptation approach that does not require complete adaptation knowledge to generate a solution (because at all times it is refining a partial-order plan, this partial plan can be completed by first-principles whenever there are gaps in the encoded domain-configurable knowledge). This is somewhat novel in modern planning research and eases the knowledge-engineering bottle-neck. A caveat to this contribution is that the ability to extend incomplete solutions generated by HIEPPR-POP into complete solutions (having no flaws) is predicated both on the quality of the first-principles planner used to complete the plans, and the quality of the domain-configurable knowledge used to produce the incomplete solution. In general, even the best implementations of partial-order planning techniques have difficulties finding solutions for even medium-sized problems – therefore, refining incomplete solutions is typically only possible when the first-principles refinements required need no, or little, backtracking.

- **Ability to study the trade-offs between knowledge given and performance gains**. As a result of the previous bullet, it is possible to investigate the trade-offs between the amount of domain-configurable knowledge given and its result on planning performance, measured in running time and percentage of actions retained.

- **A testbed for other researchers.** HIEPPR-POP is the first adaptive, domain-configurable partial-order planner freely available for the Case-Based reasoning and Planning research communities. This enables others to do focused research on other important areas of plan adaptation, such as the problem of which partial plan should be used at the start of the adaptation process (retrieval), the problems of which steps to remove from the plan to adjust and how to adjust the mapping of objects used in the previous solution with objects in the new problem, without having to first build or re-implement a successful plan-adaptation technique.

- **The capability to retain a significant portion of the plan to be adapted.** The HIEPPR-POP approach should be able to retain a significant portion of the plan when feasible. The amount retained will necessarily depend upon the input plan, the conditions of the new problem, and the domain-specific plan adaptation knowledge provided. This capability is a consequence of the ability of the algorithm to solve large problems, and the use of a partial-order plan representation that captures plan commitments at a finer level of granularity than those used in total-order plan representation (a simple sequence of actions).

- **No tradeoff between time spent to find a previous solution to adapt, versus time spent performing adaptation**. Under some (highly constrained) conditions,

the HIEPPR-POP approach (specifically, the DCPOP algorithm) was shown to take roughly the same amount of time to produce a solution to a new problem through adaptation, regardless of the source plan used to make the solution. This is counter to the well-established retrieval-adapt tradeoff commonly considered inescapable in the case-based reasoning literature.

7.1.2 Future Work

There are several ways that the work presented in this dissertation can be expanded. Many of these extensions represent challenging research questions to which there is no clear answer – quite likely another dissertation's worth of investigation. Other expansions are more practically focused, and involve ways in which to modify the algorithm to make it more useful outside of academia.

Nonclassical-planning extensions to HIEPPR-POP. The analysis and evaluation of the HIEPPR-POP algorithm is currently restricted to the assumptions made in the classical planning approach (Section 3.1). This is a relatively simplistic form of the more general problem of solving complex real-world problems. These complex problems involve such difficulties as: (1) limited state observability, where not all facts about the world are known, and therefore cannot be assumed to be false, (2) non-deterministic action outcomes, where the effects of taking an action are not always the same, (3) actions having non-instantaneous effects on the world, for example a drive operator whose effects only become true after a period of time has elapsed (4) dynamic environments, where changes in the state of the world are not restricted to the application of an action – that is, there are other agents besides the planner making changes to the world, or the world itself changes over time (5) the number of possible world states is

infinite, which is the case when the vocabulary of a problem domain is not restricted to a finite set of symbols, such as when a problem includes the use of integers, and (6) not all goals are explicitly specified, but instead the environment has varying "rewards" for taking an action. Many modern planning systems are able to solve problems even in the face of some of these complexities. It is an interesting problem to consider ways in which the HIEPPR-POP algorithm can reason in these domains as well.

Studying HIEPPR-POP in situated, dynamic environments such as games. While the study of dropping the above classical planning assumptions may at first appear to cover this extension of the algorithm, there exist even more complications when a planning algorithm is to be effective in a situated, multi-agent environment. In this instance, the planner is to behave as one actor amongst many in a complex and constantly changing world. In order to do so, it is important that the planning agent be able to: (1) balance reasoning about long term goal achievement with the short-term need to take action quickly, (2) independently change goals in response to new information, (3) compete for limited resources, (4) coordinate with "allied" agents in order to achieve goals not realizable by the planning agent on its own, possibly by sharing a single plan across multiple agents or by decomposing and distributing the problem into individually realizable parts, and (5) form strategies to overcome adversarial agents that intentionally take actions detrimental to the planning agent.

Knowledge acquisition (learning) from solutions. An expert is not always able to explain how they solve problems, in spite of being able to solve them. Furthermore, an expert's time can be limited and expensive, preventing them from having the opportunity or willingness to share their "tricks". What is instead needed is a way of learning the

153

methods an expert intuitively and repeatedly employs. The ability of the HIEPPR-POP algorithm to reason with incompletely specified knowledge is of no or limited help in this case, where high quality solutions (plans) are available, but not the "advice" (methods) about how the solutions were derived. The question of how to learn expert knowledge from previous solutions, while challenging, is therefore an important one. A solution to this question might also be useful in the case where domain-independent, uninformed approaches are able to solve smaller problems in the domain (e.g. blocks world with 5 blocks), but the need exists to solve problems beyond the reach of blind search (e.g. blocks world with 500 blocks).

Another research question regarding knowledge acquisition is whether it is possible to derive HIEPPR-POP methods from existing domain-configurable knowledge structures. For example, expert knowledge may be already be encoded in the form of HTN methods for the SHOP planner, temporal-logic statements for TLPlan, or TMKL models for REM [Murdock, 2001; Murdock and Goel 2003]. Naturally, finding an equivalence between HIEPPR-POP methods and other knowledge structures would help in the use of HIEPPR-POP in those domains for which knowledge already exists.

Use of machine learning techniques to speedup planning and improve solution quality. For each plan element appearing in the precondition list of a HIEPPR-POP method, there are often a number of ways to make a match in the plan being refined. So long as each way to match is attempted systematically, there is no effect on the correctness of the computed plan. My implementation iterates over candidates in the order that they appear in the plan. However, the ordering of candidate evaluation can not only have an effect on the quality of the final plan produced (for instance, a fixed

154

iteration order means the candidate truck to deliver packages within a transportation logistics domain problem is biased to the first truck appearing in the specification of the initial state), it can also effect the time taken to find a solution (because of backtracking over matches that make later preconditions unsatisfiable). Any automatic, unsupervised way to make the order of evaluation of candidates more intelligent is therefore desirable; it is worth investigating machine learning techniques suited to this task.

Similar to the problem of how to order plan element candidate evaluation is the problem of which method to try when multiple methods can match the subtask (which can be seen as a method having multiple precondition-list/subtask-list pairs). Currently, HIEPPR-POP evaluates each pair of precondition-list, subtask-list in a method in the order in which the pairs appear. While in some domains this order of evaluation is necessary to ensure the solution generated matches the intentions of the domain author, there may be situations where a solution can be derived regardless of the order of evaluation of the pairs. When this is the case, the order that pairs are evaluated will certainly effect the amount of time taken to find a solution, and the quality of the resulting plan. It would therefore be worth investigating how machine learning techniques (such as reinforcement learning) might be applied to guide the order of evaluation of pairs.

This dissertation has pointed to strengths, weaknesses, and possible applications of the HIEPPR-POP approach to plan adaptation. It is of interest to note that concurrent with completing this dissertation, I worked with a team to create and deliver to a well-known international company another approach, which they found successful, for reusing

previous solutions to solve new problems (by relaxing the notion of what constitutes a plan). At this time, I am using a modified version of HIEPPR-POP to solve an adaptation problem for a branch of the U.S. government. The problem of how to solve the lack of scalability of plan adaptation techniques remains an open question, and a major barrier to the successes of adaptation techniques in industrial applications. It is my belief that the approach taken in HIEPPR-POP is one promising way to do so, especially as solutions are found to the tasks outlined above as future work.

References

1. Alford, R., Kuter, U., and Nau, D. (2009) Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *Proc. International Joint Conference on Artificial Intelligence* (IJCAI), July 2009.
2. Ambite, J.L. and Knoblock, C.A. (2001) Planning by Rewriting, *Journal of AI Research*.
3. Ambite, J. S., Knoblock, C. A., & Minton, S. (2005) Plan Optimization by Plan Rewriting. In *Intelligent Techniques for Planning*. Ioannis Vlahavas and Dimitris Vrakas Eds., Idea Group Publishing, Hershey, PA.
4. Au, T.C., Muñoz-Avila, H., & Nau, D.S. (2002) On the Complexity of Plan Adaptation by Derivational Analogy in a Universal Classical Planning Framework. In *Proceedings of the Sixth European Conference on Case-Based Reasoning.* Berlin: Springer.
5. Avesani, P., Perini, A., and Ricci, F. (1993) Combining CBR and constraint reasoning in planning forest fire fighting. *Proceedings of the first European workshop on Case-Based reasoning* (ECCBR-93).
6. Ayan, N. F., Kuter, U., Yaman, F., and Goldman, R. (2007) HOTRiDE: Hierarchical Ordered Task Replanning in Dynamic Environments. *Proceedings of the ICAPS-07 Workshop on Planning and Plan Execution for Real-World Systems -- Principles and Practices for Planning in Execution*.
7. Bacchus, F. and Ady, M. (2001) Planning with Resources and Concurrency: A Forward Chaining Approach, *International Joint Conference on Artificial Intelligence (IJCAI-2001)*, AAAI press.
8. Bacchus, F. and Kabanza, F. (2000) Using Temporal Logics to Express Search Control Knowledge for Planning, *Artificial Intelligence.*
9. Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*. Elsevier.
10. Boella, G., and Damiano, R. 2002. A replanning algorithm for a reactive agent architecture. In *Artificial Intelligence: Methodology, Systems, and Applications*, 183–192. Springer Verlag.
11. Botea A., Enzenberger M., Mueller M., and Schaeffer J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*.
12. Carbonell, J.G. (1986) Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. Machine Learning.
13. Choi, D., & Langley, P. 2005. Learning teleoreactive logic programs from problem solving. *Proceedings of the Fifteenth International Conference on Inductive Logic Programming*. Bonn, Germany: Springer.
14. Coles, A.I., Fox, M., Long, D., Smith, A.J., 2008. Teaching forward-chaining planning with JavaFF, colloquium on AI education. In: *Twenty-Third AAAI Conference on Artificial Intelligence*, July.

15. Corchado, J.M., Bajo, J., and Rodríguez, S. (2007) Intelligent Guidance and Suggestions Using Case-Based Planning. *Proceedings of the Seventh International Conference on Case-Based Reasoning (ICCBR-07)*. Springer.

16. Costas, T, & Kashyan, P. (1993) Case-based reasoning and learning in manufacturing with TOTLEC planner. *IEEE Transactions on Systems, Man, and Cybernetics.*

17. Cox, M. T., Muñoz-Avila, H., & Bergmann, R. 2006. Case-based planning. *Knowledge Engineering Review*. 20(3): 283-287.

18. Cunningham, P., Finn, D. and Slattery, S. (1994) Knowledge engineering requirements in derivational analogy. *Proceedings of the European Conference on Case-Based Reasoning (ECCBR-94)*. Springer.

19. Do, M. B. and Kambhampati, S. (2002) Planning Graph-based heuristics for Cost-sensitive Temporal Planning. *Proceedings of the International Conference of AI Planning Systems (AIPS)*, 2002.

20. Erol, K., Hendler, J., and Nau, D. (1994) HTN planning: Complexity and expressivity. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 123-1128). Seattle, WA: AAAI Press.

21. Etzioni, O. (1993) Acquiring search-control knowledge via static analysis. *Artificial Intelligence*.

22. Fagan, M. & Cunningham, P. (2003) Case-Based Plan Recognition in Computer Games. *Proceedings of the Fifth International Conference on Case-Based Reasoning (ICCBR-05)*. Springer.

23. Fern, A., Yoon, S.W., and Givan, R. (2004) Learning Domain-Specific Control Knowledge from Random Walks. *Proceedings of the International Conference on Automated Planning & Scheduling (ICAPS-04)*. AAAI Press.

24. Fikes, R. and Nilsson, N. (1971) STRIPS: A new approach to the application of theorem proving to problem solving. Artif. Intell., 2(3/4):189-208, 1971.

25. Fox, M., Gerevini, A., Long, D., and Serina, I. (2006) Plan Stability: Replanning versus Plan Repair, *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*. AAAI Press.

26. Gerevini, A, & Serenia, I. (2000). Fast Plan Adaptation through Planning Graphs: Local and systematic search techniques. Proceedings of the *Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*. Breckenridge, CO: AAAI Press.

27. Gerevini, A. and Serina, I. (2009) Efficient Plan Adaptation through Replanning Windows and Heuristic Goals. *Journal of Algorithms in Cognition, Informatics and Logic*, (Elsevier, ISSN: 0196-6774), to appear.

28. Ghallab, M., Nau, D., and Traversa, P. 2004. Automated Planning: Theory and Practice. Morgan Kaufmann Publishers.

29. Greene, D., Freyne, J., Smyth, B., and Cunningham, P. 2008. An Analysis of Research Themes in the CBR Conference Literature. In Proceedings of the 9th European Conference on Case-Based Reasoning (ECCBR 2008), Trier, Germany. Springer Verlag.

30. Haigh, K. Z., Shewchuk, J. R., & Veloso, M. M. (1997). Exploiting Domain Geometry in Analogical Route Planning, Journal of Experimental and Theoretical Artificial Intelligence. 9: 509-541

31. Hammond, K. (1986) Chef: A model of case-based planning. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI-86)*. AAAI Press, 1986.
32. Hanks, S. and Weld, D. 1995. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligece Research*, 2.
33. Hart, P.E., Nilsson, N.J., Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, IEEE press.
34. Hoffmann, J., and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 253-302.
35. Hogg, C. & Munoz-Avila, H. (2007) Learning of Tasks Models for HTN Planning. *Proceedings of the ICAPS-07 Workshop on AI Planning and Learning (AIPL)*. AAAI Press.
36. Ihrig, L.H., & Kambhampati, S. (1994). Derivational replay for partial order planning. Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94). Seattle, WA: AAAI Press, 1994.
37. Ihrig, L., Kambhampati, S.: Design and Implementation of a Replay Framework Based on a Partial Order Planner. In: AAAI/IAAI-96, pp. 849-854. AAAI Press (1996)
38. Ihrig, L. & Kambhampati, R. (1997) Storing and Indexing plan derivations through explanation-based analysis of retrieval failures L. and S. *Journal of Artificial Intelligence Research*.
39. Jin, L., Decker, K., & Schmidt, C. (2009) BioPlanner: A Plan Adaptation Approach for the Discovery of Biological Pathways across Species. *Proceedings of the Innovative Applications of AI (IAAI-09)*. AAAI Press.
40. Kambhampati, S. (1994). Exploiting causal structure to control retrieval and refitting during plan reuse. *Computational Intelligence*.
41. Kambhampati, S. 1997. Refinement Planning as a Unifying Framework for Plan Synthesis. AI Magazine 18(2): 67–97.
42. Katukam, S., and Kambhampati, S. (1994) Learning Explanation-based search control rules for Partial-order planning. *Proceedings of the conference for the American Association for Artificial Intelligence (AAAI-94)*. AAAI Press.
43. Knoblock, C.: Generating Parallel Execution Plans with a Partial-Order Planner. In: AIPS-94, pp. 98-103. AAAI Press (1994)
44. Kuchibatla, V., and Munoz-Avila, H. (2006) An Analysis on Transformational Analogy: General Framework and Complexity. In *Proceedings of European Conference in Case-based reasoning* (ECCBR-06). Springer.
45. Kuter, U., and Nau, D. 2005. Using Domain-Configurable Search Control in Probabilistic Planners. *Proceedings of the National Conference on Artificial Intelligence (AAAI-05)*. Springer.
46. Kvarnström, J., and Doherty, P. (2001) TALplanner: A Temporal Logic Based Forward Chaining Planner. Annals of Mathematics and Artificial Intelligence (AMAI), Volume 30, pages 119-169.
47. Kvarnström, J., Doherty, P., and Haslum, P. Extending TALplanner with concurrency and resources. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000),* 2000.

48. Kvarnström, J. and Magnusson, M. (2003) TALplanner in the third International Planning Competition: Extensions and Control Rules. *Journal of Artificial Intelligence Research*, 20:343–377

49. Lee-Urban, S., and Munoz-Avila H. (2009) Adaptation Versus Retrieval Trade-Off Revisited: an Analysis on Boundary Conditions. *Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR-09)*. Springer.

50. Lopez de Mántaras, R., McSherry, D., Bridge, D., Leake, D., Smyth, B., Craw, S., Faltings, B., Maher, M. L., Cox, M. T., Forbus, K., Keane, M., Aamodt, A., & Watson, I. (2006). Retrieval, reuse and retention in case-based reasoning. *Knowledge Engineering Review*.

51. Marthi, B., Russell, S. J., and Wolfe, J. (2008). Angelic Hierarchical Planning: Optimal and Online Algorithms. *Proceedings of the International Conference in AI Planning and Scheduling (ICAPS-08)*. AAAI Press.

52. Martin, M. and Geffner, H. (2000) Learning generalized policies in planning using concept languages. *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR 2000)*. Morgan Kaufmann.

53. McAllester, D. A., & Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pp. 634–639, Anaheim, CA. AAAI Press.

54. Miksch, Silvia. (1999) Plan management in the medical domain. *AI Communications* 12, pp 209-235.

55. Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Kluwer Academic Publishers, Boston, MA.

56. Mitchell, S.W. (1997). A hybrid architecture for real-time mixed-initiative planning and control. *Proceedings of the Ninth Conference on Innovative Applications of AI (IAAI)*. AAAI Press.

57. Mitchell, T.; Keller, R.; and Kedar-Cabelli, S. (1986) Explanation-based generalization: A unifying view. *Machine Learning*.

58. Mooney, R.J. (1988) Generalizing the Order of Operators in Macro-Operators. *Machine Learning*.

59. Munoz-Avila, H., Aha, D.W., Breslow, L.A., & Nau, D. (1999). HICAP: An Interactive Case-Based Planning Architecture and its Application to Noncombatant Evacuation Operations. In: *Proceedings of the Innovative Applications of AI (IAAI-99)*. AAAI Press.

60. Munoz-Avila, H. and Cox, M.T. (2008) Case-Based Plan Adaptation: An Analysis and Review. IEEE Intelligent Systems. IEEE inc.

61. Muñoz-Avila, H., Hüllen, J.: Feature Weighting by Explaining Case-Based Planning Episodes. In: EWCBR-96. LNCS, vol. 1168, pp. 280-294. Springer (1996)

62. Muñoz-Avila, H & Weberskirch, F. (1996) Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. *Proceedings of the International Conference on AI Planning Systems (AIPS)*. Edinburgh: AAAI Press.

63. Muñoz-Avila, H. & Weberskirch F. (1997) A Case Study on the Mergeability of cases with a Partial-Order Planner. In S. Steel & R. Alami (Eds.). *Proceedings of the European Conference on Case-based Reasoning (ECP)*, Springer.

64. Murdock, J.W. (2001). Self-Improvement Through Self-Understanding: Model-Based Reflection for Agent Adaptation. PhD thesis, Georgia Institute of Technology.

65. Murdock, J.W, and Goel, A. K. (2003) Localizing planning with functional process models. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*.

66. Myers, K. L. (2006) Metatheoretic Plan Summarization and Comparison. *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*. AAAI Press.

67. Nau. D. S. (2007) Current trends in automated planning. *AI Magazine*.

68. Nau, D., Au, T., Ilghami, O., Kuter, U., Muñoz-Avila, H., Murdock, J., Wu, D., Yaman, F.: Applications of SHOP and SHOP2. IEEE Intelligent Systems 20(2), 34-41. (2005)

69. Nau, D., Cao, Y., Lotem, A., and Muñoz-Avila, H. (1999) SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. AAAI Press.

70. Nebel, B. and J. Koehler, (1995) Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis, *Artificial Intelligence* (Special Issue on Planning and Scheduling).

71. Nguyen, X., & Kambhampati, S. (2001). Reviving partial order planning. In Nebel, B. (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pp. 459–464, Seattle, WA. Morgan Kaufmann Publishers.

72. Onder, N., Whelan G. C., and Li L. (2006) Engineering a Conformant Probabilistic Planner. *Journal of AI research (JAIR)*.

73. Ontañón, S., Mishra, K., Sugandh, N., Ram A. (2007) Case-Based Planning and Execution for Real-Time Strategy Games. *Proceedings of the Seventh International Conference on Case-Based Reasoning (ICCBR-07)*. Springer.

74. Paulokat, J., Wess, S.: Planning for Machining Workpieces with a Partial-Order, Nonlinear Planner. In: Proceedings of the AAAI 1994 Fall Symposium on Planning and Learning. AAAI Press (1994)

75. Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B., Rich, C., & Swartout, W. (Eds.), Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, pp. 103–114, Cambridge, MA. Morgan Kaufmann Publishers.Perez, A., & Carbonell, J. (1994). Control knowledge to improve plan quality. Proceedings of the International Conference on AI Planning Systems (AIPS-96). (AIPS-94). Chicago: AAAI Press.

76. Pollack, M. E., Joslin, D., and Paolucci, M. (1997) Flaw selection strategies for partial-order planning. *Journal of AI Res*earch.

77. Ram, A., and Francis, A. (1996) Multi-Plan Retrieval and Adaptation in an Experience-Based Agent. In *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, D.B. Leake, editor, AAAI Press.

78. Reddy, C., and Tadepalli, P. (1997) Learning goal decomposition rules using exercises. *Proceedings of the International Conference on Machine Learning (ICML)*. ACM.

79. Ruby, D., and Kibler, D. F. (1991). SteppingStone: An Empirical and Analytic Evaluation. *Proceedings of the Ninth National Conference on Artificial Intelligence*, 527--531, Morgan Kaufmann.

80. Sacerdoti, E. D. 1975. The Nonlinear Nature of Plans. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75), 206-214.

81. Salem, A.-B. M., Nagaty, K. A., & El Bagoury, B. (2003). A Hybrid Case-Based Adaptation Model for Thyroid Cancer Diagnosis. In ICEIS 2003, *Proceedings of the 5th International Conference on Enterprise Information Systems*.

82. Sanchez Ruiz-Granados, A., Lee-Urban, S. & Munoz-Avila, H., Gonzalez Calero, P. A., Diaz Agudo, B. (2007) Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. *Proceedings of the ICAPS-07 Workshop on ICAPS 2007 Workshop on Planning in Games*. AAAI Press.

83. Schmidt, R., Vorobieva, O., & Gierl, L. (2001). Case-based Adaptation Problems in Medicine. *International Journal of Medical Informatics*.

84. Schmidt, R., Vorobieva, O., & Gierl, L. (2003). Adaptation Methods in an Endocrine Therapy Support System. Proceedings of *ICCBR-03 Workshop on CBR in the Health Sciences*.

85. Tonidandel, F. and Rillo, M. (2002) The FAR-OFF system: A heuristic search case-based planning. *Proceedings of the International Conference on AI Planning Systems (AIPS)*, AAAI Press, 2002

86. Tonidandel, F. and Rillo, M. (2005) Case Adaptation by Segment Replanning for Case-Based Planning Systems. *Proceedings of the Seventh International Conference on Case-Based Reasoning (ICCBR-07)*. Springer.

87. US Department of Education. (2009) *Practical Information on Crisis Planning*. http://www.higheredcenter.org/resources/practical-information-crisis-planning-guide-schools-and-communities (last viewed: November, 2009).

88. van der Krogt, R.P.J. and de Weerdt, M.M.. (2005) Plan Repair as an Extension of Planning. *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*. AAAI Press.

89. Veerakamolmal, P. and Gupta, S. M., (2002) A Case-Based Reasoning Approach for Automating Disassembly Process Planning, *Journal of Intelligent Manufacturing*.

90. Veloso, M. (1994) *Planning and learning by analogical reasoning*. Berlin: Springer-Verlag.

91. Veloso, M. & Carbonell, J. 1993. Derivational Analogy in PRODIGY: Automating Case Acquisition, Storage, and Utilization. *Machine Learning*, 10(3):249-278.

92. Veloso, M. M., Mulvehill, A. M., & Cox, M. T. (1997). Rationale-supported mixed-initiative case-based planning. In Proceedings of the *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*. AAAI Press / The MIT Press.

93. Vidal, V. and Geffner, H. (2006) Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Artificial Intelligence*.

94. Warfield, I., Hogg, C., Lee-Urban, S., Munoz-Avila, H. (2007) Adaptation of Hierarchical Task Network Plans. Proceedings of the *Twentieth Flairs International Conference (FLAIRS-07)*. AAAI Press.

95. Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine, 15* (4), 27–61.

96. Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc.

97. Wilkins, D. E., & desJardins, M. (2001) A Call for Knowledge-Based Planning. AI Magazine. AAAI Press.

98. Williamson, M., & Hanks, S. (1996). Flaw selection strategies for value-directed planning. In Drabble, B. (Ed.), Proceedings of the Third International Conference on Artificial Intelligence in Planning Systems, pp 237-244, Edinburgh, Scotland. AAAI Press.

99. Intelligence Planning Systems, pp. 237–244, Edinburgh, Scotland. AAAI Press.

100.   Winner, E., and Veloso, M. M. (2003). DISTILL: Learning domain-specific planners by example. *In Proceedings of the International Conference on Machine Learning (ICML-03).*

101.   Wolldridge, M. 2002. An Introduction to MultiAgent Systems. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England.

102.   Younes, H.L.S. and Simmons, R.G. (2003) VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20, pp. 405-430.

103.   Younes, H.L.S. (2006). The VHPOP Story. *Tempastic.ORG*. Retrieved March 2012, from http://www.tempastic.org/vhpop/vhpopstory.html

Appendix D: Glossary of definitions

Definition 3-1. A **constant** symbol is syntactically composed from a unique sequence of one or more alpha-numeric characters, which are, by convention, the upper and lowercase versions of the 26 letters in the English alphabet, the digit characters '0' through '9', the dash character'-' and the underscore character '_'), for example 'table' or 'truck-1' (the single quotes are not part of the symbol); a constant is therefore a string semantically used to refer to a specific object in the problem being modeled.

Definition 3-2. A **variable** is a symbol that can be used in the place of a constant, akin to the concept of variables in algebra. As is common in the automated planning field, I follow the syntactic convention that variables begin with a question mark, followed by a sequence of one or more characters. Two examples of variables are '?x' and '?truck' (again, the single quotes are not part of the symbol).

Definition 3-3. A **term** is either a variable or a constant.

Definition 3-4. A particular variable is referred to as **bound** (past tense of 'bind') if and only if there is an assignment of the variable to a term. For example, if the variable '?x' were to be assigned to the constant 'table', then one would say that ?x is bound to table. When a variable is bound to a constant, the variable is said to be **grounded** (a reference to the Earth's surface, I believe).

Definition 3-5. A **predicate name**, or **predicate symbol**, is a character sequence having the same syntax as a constant. However, rather than referring to an *object* in

the modeled world, a predicate name is used to semantically refer to a *relation* in that world. Some examples of predicate names are 'on', 'at', and 'in-city'. Predicate symbols also have an **arity**, which indicates the number and names of terms taken as arguments by that predicate. For example on ?x a has a predicate symbol named 'on' of arity 2, the first term is a variable named ?x and the second is a constant symbol a.

Definition 3-6. An **atomic formula**, or **atom**, is a statement of fact about the modeled world. It is syntactically formed by an opening parenthesis symbol '(' followed by a predicate symbol, followed by a space separated list of terms equal in number to the predicate's arity, followed by a closing parenthesis symbol ')'. The space separated list of terms is referred to as **arguments** or **parameters**. If all arguments are grounded, then the atom is also grounded. The following is a comma separated list of example atoms – the comma is not part of the syntax: (on ?x table), (in package1 truck1), (in-city Lehigh Bethlehem). Each of the predicate symbols on, in, in-city have an arity two. The only argument that is a variable is ?x, the rest are constant symbols.

Definition 3-7. A **substitution** is a collection of variable bindings. When a substitution is **applied** to an atomic formula *a*, a new atom is created by replacing each of the variables in *a* with the term to which it is mapped (if such a mapping exists in the substitution).

Definition 3-8. A **world state** is a finite set of grounded atoms. Semantically, a world state is an assertion about all facts that are true. By convention, any atom not

171

173

Vita

Stephen Montgomery Lee-Urban was born in Long Island, NY, USA on October 7, 1981 to Kathryn and James Urban; his immersion in computers can be traced to Kathy's husband of over twenty years, Alfred Lee. Steve attributes much of his achievements to having three loving and supportive parents.

He attained his Bachelor of Science in Computer Engineering from Lehigh University in 2003, and his Master's in Computer Science from the same university in 2005, graduating with highest honors for both degrees. For a total of two years between 2001 and 2004, Steve worked at AirClic Inc. as a software developer, where he transformed from an academic and hobbyist programmer into an industrial-grade designer and implementer.

Between 2003 and 2004, he was a teaching assistant for 7 courses, ranging from sophomore level programming, senior level computer networking and network programming, introduction to artificial intelligence, to the design of computer games. From 2004 to 2011, he worked as a research assistant in the InSyTe lab under Dr. Muñoz-Avila, researching and publishing on automated planning, reinforcement learning, hierarchical task network planning, case-based reasoning, and the use of artificial intelligence techniques in commercial computer games. During that time, he served for four years on the executive board as the communications officer of Lehigh's graduate student senate, and received in 2008 the Graduate Student Life Leadership award for exemplary scholarship, leadership, and service to the Lehigh graduate student

community. That same year, he was inducted into the Rossin Doctoral Fellows Program of Lehigh's engineering college, which is for high potential Ph.D. candidates. In 2010, he was the recipient of a Dean's Teaching Assistantship, which is awarded to one advanced Ph.D. student per department who demonstrates an affinity for teaching, and the ability to provide high-quality assistance in the classroom. That year, he was also the instructor of record for a four-credit, non-elective Introduction to Computer Science course.

Stephen took one-year off at the end of his doctoral program, in 2011, to be a consultant as technical lead on a project for Walt Disney Imagineering (through Dr. Ashwin Ram). While on that project he met his current boss, Dr. Mark Riedl, for whom he began working in fall of 2011 as a research scientist at The Georgia Institute of Technology, investigating approaches to narrative computing.

Publications

Conference Papers:

- Zook, A., Lee-Urban, S., Riedl, M., Holden, H., Sottilare, R., and Brawner, K. (2012) Automated Scenario Generation: Toward Tailored and Optimized Military Training in Virtual Environments. Proceedings of the 7th International Conference on the Foundations of Digital Games, Raleigh, North Carolina, 2012.

- Gillespie, K., Karneeb, J., Lee-Urban, S., and Munoz-Avila, H. (2010) Imitating Inscrutable Enemies: Learning from Stochastic Policy Observation, Retrieval and Reuse. Proceedings of the 18th International Conference on Case Based Reasoning (ICCBR 2010). Springer.

- Lee-Urban, S., Munoz-Avila, H. (2009) Adaptation Versus Retrieval Trade-Off Revisited: an Analysis of Boundary Conditions. In Proceedings of the 8th International Conference on Case-Based Reasoning (ICCBR-09). Springer.

- Auslander, B., Lee-Urban, S., Hogg, C., and Munoz-Avila, H. (2008) Recognizing The Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning. In Proceedings of the 9th European Conference on Case-Based Reasoning (ECCBR-08). Springer.

- Vasta, M., Lee-Urban S. & Muñoz-Avila, H. (2007) RETALIATE: Learning Winning Policies in First-Person Shooter Games. In Proceedings of the Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI-07). AAAI Press.

- Warfield, I., Hogg, C., Lee-Urban, S., Muñoz-Avila, H. (2007) Adaptation of Hierarchical Task Network Plans. In Proceedings of the Twentieth Flairs International Conference (FLAIRS-07). AAAI Press.

- Lee-Urban, S. Muñoz-Avila, H. (2006) A study of Process Languages for Planning Tasks. In Proceedings of the sixteenth International Conference on AI Planning and Scheduling (ICAPS-06) Doctoral Consortium.

- Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. (2005) Hierarchical Plan Representations for Encoding Strategic Game AI. In Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05). AAAI Press.

Book Chapters:

- Hogg, C., Lee-Urban, S., Muñoz-Avila, H., Auslander, B., Smith, M. Game AI for Domination Games. In Pedro Gonzales Calero (Ed.) Artificial Intelligence for Computer Games. Springer Verlag, 2011.

- Lee-Urban, S., Smith, M. & Munoz-Avila, H. 2008. Learning Winning Policies in Team-Based First-Person Shooter Games. AI Game Programing Wisdom 4. Charles River Media.

Theses:

- Lee-Urban, S. Hierarchical Planning Knowledge for Refining Partial-Order Plans. Doctoral Thesis, 2012.

- Lee-Urban, S. TMK Models to HTNs: Translating Process Models into Hierarchical Task Networks. Master's thesis, 2005.

Workshop Papers:

- Zook, A., Lee-Urban, S., Drinkwater, M., and Riedl, M. (2012) Skill-based Mission Generation: A Data-driven Temporal Player Modeling Approach. In Proceedings of the 3rd Workshop on Procedural Content Generation in Games, Raleigh, North Carolina, 2012.

- Li, B., Lee-Urban, S., Appling, D.S., and Riedl, M. (2012) Automatically Learning to Tell Stories about Social Situations from the Crowd. In Proceedings of the LREC 2012 Workshop on Computational Models of Narrative, Istanbul, Turkey, 2012.

- Hogg, C., Lee-Urban, S., Auslander, B., and Munoz-Avila, H. (2008) Discovering Feature Weights for Feature-Based Indexing of Q-Tables. In

Proceedings of the Uncertainty and Knowledge Discovery in CBR Workshop at the 9th European Conference on Case-Based Reasoning (ECCBR-08).

- Sanchez-Ruiz, A., Lee-Urban, S., Muñoz-Avila, H., Diaz-Agude, B., & Gonzalez-Calero, P. (2007) Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval. In Proceedings of the workshop on Planning in Games at the International Conference on Automated Planning and Scheduling (ICAPS-07).

- Lee-Urban, S., Parker, A., Kuter, U., Muñoz-Avila, H., & Nau, D. (2007) Transfer Learning of Hierarchical Task-Network Planning Methods in a Real-Time Strategy Game. In Proceedings of the AI Planning and Learning Workshop (AIPL) at the International Conference on Automated Planning and Scheduling (ICAPS-07).

- Ponsen, M., Lee-Urban, S., Muñoz-Avila, H., Aha, D., and Molineaux, M. (2005) Stratagus: An Open-Source Game Engine for Research in Real-Time Strategy Games. Workshop for International Joint Conference on Artificial Intelligence (IJCAI-05).