Effective Search Techniques for Non-Classical Planning via Reformulation

by

Jorge A. Baier

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Effective Search Techniques for Non-Classical Planning via Reformulation

Jorge A. Baier
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

Automated planning is a branch of AI that addresses the problem of generating a course of action to achieve a specified objective, given an initial state of the world. It is an area that is central to the development of intelligent agents and autonomous robots. In the last decade, automated planning has seen significant progress in terms of scalability, much of it achieved by the development of heuristic search approaches. Many of these advances, are only immediately applicable to so-called *classical planning* tasks. However, there are compelling applications of planning that are non-classical. An example is the problem of *web service composition*, in which the objective is to automatically compose web artifacts to achieve the objective of a human user. In doing so, not only the hard goals but also the *preferences* of the user—which are usually not considered in the classical model—must be considered.

In this thesis we show that many of the most successful advances in classical planning can be leveraged for solving compelling *non-classical* problems. In particular, we focus on the following non-classical planning problems: planning with temporally extended goals; planning with rich, temporally extended preferences; planning with procedural control, and planning with procedural programs that can sense the environment. We show that to efficiently solve these problems we can use a common approach: *reformulation*. For each of these planning tasks, we propose a reformulation algorithm that generates another, arguably simpler instance. Then, if necessary, we adapt existing techniques to make the reformulated instance solvable efficiently. In particular, we show that both the problems of planning with temporally extended goals and with procedural control can be mapped into classical planning. Planning with rich user preferences, even after reformulation, cannot be mapped into classical planning and thus we develop specialized heuristics, based on existing heuristics, together with a branch-and-bound algorithm. Finally, for the problem of planning with programs that sense, we show that under certain conditions programs can be reduced to simple operators, enabling the use of a variety of existing

planners. In all cases, we show experimentally that the reformulated problems can be solved effectively by either existing planners or by our adapted planners, outperforming previous approaches.

Dedication

To Daniela, Valentina, and Benjamin.

Acknowledgements

This work would have not been possible without my supervisor Sheila McIlraith. Sheila, with her knowledgeable and impressively broad view of AI, provided me with a constant stream of ideas, inspiration, and mental stimulation. I feel very grateful for her extremely dedicated support to my work, and for the significant effort she put on maximizing the span of my research and my visibility as a member of the AI community. There are many more reasons beyond academia to thank Sheila. Her support usually exceeded the limits of the academic life and she constantly transmitted me optimism and good spirit. Without any doubt, working with her was a fantastic experience.

I thank also the other members of my thesis committee: Fahiem Bacchus, Hector Levesque, and Subbarao (Rao) Kambhampati. Fahiem enriched significantly my formation as a researcher as we had the chance to work together in some of the work presented in this thesis. I owe him much of the success we obtained in the 2006 International Planning Competition, for which he collaborated not only with ideas and experience but with several lines of C code. Hector provided a tremendous amount of feedback and help on parts of the work included in this thesis. Rao provided important feedback, helping to present my work in a broader manner.

I thank my colleague friend Christian Fritz, with whom I coauthored a couple of papers during my studies. Working with Christian was a great pleasure. I also thank him for sharing plenty of discussions on other topics related to my research.

During my studies I had the opportunity to interact with a number students and professors from other institutions. I want to give a special thanks to Héctor Geffner, who was my mentor in ICAPS-06 and provided me with questions that I still cannot answer well, and that inspired some of my thesis and current research.

I thank my friends from the KR group: Shirin Sohrabi, Christian Fritz, Eric Hsu, Christian Muise, Toby Hu, Letao Wang, Horst Samulowitz, and Jessica Davis. And also thank my friends Marcelo Arenas, Pablo Barceló, Arnold Binas, Andrés Lagar Cavilla, Claudia García, Jocelyn Simmonds, Sebastián Sardiña, Leo Triviño, Daisy Guerrero (and someone else that I just forgot). Without their support this experience would have been a lot harder. A hug to each one of them!

I thank the unconditional support from my parents and in-laws, who made my stay in Toronto a lot easier, and thus contributed indirectly to the success in my studies.

Last, but far from least, I thank my wife Daniela, for her love, patience, and sacrifice. I thank her for her continuous support even when things where not easy. Thanks to my daughter Valentina, for filling me with love by the end of my studies. Finally, thanks to my son Benjamin, for making me so happy, and for not being born the day of my defense ;). I love them so much!

Contents

Chapter 1

Introduction

Automated planning is a fundamental reasoning task for autonomous agents. Traditionally, it corresponds to the problem of generating a course of action to achieve a specified goal state, given an initial state of the world, and a description of the dynamics of the world. Different variants of planning—more formally known as *paradigms*—are created by making different assumptions about the dynamics of the world and about the knowledge and sensing capabilities of the agent. The simplest of these paradigms is *classical planning*.

Classical planning has seen a great deal of advancement in the last few years, most of it due to the development of domain-independent heuristics. Nevertheless, many compelling applications of automated planning do not fall under the classical paradigm. These compelling applications may consider characterizations of the planning objective in terms of a rich goal and preference representation. Moreover, as opposed to classical planning, the building blocks for plans might not be simple primitive actions but rather relatively complex procedures. Unfortunately, as we move beyond the classical planning paradigm, many of the highly optimized planning techniques that have led to this recent success are no longer immediately applicable.

In this thesis, we investigate how recent advances in the automated planning community can be leveraged to solve some compelling non-classical planning tasks. In the rest of this chapter, we describe classical planning at an intuitive level. We continue by describing a in which classical planning does not offer a satisfactory solution. The chapter finishes by describing in some detail the specific contributions of this thesis, and outlining the remainder of this document.

1.1 Recent Advances in Classical Planning

In classical planning it is assumed that the domain is deterministic. This means that an action transforms a state into a single successor state. Moreover, in classical planning we assume that the initial state is

a complete description of the world, i.e., includes all facts that hold true. This implies that the agent knows everything that holds true as a result of performing any sequence of actions. The objective of the problem is to find a plan that satisfies the goal. This goal is a condition that must be satisfied at the state that is reached after performing the plan in the initial state (i.e., the final state).

Even though classical planning makes quite stringent assumptions, it is a computationally hard problem. Indeed, deciding the existence of a plan is PSPACE-complete (Bylander, 1994), which means that it is very unlikely that there exists a polynomial-time algorithm to solve it. In most cases, however, automated planners are expected to carry out tasks that can be solved by humans with relatively low difficulty (e.g., "deliver mail to the professors' offices", "organize a travel using a collection of web services", "create a plan to transport packages to their destinations using trucks"). When non-optimal solutions are required, these problems do not have a combinatorial nature and, indeed, for many of them it is possible to construct solvers that run in polynomial time (Helmert, 2003, 2006b).

Although automated planning has been a topic of research for three decades, for a number of years planners could not scale well in domains that humans can easily solve. Only in the last decade has the planning community produced planning technology that can scale relatively well in many of the aforementioned "easy problems".

Most of the recent success in *satisficing* (i.e., non-optimal) planning can be attributed to the development of domain-independent heuristics (e.g. McDermott, 1996; Bonet and Geffner, 2001). These heuristics are functions that estimate the cost of solving the planning problem given the current state. For example, Bonet and Geffner (2001) propose a heuristic $h(\cdot)$, such that $h(s)$ estimates the cost of achieving the goal from s by solving a relaxed version of the problem, in which the negative effects of the actions are ignored. This relaxation is usually referred to as the *delete relaxation*. Given the heuristic $h(s)$ it is possible to use standard algorithms (e.g., best-first search) to solve the planning problem. Most recent planners, however, use their own specific search algorithms, which provide a better tradeoff between the computation time required to compute the heuristic and the nodes that are expanded by the planner.

Most of the top-performers in recent occurrences of the International Planning Competition (IPC) use some sort of heuristic in the search for a plan (see Figure 1.1). Many of them rely, at least in part, on computing a solution to the delete relaxation. The FF planner is probably one of the most influential of the recent planners: a subset of the techniques it introduced have been used by most of the subsequent IPC winners (satisficing track). Many other planners not shown in the figure also use techniques introduced by FF.

FF (Hoffmann and Nebel, 2001) It uses domain-independent heuristics to guide search. The heuristic is computed by finding a plan for the delete relaxation using a modification of the Graphplan (Blum and Furst, 1997) algorithm. Uses enforced hill-climbing for search, a modification of hill-climbing. Winner of IPC, satisficing track, in 2000.

LPG (Gerevini, Saetti, and Serina, 2003) Was inspired by local-search SAT solvers (e.g., walksat), this planner uses local search in the space of plans. Winner of IPC, satisficing track, in 2002.

FAST-DOWNWARD (Helmert, 2006a) It uses a domain-independent heuristic that is computed by solving relaxed causal-graph representation of the planning problem. In addition, it uses the FF heuristic. Winner of IPC, satisficing track, in 2004.

SGPlan$_5$ (Hsu, Wah, Huang, and Chen, 2007) It uses an optimization approach to partition the planning problem into different subproblems. It obtains solutions for the subproblems using the a modification of the heuristic planner Metric-FF (Hoffmann, 2003). Winner of IPC, satisficing track, in 2006.

LAMA (Richter, Helmert, and Westphal, 2008) In a preprocessing phase, it computes a set of *landmarks*, which are sub-goals that have to be achieved before achieving any goal. For planning, it uses a pseudo-heuristic designed to exploit the landmarks. In addition, it uses the FF heuristic for planning. Winner of IPC, satisficing track, in 2008.

Figure 1.1: Brief description of the winners of the last 5 International Planning Competitions (IPC) in the non-optimizing (satisficing) track.

1.2 Classical Planning Is Not Enough: An Example

As noted above, a planning task must satisfy a number of restrictions in order to be classical. As a result, many compelling applications cannot be represented within the classical planning paradigm. One example is *component software composition*, in which it is necessary to create new software by re-using existing components. An instance of this problem is Web Service Composition (WSC).

WSC involves the automatic composition of web services to perform some task, given a high-level description of the tasks objective (McIlraith, Son, and Zeng, 2001). On the other hand, a Web Service is a piece of software that is available via the Web, and whose features, and (possibly) its functionality are described using a formal language. As an example, consider the task "make travel arrangements to attend the IJCAI-09 conference." This task necessitates selecting and executing a variety of web services to perform tasks such as booking accommodation, purchasing air tickets, and possibly arranging other types of transportation.

WSC is a compelling problem: both the academic community and industry have shown considerable interest in the problem. As defined above, WSC is clearly a planning problem. Nevertheless, researchers have pointed out many aspects of WSC that do not fit into the classical paradigm (e.g. Hendler, 1999; McIlraith *et al.*, 2001; McIlraith and Son, 2002; Srivastava and Koehler, 2003; Sohrabi, Prokoshyna, and McIlraith, 2006). We enumerate some of them below.

Characteristic 1 The planning problem cannot be expected to take place at the level of primitive actions. Rather, *complex actions*—which in WSC correspond to the web services themselves—are the building blocks to construct the required plans.

Characteristic 2 Unlike the classical model, in WSC there is incomplete information about the initial state. While this implies that we do not have complete information at planning time, it may also be true that we do not have complete information at execution time.

Characteristic 3 Web services have *inputs* and *outputs* as well as preconditions and effects. Inputs and outputs are not easy to describe in the classical paradigm. They can be represented with relative ease if we allow the agent to reasoning about knowledge. For example, at planning time, we could express that an agent *knows* that it have received an output from a service and therefore knows, say, the price of the flight, although we still do not have such a value.

Characteristic 4 Web services can be viewed as entities that both *sense* and *alter* the environment. As an example, consider the purchase of an air ticket using a web service. We sense the environment by acquiring new information about different alternative flights between the origin and the destination (e.g., prices, departure times, etc.). On the other hand, the execution of the service actually changes some properties of the world (e.g., a booking is registered in the airline database, a credit card is charged).

Characteristic 5 Compositions (i.e., plans), rather than simple sequences of actions, may need to contain complex control structures involving loops, non-determinism and choice. This means that the solution to a WSC task may look more similar to an imperative program rather than to a sequence of actions.

Characteristic 6 WSC is usually required to achieve *rich goals* in the presence of *rich user preferences*. As such, goals might not only refer to the final state, but also to different events that occur during the execution of the plan. They may also refer to the order of these events, or specify explicit time constraints. In addition, compositions should take into account the preferences of the user. In our travel example above, these may account to airline/hotel preferences, preferred times for travel, and preferred methods of payment. We also would expect a rich language to express preferences, that also allows expressing preferences over events, explicit time, action occurrences, etc.

We have presented WSC as a motivating task that presents many interesting characteristics. Many of these characteristics, however, do appear in other compelling applications. For example, in agent programming (or robot programming), we may require almost all of them.

The goal of this thesis is *not* to provide a solution to the WSC problem. Rather we use it here as a *catch-all* example, that shows many interesting problems that we are going to address in this thesis. To

this extent, WSC can be viewed as one of many potential applications for which our techniques could be applicable.

1.3 The Problems We Address

In this thesis, we deal with the following non-classical planning tasks.

- *Planning with temporally extended goals* (TEGs) (Chapter 3). Temporally extended goals have the ability to refer properties that may occur throughout the execution of a plan. As such, this problem addresses Characteristic 6 above.
- *Planning with rich user preferences*. In particular, we address the problem of planning with temporally extended preferences (TEPs) (Chapter 4). This relates to Characteristic 6.
- *Planning in the presence of procedural domain control knowledge* (Chapter 5). In particular, we consider procedural constraints expressed in a Golog-like language. Golog is a high-level robot language that can be used to specify the behaviour of agents. In relation to our example, Golog can be used to specify a "skeleton" of a solution to a planning task (such as a WSC task), containing loops and conditional constructs, but also containing "open parts" that need to be filled in by the a planner. As such, the work presented in this chapter relates to Characteristic 5 specified above.
- *Planning with programs that sense* (Chapter 6). Here we assume that the building blocks for plans are Golog programs. In addition, we consider that these programs can sense the environment. To deal with sensing, we move to a planning framework in which we can refer to the knowledge of the agent. In particular, the framework deals with incomplete initial states. As such, this chapter relates to Characteristics 2, 4, and 3.

1.4 Approach

To address each of the problems described above we use a common approach: *reformulation*. Our reformulation algorithms will take a non-classical planning task, and generate a new task. This new task is more amenable to be solved by current state-of-the-art techniques. In some cases, we can generate a *classical* task from a non-classical one. In other cases, we will not obtain a task that can be directly exploited by current state-of-the-art techniques and thus we *adapt* existing techniques to handle the reformulated tasks. Figure 1.2 shows a schematic view of our approach.

Our reformulation approach has the following advantages:

1. The main advantage is that in the majority of cases we generate some form of standard output, which can be directly input to a wide variety of planners. This is important because it means

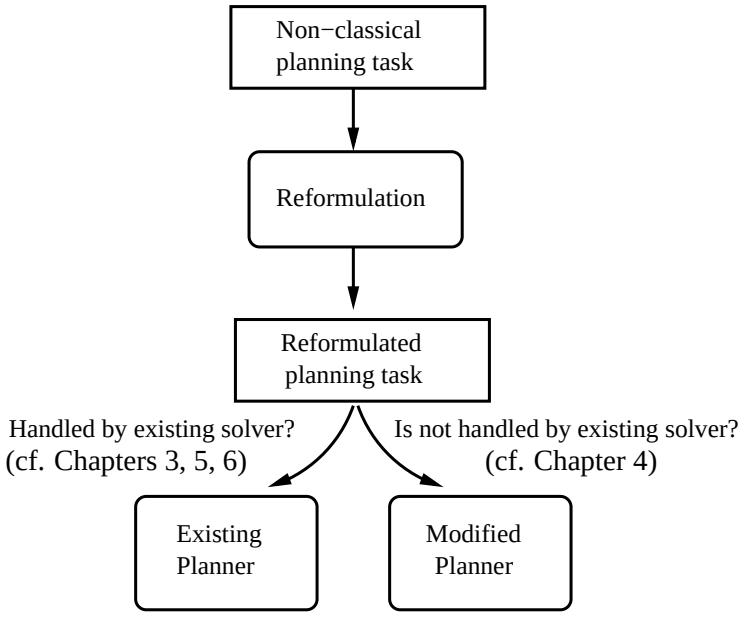


Figure 1.2: The reformulation approach that is taken in all chapters of this thesis. In most cases we solve the reformulated task with an existing planner. In others, we have modified and extended existing planners to handle the reformulated tasks.

that advances in classical planning can be immediately leveraged for these non-classical planning tasks.

2. Our approach is generalizable to other planning paradigms. Indeed, it can be utilized in multiple other scenarios that are not explicitly dealt with in this thesis. For example, our reformulation for temporally extended goals is not bound to deterministic planning, but also can be applied in a non-deterministic scenario. In that case our techniques admit minor modifications that enables our approach to generate a non-deterministic problem with final-state goals only. Similarly, the approach can be applied when in the original planning instance actions have associated costs, when there is incomplete knowledge, etc.

3. Our reformulation techniques are composable. That is, they can be applied in succession in order to address problems that are non-classical along several dimensions.

4. A reformulation approach serves as a *baseline* for future comparison. Indeed, if a classical planning technique T is adapted for planning in any of the tasks for which we have proposed reformulations, producing a new technique T', then T' should outperform technique T on the reformulated task in order to prove that is a valuable approach.

5. Finally, using the reformulation approach, it is possible to gain insights into how to adapt existing techniques for these non-classical planning tasks. Indeed, by observing how a planner behaves

with the reformulated instance it is possible to spot potential drawbacks of the classical approach when solving the particular reformulated task. This information can then be used to inspire the development of novel heuristics that avoid those drawbacks.

The notion of exploiting reformulation to solve planning problems is not new. Some previous work however reformulates the domain description – effectively the transition system (e.g. Palacios and Geffner, 2006; Yoon, Fern, Givan, and Kambhampati, 2008). Other previous work (e.g. Gazen and Knoblock, 1997; Helmert, 2009) represents the entire planning problem in a different language. Our work is more focused on the reformulation of the planning *objective*, where our notion of objective is defined in the large to include temporally extended goals, preferences, and also domain control knowledge. It is only in Chapter 6 that we more closely align ourselves to approaches that reformulate the entire problem. We discuss related work in each of the technical chapters of this document and also in Section 7.2.

1.5 Outline and Contributions

The remainder of this document is organized as follows. Chapter 2 describes the basics of classical planning and current state-of-the-art techniques. This chapter provides the necessary background for most of the rest of the document: Chapters 3, 4, 5. The necessary background for Chapter 6 will be given within the chapter. We draw conclusions and discuss future work in Chapter 7.

The major contributions of this thesis follow.

Planning with Temporally Extended Goals using Heuristic Search (Chapter 3)

As we have noted above, many compelling applications have planning goals that are not naturally characterized as conditions to achieve in the *final* state. In this chapter we deal with the problem of planning with TEGs. TEGs refer to properties that must hold over intermediate and/or final states of a plan. Current techniques for planning with TEGs only consider pruning the search space during planning via *goal progression* (e.g. Bacchus and Kabanza, 1998). Nevertheless, as we noted above, the fastest classical domain-independent planners rely on heuristic search. We propose a method for planning with TEGs using heuristic search. Thus, we reformulate planning task with TEGs into an equivalent *classical* planning task. With this translation in hand, we exploit heuristic search to determine a plan. Our translation is based on the construction of nondeterministic finite automata for the TEG. We propose two alternative translations: the first generates a task that contains operators with conditional effects, and the second (more efficient) uses *derived predicates*, a way to describe predicates of the domain using an axiomatic definition.

We prove the correctness of our algorithm and analyze the complexity of the resulting representation.

The translator is fully implemented and available. We show that our approach consistently outperforms existing approaches to planning with TEGs.

An abridged version of this chapter appeared in the Proceedings of AAAI'06 (Baier and McIlraith, 2006b).

Heuristic Planning with Temporally Extended Preferences (Chapter 4)

As previously observed, the objectives of many planning applications are stated not only in terms of hard goals but also in terms of preferences. In this chapter we focus on the problem of planning in the presence of rich user preferences. Planning with preferences involves not only finding a plan that achieves the goal, it requires finding a *preferred* (and ideally optimal) plan that achieves the planning objective, where preferences over plans are specified as part of the planner's input. In this chapter we propose a technique for accomplishing this objective. Our technique can deal with a rich class of preferences, including *temporally extended preferences*. Unlike simple preferences which express desired properties of the final state achieved by a plan, TEPs can express desired properties of the entire sequence of states traversed by a plan, allowing the user to express a much richer set of preferences. Our technique involves reformulating a planning problem with TEPs into an equivalent planning problem containing only simple preferences. This conversion is accomplished by augmenting the original planning domain with a new set of predicates and actions for updating these predicates.

The resulting task is not classical, since it contains preferences. To this end, we provide a collection of new heuristics and a specialized search algorithm that can guide the planner towards preferred plans. Under some fairly general conditions our method is able to find a most preferred plan—i.e., an optimal plan. It can accomplish this without having to resort to admissible heuristics, which often perform poorly in practice. Nor does our technique require an assumption of restricted plan length or make-span, as is the case of SAT/CSP-based approaches. We have implemented our approach in a planning system we call HPlan-P, and used it to compete in the 5th International Planning Competition, where it achieved distinguished performance in the *Qualitative Preferences* track.

A version of this chapter appeared published in *Artificial Intelligence* (Baier, Bacchus, and McIlraith, 2009).

Golog-Like Domain Control Knowledge in State-of-the-Art Planners (Chapter 5)

In the context of planning, Golog-like programs (Levesque, Reiter, Lespérance, Lin, and Scherl, 1997) are suitable for declaring procedural constraints that restrict the search space significantly, allowing a solver to find solutions more quickly. These procedural constraints are step-by-step specifications—like those usually specified by an imperative program—of how a goal must be achieved. Additionally, Golog allows specifying non-determinism in its programs. These provide "open parts" that must be filled in by

the planner.

Thus, Golog programs can be used to specify plan skeletons, which have been proposed as an approach to WSC (McIlraith and Son, 2002). Golog, however, can also be used to represent *domain control knowledge* (DCK). DCK, used in conjunction with blind search has proven a to be a successful technique for increasing planning speed, sometimes by orders of magnitude. It is only successful however when very well-crafted control knowledge can be written.

The contribution of this chapter is threefold. First, we propose a new procedural control language for representing DCK, specifically tailored for planning applications. The language is closely inspired by Golog, offering natural constructs for DCK specification, such as iteration and nondeterminism.

We show that any planner that can input planning tasks in PDDL (the current *de facto* input standard of the planning community) is able to plan with our DCK. We do this by giving an algorithm that reformulates any PDDL planning task and a control program, into an equivalent, program-free PDDL task whose plans are only those that "behave" according to the control program.

Third, we show that the resulting planning task is amenable for use with domain-independent heuristic planners. In particular, we propose three approaches. The first (control-aware) directly uses the resulting PDDL task, the second (control-unaware) uses the set of operators of the original task to compute the heuristic, and the third (control-aware) uses a modified set of operators from the resulting PDDL task to compute the heuristic to better inform the heuristic about the control. Our experiments on familiar planning benchmarks show that the combination of DCK and heuristics produce better performance than using DCK with blind search and than using heuristics alone.

A version of this chapter appeared in the Proceedings of ICAPS'07 (Baier, Fritz, and McIlraith, 2007).

Planning with Programs that Sense (Chapter 6)

In this chapter we address the problem of planning by composing *programs*, rather than or in addition to primitive actions. The programs that form the building blocks of such plans can, themselves, contain both sensing and world-altering actions. This is primarily motivated by the problem of automated composition of component software, since Web services are programs that can sense and act. Our further motivation is to understand how to exploit macro-actions in existing operator-based planners that plan with sensing. We study this problem in the language of the situation calculus, appealing to Golog to represent our programs. To this end, we propose an offline execution semantics for Golog programs with sensing. We then propose a compilation method that transforms our action theory with programs into a new theory where programs are replaced by primitive actions. This enables us to use state-of-the-art, operator-based planning techniques to plan with programs that sense for a restricted but compelling class of programs. Finally, we discuss the applicability of these results to existing operator-based planners that support sensing and illustrate the computational advantage of planning with programs that sense

via an experiment. The work presented here is cast in the situation calculus to facilitate formal analysis. Nevertheless, both the results and the algorithm can be trivially modified to take PDDL as input and output. This work has broad applicability to planning with programs or macro-actions with or without sensing.

An abridged version of this chapter appeared in the Proceedings of KR'06 (Baier and McIlraith, 2006a).

Chapter 2

Planning: Languages and Algorithms

This chapter describes necessary background in classical planning. It first describes classical planning, along with the popular formalisms and languages used to represent these problems. It finishes by describing some of the most successful techniques developed in the last few years to effectively solve these problems.

2.1 Classical Planning

A *classical planning task (or instance)* is a tuple $I = (\mathcal{F}, s_0, \mathcal{O}, \mathcal{G})$, where $\mathcal{F}$ is a finite set of facts[1], $s_0 \in \mathcal{F}$ is the initial state, $\mathcal{O}$ is a finite set of action operators, and $\mathcal{G}$ is a goal condition. An action operator $o \in \mathcal{O}$ maps a state into another state. The classical planning problem consists of finding a sequence of action operators $a_1 a_2 \cdots a_n$, which, when applied to the initial state, will produce a state that satisfies the goal condition $\mathcal{G}$.

Classical planning, as defined above, requires to find *any* plan. However, in most applications we are required to find high-quality solutions. Although adding a quality measure immediately means that we are out of the classical planning paradigm, the quality of solutions is still considered when evaluating a planner's performance. A usual quality measure for plans is some cost function defined over the set of possible plans. The planning community usually distinguishes between planners that, given such a cost function, will seek for an optimal solution, and planners that will not. In the former case, we use the term *optimal* planner to refer to such a planner, and in the latter will use the term *satisficing* planner.

The planning community has developed a variety of languages to define planning instances. STRIPS (Fikes and Nilsson, 1971) and ADL (Pednault, 1989) are two of the most prominent languages for the representation of classical planning problems. We describe them in more detail below. Additionally, we

[1] Although not required by the model, facts are usually represented by first-order ground literals.

briefly describe PDDL, a language designed to represent planning problems that is used as a standard in the planning community. We explicitly leave out a review of the SAS+ formalism (Bäckström and Nebel, 1995), as it has little relevance to the contents presented in this thesis.

2.1.1 STRIPS

STRIPS (Fikes and Nilsson, 1971), is the oldest and most used formalism to represent classical planning problems. Here, operators are described as triples $(pre(o), add(o), del(o))$. Each of the components of the triple is a list of facts in $\mathcal{F}$. List $pre(o)$ contains the *preconditions* of the operator, i.e., the facts that must hold true prior to the application of operator o. List $add(o)$—the *add-list* of o—contains the positive effects of o. Finally, list $del(o)$—the *delete-list* of o, contains the negative effects of an operator.

A planning state is simply a collection of facts in $\mathcal{F}$. An operator is applicable in state s iff $pre(o) \subseteq s$. The result of applying operator o in s, denoted as $\gamma(s,o)$, is defined as

$$\gamma(s,o) = (s \setminus del(o)) \cup add(o).$$

2.1.2 ADL for Classical Planning

In Pednault's ADL formalism (1989), facts in $\mathcal{F}$ are typically first-order ground atoms formed from a finite set of predicates *Pred* and a finite set of objects *Objs*.[2] On the other hand, preconditions and effects, can now be more than simple lists of ground predicate literals. ADL preconditions can be arbitrary boolean formulae, existentially or universally quantified over the set of objects *Objs*. ADL effects can be *conditional*, which means that adds and deletes can be conditioned on the satisfaction of arbitrary boolean formulae. Effects can also be *universal* in the sense that they affect *all* objects that satisfy a certain condition. For example, assume we are describing a domain where objects can contain other objects. Further, assume action $move(x,y,z)$ moves object x from location y to location z and in the process moves all objects in x to z as well. The precondition for this action is just $at(x,y)$; i.e., the object x has to be at location y, while its effects can be defined by the list:

$$Eff = \{\textbf{add}\ at(x,z), \forall v\,[in(v,x) \Rightarrow \textbf{add}\ at(v,z)], \textbf{del}\ at(x,y), \forall v\,[in(v,x) \Rightarrow \textbf{del}\ at(v,y)]\}$$

Thus, the location of the object x and all objects inside x changes to z.

In addition to more expressive preconditions and effects, ADL also allows for the representation of functions. This means that states can contain, in addition to propositional facts, sentences of the form $f(\vec{c}) = z$, where f is a function name, $\vec{c}$ is a tuple of objects in *Objs*, and z is an object in *Objs*. Actions can change the functions by assigning $f(\vec{c})$ a different value as an add effect.

[2] It is also standard to utilize ground predicate facts in STRIPS, even though the notion of object is not really required.

```
(:action LOAD-TRUCK
   :parameters    (?pkg - package ?truck - truck ?loc - place)
   :precondition  (and (at ?truck ?loc) (at ?pkg ?loc))
   :effect        (and (not (at ?pkg ?loc)) (in ?pkg ?truck)))

(:action DRIVE-TRUCK
  :parameters (?truck - truck ?loc-from - place ?loc-to - place ?city - city)
  :precondition
   (and (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?city))
  :effect
   (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
```

Figure 2.1: Two operators defined in the *logistics* domain. Actions have parameters, preconditions, and effects. Parameters are of a specific *type*. Positive effects are described by positive literals, while negative effects are represented by negated literals.

Finally, in ADL, *Goal* can be any formula (possibly quantified) that describes a condition that must be satisfied by a goal state. For more details on ADL we refer the reader to Pednault's paper (1989).

2.1.3 PDDL

The *Planning Domain Definition Language* (PDDL) was proposed by McDermott (1998) as a standard input language for the first International Planning Competition. Since its inception, many planners have adopted and thus it has become a *de facto* standard input language.

Although PDDL is a standard, there are many variants of the language. Indeed, new features have been introduced with almost every planning competition. Current versions of PDDL allow the definition of ADL planning problems, but also go beyond ADL, by allowing the user to express explicit time and functional fluents (cf. PDDL2.1, Fox and Long, 2003), *derived predicates* or axioms (cf. PDDL2.2, Edelkamp and Hoffmann, 2004), temporally extended preferences and hard constraints (cf. PDDL3, Gerevini, Haslum, Long, Saetti, and Dimopoulos, 2009), and *object fluents* (cf. PDDL3.1[3]). We will give a more in-depth description of PDDL3 later in this document (Section 4.2.3, p. 57), just before we describe our techniques for planning with preferences.

PDDL separates the definition of a planning instance in two parts: the *domain definition*, and the *problem definition*. The domain part describes domain-specific elements, including a declaration of the predicates used to describe the domain, and a definition of the object types. Moreover, the dynamics of the domain is defined using a set of action operators. Figure 2.1 shows the definition of two action operators used in the *logistics* planning benchmark.

[3]No formal publication exists at the moment. See `http://ipc.informatik.uni-freiburg.de/PddlExtension` for details.

In the problem definition, one declares the objects, of the problem, the initial state, and the goal.

2.1.4 Some Complexity Results

Complexity of STRIPS planning

Classical planning, even in the limited STRIPS formalism, is a hard problem, as is established by the following theorem.

Theorem 2.1 (Bylander, 1994) PLANSAT, *i.e, the problem of deciding whether there exists a plan for a* STRIPS *planning instance I is* PSPACE*-complete.*

Proof sketch: First, note that because there are $2^{|\mathcal{F}|}$ states, in the worst case an instance can be such that a plan has to visit all states before achieving the goal. The size of a plan in the STRIPS formalism could be therefore exponential in $|I|$. A "real-world" example of the need for exponential plans is the classical *Towers of Hanoi* game.

However, for determining plan existance we do not need to construct a plan. Since we know that the plan is of size at most $2^{|\mathcal{F}|}$, we can verify existence with a non-deterministic polynomial-space algorithm shown in Figure 2.2, with $s = s_0$, and $n = 2^{|\mathcal{F}|}$. Such an algorithm does not need more than polynomial space, since its parameter n can be represented efficiently with $\log n$ bits. This means that PLANSAT is in NPSPACE. Membership in PSPACE follows from Savitch's theorem (1970).

The proof that PLANSAT is PSPACE-hard is tedious, and we do not replicate it here. In short, the proof follows from the fact that the transitions of any polynomial-space Turing machine can be encoded by a polynomial number of operators. Details can be found in Bylander's paper (1994). ■

Another problem of interest is that of finding a plan of bounded length for a given instance. This problem is also PSPACE-complete.

Theorem 2.2 (Bylander, 1994) PLANMIN, *i.e., the problem of deciding whether there exists a plan for a* STRIPS *planning instance I with k or fewer actions, where k is given as input, is* PSPACE*-complete.*

Proof sketch: PLANSAT is reducible to PLANMIN in polynomial space, since we just need to output (I,k) for $k = 2^{|\mathcal{F}|}$. This proves PSPACE hardness. Furthermore, it is possible to determine the existence of a plan with at most k actions by calling algorithm of Figure 2.2, with $n = k$. This proves that PLANMIN is in PSPACE. ■

Although PLANSAT and PLANMIN are PSPACE-complete in general, under some restrictions these problems can be shown to be lower in the polynomial hierarchy. Indeed, if we modify PLANSAT, by restricting the plan length to be bounded by $p(|I|)$, where $p(n)$ is a polynomial in n with $p(n) \geq n$, then the resulting decision problem is NP-complete (see e.g. Baral, Kreinovich, and Trejo, 2000). On the other hand, Helmert (2003, 2006b) has shown that for many of the planning benchmarks that have been

```
1:  function PLAN-EXISTS(state s, natural n)
2:    if G is satisfied by s then
3:      return true
4:    else if n > 0 then
5:      for some s' that is a successor of s do
6:        if PLAN-EXISTS(s', n − 1) then return true
7:      end for
8:    else return false
9:    end if
10: end function
```

Figure 2.2: A nondeterministic, polynomial-space algorithm to determine the existence plan of length at most n

used in the International Planning Competition, PLANMIN and PLANSAT are in NP. As an example, in the *logistics* domain, PLANSAT is in P, whereas PLANMIN is NP-complete. The *blocks world* has also be proven to be NP-complete for PLANMIN and tractable for PLANSAT (Gupta and Nau, 1992).

Another important aspect is the relationship between the complexity of plan existence and *plan generation*. Although it is the latter task which is of more interest, there are fewer results available in the literature. Note that since plans could be exponential in the size of the problem plan generation is generally in EXPTIME. This complexity gap appears even when plan existence is tractable. For instance, Jonsson and Bäckström (1998) have shown a family of planning instances in which plan existence is a tractable task whereas plan generation is exponential.

Complexity of ADL vs. STRIPS

Any ADL problem can be translated into a STRIPS instance. However, existing compilation techniques—such as Gazen and Knoblock's (1997)—are worst-case exponential. This wort case cannot be improved if we are willing to preserve the length of plans polynomially (Nebel, 2000), and thus ADL is strictly more succinct than STRIPS.

ADL planning, however, is still a PSPACE-complete problem. Observe that the algorithm of Figure 2.2 also requires polynomial space even if the preconditions and effects are complex formulae.

Interestingly, most of the top-performing approaches to classical planning internally utilize a STRIPS-like representation. In fact, all planners shown in Figure 1.1 first translate the ADL instance into one that is essentially a STRIPS one (without conditional or quantified effects or goals).

Most of the reformulation algorithms we propose later in this document generate ADL problems. As we will see, this has some practical implications if we want to use some state-of-the-art planners. Most of these issues can be addressed by moving to more expressive solvers, or by utilizing ADL-native

solvers. More details are discussed in Chapters 3 and 4.

2.2 Planning as Heuristic Search

Many state-of-the-art domain-independent planners use domain-independent heuristics to guide the search for a plan. Heuristics estimate the cost of achieving the goal from a certain state. They can be used with standard search algorithms, and are usually key to good performance. They are typically computed by solving a relaxed version of the original problem.

There are a few domain-independent relaxations that are widely used by current planners. One of them corresponds to ignoring the negative effects of actions. This relaxation is called the *delete-relaxation*.

Definition 2.1 (Delete-Relaxation) *Let I be STRIPS planning instance. The delete relaxation of I, denoted I^+, is a instance just like I but in which operators in $\mathcal{O}$ have an empty delete list.*

Classical planners like HSP (Bonet and Geffner, 2001) and FF (Hoffmann and Nebel, 2001), among others, use this relaxation to compute its heuristic.

We will focus our attention on the delete-relaxation, but we do not want to omit a brief note on Helmert's relaxation of the causal graph of a task (2006a). Here, the domain is represented by a set of variables (SAS+ representation). A *causal graph* represents dependencies between variables. If such a graph is acyclic, a solution to the problem can be computed in polynomial time. Helmert's causal graph heuristic (2006a) is computed by relaxing the causal graph (by ignoring certain preconditions) to the point that it becomes acyclic. Then, polynomial algorithms are used to obtain an estimation of the cost to a solution.

The rest of the section describes key aspects of the FF planner. There are two reasons to look into this planner more closely. First, FF is one of the most influential planners developed in the last decade: many other classical planners used techniques developed by FF in some way. Second, some of the heuristics we propose in Chapters 4 and 5 are modifications of the standard FF heuristic.

2.2.1 FF

FF (Hoffmann and Nebel, 2001) is a classical planner that employs heuristic, forward search to find a plan. The key novel aspects of the planner are its heuristic, and its search algorithm. We describe each of those in turn.

FF Heuristic

The FF heuristic for a state s is computed by finding a plan from s in the delete-relaxation of the problem. This plan is referred to as *relaxed plan*. FF computes the relaxed plan using a modification of

the GRAPHPLAN planner (Blum and Furst, 1997). It thus, computes a *relaxed planning graph*, which is the graph that would be generated by GRAPHPLAN for the delete-relaxation. This graph is composed of fact layers—or *relaxed states*—and action layers. The action layer at level n contains all actions whose preconditions are contained in the relaxed state at depth n. The relaxed state at depth $n+1$ contains all the facts that hold at layer $n+1$ and is generated by applying all the positive effects of actions in action layer n.

Since in the delete relaxation, actions have no negative effects, the *relaxed planning graph* contains no mutexes (mutually exclusive facts or actions). This implies that, to find a plan, we only need to expand the graph just until the point at which the goal is satisfied. FF computes a relaxed plan for the goals by regression from the goal facts in the graph to the current state s. The length of this plan is then used as a heuristic estimator of the cost for achieving the goal. Henceforth we refer to the FF heuristic value as $h^{FF}(s)$. This takes polynomial time, since it only implies a traversal of the graph, whose size is polynomial in the size of the problem.

Before explaining some details on the extraction algorithm, note that if the goal does not appear in any fact layer of the relaxed graph, then the problem is proven unsolvable. To some extent, this dead end detection can be quite powerful. We exploit this power in Chapter 4, when we design a pruning function for planning with preferences.

The FF extraction algorithm (Figure 2.3) has a built-in heuristic that aims at extracting the *smallest* possible relaxed plan. The objective is to be as close as possible from the optimal solution to I^+ from s.[4] Specifically, the heuristic rule specifies that whenever an achieving action (achiever) is chosen, then we prefer always the *earliest* achiever, i.e., the one that appears at the lowest level in the relaxed graph. If there are ties, it will prefer the achiever that has the lowest *precondition cost*, were the precondition cost is defined as the sum of the levels at which the achiever's precondition facts first appear in the relaxed graph.

FF's Search Algorithm

The FF's search algorithm is less relevant to this thesis. We explain it here basically to introduce the concept of *helpful action*, which is one the most interesting enhancement introduced in the search algorithm. Later, in Chapter 4, we mention an extension to our HPLAN-P system that benefits from this technique.

FF uses two search algorithms that are used in turn. The first is *enforced hill climbing* (EHC), a modification of the standard hill-climbing search. If EHC fails to find a plan, then a standard best first search is invoked, in which $h^F F(s)$ is used as the evaluation function.

EHC is a greedy and incomplete algorithm for planning. It builds a plan by performing a sequence

[4] The length of the optimal plan from s in the delete relaxation is usually referred to as $h^+(s)$. Its computation NP-hard (Bylander, 1994).

```
1: function EXTRACTPLAN(plan graph S_0 A_0 S_1 ··· A_{n-1} S_n, goal G)
2:     for i = n ... 1 do
3:         G_i ← goals first reached at level i
4:     end for
5:     for i = n ... 1 do
6:         for all g ∈ G_i not marked TRUE at time i do
7:             Find min-cost a ∈ A_{i-1} such that g ∈ add(A_{i-1})
8:             RP_{i-1} ← RP_{i-1} ∪ {a}
9:             for all f ∈ prec(a) do
10:                G_{layerof(f)} = G_{layerof(f)} ∪ {f}
11:            end for
12:            for all f ∈ add(a) do
13:                mark f as TRUE at times i − 1 and i.
14:            end for
15:        end for
16:    end for
17:    return RP
18: end function
```

Figure 2.3: The FF extraction algorithm (Hoffmann and Nebel, 2001) receives a relaxed planning graph as a succession of pairs of state and action layers, and a set of goal facts G. $layerof(f)$ denotes the depth of the fact layer at which f first appears.

of improvement phases. In each phase, it takes the current state s and searches for a descendant of s, s', such that $h^F F(s') < h^F F(s)$, i.e, such that its heuristic value has improved. Once s' has been found, the actions that lead to s are added to the current plan prefix, and a new improvement phase is started. The pseudo-code for EHC is shown in Figure 2.4.

The search for s' is a simple *breadth-first* search. During this search however, only the successors of a node that are produced by a *helpful action* are added to the search space.

Definition 2.2 (Helpful Actions) *Let F be the set of facts in variable G_1 after the plan extraction of the algorithm of Figure 2.3 finishes. The helpful actions for state s are those actions that are applicable in*

```
1: function EHC(initial state I, goal G)
2:     plan ← EMPTY
3:     s ← I
4:     while h(s) ≠ 0 do
5:         from s, search for s' such that h(s') < h(s).
6:         if no such state is found then
7:             return fail
8:         end if
9:         plan ← plan ∘ "actions on the path to s'"
10:        s ← s'
11:    end while
12:    return plan
13: end function
```

Figure 2.4: Enforced Hill Climbing (EHC) (Hoffmann and Nebel, 2001)

s and that achieve a fact in F.

The restriction to use helpful actions only to generate successors in the breadth first phase contributes to the incompleteness of the EHC algorithm. Nevertheless, the impact that it has in performance is such that it is still worth to use it. Hoffmann and Nebel (2001) show that this technique contribute to the overall performance of FF.

Other classical planning systems use similar concepts. For example, both FAST-DOWNWARD and LAMA use *preferred operators*. In the case of FAST-DOWNWARD, this notion is defined analogously to helpful actions but for the causal graph heuristic. LAMA on the other hand, uses the FF helpful actions.

FF uses a handful of additional techniques that are important for performance. Among them is the *goal agenda*, which specifies an ordering between the goals that are achieved. Also, a heuristic is used to order helpful actions. We refer the reader to the original paper for more details.

Chapter 3

Heuristic Planning for Temporally Extended Goals

3.1 Introduction

As we have seen in the first chapter, compelling applications of planning require the ability to express goals and/or preferences that refer to properties that must be achieved at various states during the execution of the plan. Examples of these include achieving several goals in succession (e.g., deliver all the priority packages and then deliver the regular mail, pick up mail from the mail room before making deliveries to offices, book my hotel after you book my flight), safety goals such as maintenance of a property (e.g., always maintain at least 1/4 tank of fuel in the truck, ensure my credit card is never over its limit), conditional temporal goals (if the robot reaches a low battery level, it should immediately recharge), and achieving a goal within some number of steps (e.g., the truck must refuel at most 3 states after its final delivery). All these goals are known as *temporally extended goals* (TEGs), since they refer to different states of the execution in a temporal manner. We distinguish TEGs explicitly from *temporal goals*. The former do not refer to time in an explicit way while the latter do.

TEGs are typically represented (see e.g. Bacchus and Kabanza, 1998) using Linear Temporal Logic (LTL) (Pnueli, 1977). This logic allows specifying properties of infinite sequences of states, and therefore it can be naturally used for representing TEGs.

In the current literature, however, there is a clear mismatch between state-of-the-art techniques used for classical planning and the techniques used for planning with TEGs. Among the few planners that are able to plan with TEGs we find TLPLAN (Bacchus and Kabanza, 1998). TLPLAN can be configured to use human-encoded, domain-dependent heuristics, but in the absence of these it simply uses *blind* search to plan for a goal. In addition, it will prune from the search space those states that can be proven

to violate the LTL formula. This is achieved through a *progression* mechanism, which re-writes the LTL formula, into an equivalent one, which is written in terms of a property that has to be checked in the current state and a property that has to be checked in a successor state. A state can be effectively pruned if the progressed formula is equivalent to *false*, i.e., it is logically unsatisfiable.

Blind search in conjunction with state pruning via progression of the LTL formula can be very effective when the LTL formula is intended to constrain or *control* the search. Indeed, Bacchus and Kabanza (2000) have shown that the efficiency of classical planning can be significantly improved by expressing domain-specific search-control knowledge in the form of LTL constraints. TLPLAN, enhanced with such rules, won first place in the International Planning Competition in 2002 (hand-coded track).

Nevertheless, state pruning by TEG progression will not provide any pruning for many natural LTL properties—and therefore no improvement whatsoever over blind search. Consider, for example, the property "eventually p". Such a goal will never progress to *false*. Without getting into the technical details of why this is true, we explain this in an intuitive manner.[1] Intuitively, such a goal formula progresses to *false* in a state s if and only if it is not possible to reach p by any means from s. Although the latter statement could be true for s, the only way to actually prove it would be by doing some kind of domain analysis. Such an analysis is not done by progression because it is a mechanism that only manipulates the LTL goal formula syntactically.

On the other hand, as noted earlier, among the fastest domain-independent planners are those that use heuristics. In planning for a TEG a heuristic should be expected to estimate the cost of achieving the TEG. However, it is not clear how to adapt current heuristic methods to TEGs. This is due to the fact that current heuristics only work for final-state goals.

3.1.1 Contributions of this Chapter

In this chapter we propose a method for performing heuristic search on planning problems with TEGs by exploiting the relationship between temporal logic and automata. Our approach is as follows. Given planning problem for a TEG, we transform it into a classical planning problem and apply a domain-independent heuristic search planner to actively guide search towards the goal. This new augmented domain, contains additional predicates that allow us to describe the (level of) achievement of the TEG. In particular, in this new domain there is a *classical goal* that is satisfied iff the TEG of the original problem is achieved.

The contributions of this chapter are the following:

1. We introduce a new logic for describing TEGs: the f-FOLTL logic. f-FOLTL is a version of LTL which we modify to include first-order (FO) quantifiers and to be interpreted only by finite

[1] More technically, $\Diamond p$ always progresses to $p \vee \bigcirc \Diamond p$ which never reduces to false, independent of the truth value of p in the current state.

computations. While this logic is not new, its application to planning is new. Its use enables the construction of a sound translation algorithm. Moreover, we argue that it is more intuitive when dealing with finite plans.

2. We provide and prove the correctness of an algorithm that given an f-FOLTL formula φ, generates a parametrized, nondeterministic finite automata (PNFA), A_φ, whose is the set of models of φ. Parametrized automata avoid grounding the goal formula, and in doing so, avoid potential blowups. In addition a parametric translation can be better exploited by planners that do not ground the planning instance. Nevertheless, the size of A_φ is worst-case exponential in the size of φ. This motivates our next contribution.

3. We provide a simplification technique that allows reducing the size of the resulting compilation. In particular, this avoids blowups of simple goals that are quite "natural" in planning with TEGs.

4. We provide two alternative methods for representing the PNFA within a planning domain. In both translations, each of the states of the PNFA is represented by a planning predicate. In particular the accepting states of the PNFA are regular predicates. The output of both methods is a PDDL problem description, making our approach amenable to use with a variety of classical planners. The first method defines the dynamics of the new predicates using goal regression (Waldinger, 1977); the second, defines them axiomatically, using PDDL axioms (Hoffmann and Edelkamp, 2005), a recent extension to the PDDL language. By representing the PNFA in the planning domain we actually provide a compilation of TEGs into classical goals.

5. We show, through an experimental analysis, that our approach, used with the heuristic search planners FF and $\text{FF}_\mathcal{X}$, consistently outperforms non-heuristic techniques. The analysis is carried out in benchmark domains extended with TEGs. We also experimentally observe that the worst-case exponential blowup does not manifest itself for practical goals.

3.2 Preliminaries

In this section we define the background for the rest of the chapter. We start by introducing f-FOLTL, a logic for representing quantified TEGs for finite plans. We continue by formally defining the planning instance for TEGs, and by reviewing regression, a technique that will be used in the rest of the chapter.

3.2.1 f-FOLTL: Finite LTL with FO Quantifiers

LTL (Pnueli, 1977) allows specifying temporal properties about infinite sequences of states. Therefore, it can be naturally used to express temporally extended goals (see e.g. Bacchus and Kabanza, 1998). In

this chapter we introduce f-FOLTL, a variant of LTL, that we use to represent our TEGs. f-FOLTL, as opposed to LTL, is interpreted over *finite computations* rather than over infinite ones.

There are two motivations for introducing a new logic to represent TEGs. The first motivation is a methodological one: we want to apply state-of-the-art planning technology for planning with TEGs, and current technology generally only applies to the generation of finite, linear sequences of actions. The second motivation—a more pragmatical one—is that our translation to finite-state goals will be done by representing a TEG by an automaton. Logics with infinite models, like LTL, require the use of Büchi automata, which have an accepting condition that is difficult to express as a simple expression involving domain predicates.

f-FOLTL is not a new logical language; the decidability of a slightly different version was analyzed by Cerrito, Mayer, and Praud (1999). However, to the best of our knowledge, it has not been used for representing planning goals before.

Syntax

f-FOLTL formulae augment LTL formulae with first-order quantification and by the use of the distinguished predicate final, which is only true in final states of computation. As usual, we assume our f-FOLTL formulae are built using standard temporal and boolean connectives from a set S of symbols for predicates and functions.[2] We denote by $\mathcal{L}_{FO}(S)$ the set of first-order formulae over the set of symbols S, the boolean connectives $\vee$, $\wedge$, and the quantifier $\forall$.

Definition 3.1 (f-FOLTL formula) *The set $\mathcal{L}(S)$ of f-*FOLTL *formulae over set of symbols S is the least satisfying the following properties.*

1. *The 0-arity predicates* final, true *or* false *are in $\mathcal{L}(S)$.*
2. *If $\varphi \in \mathcal{L}_{FO}(S)$ then $\varphi \in \mathcal{L}(S)$.*
3. *$\neg\psi$, $\psi \wedge \chi$, $\bigcirc\psi$, or $\psi \mathsf{U} \chi$, are all in $\mathcal{L}(S)$ if ψ and χ are in $\mathcal{L}(S)$.*
4. *$(\forall x)\,\varphi$, $(\exists x)\,\varphi$ are in $\mathcal{L}(S)$ if so is φ.*

As usual, a f-FOLTL *sentence* is a formula with no free variables. Moreover, to simplify the notation, we assume precedence of the $\wedge$ connective over the $\vee$ connective.

Semantics

f-FOLTL formulae are interpreted over finite computations. Finite computations are finite sequences of first-order interpretations that share a common domain and a common interpretation for function symbols.

[2] Note that *constants* are function of arity 0.

Definition 3.2 (Finite First-Order Computation) *Given a set of symbols for predicates and functions S, a* finite first-order computation *on S is a sequence $\sigma = s_0 s_1 \cdots s_n$, where each $s_i \in \sigma$ is a first-order interpretation $\langle \mathcal{D}, \mathcal{I}_F, \mathcal{I}_P^i \rangle$, where $\mathcal{D}$ is the (unique) non-empty domain, $\mathcal{I}_F$ is a (unique) interpretation for function symbols in S, and $\mathcal{I}_P^i$ is an interpretation for predicate symbols in S.*

As a consequence of interpreting a formula using these computations, the logic provides *rigid* functions, i.e., all constants and functions in the language refer to the same objects at any time point.

Definition 3.3 (Truth of an f-FOLTL Formula) *Let φ be an f-*FOLTL *formula, σ be a finite first-order computation over domain $\mathcal{D}$, and ν be a function mapping the variables in φ to elements in $\mathcal{D}$. Moreover, let σ_i denote the suffix $s_i s_{i+1} \cdots s_n$ of σ. We say that $\sigma \models \varphi$ (i.e., σ is a model of φ) iff $\langle \sigma_0, \nu \rangle \models \varphi$, for any ν. Furthermore,*

- $\langle \sigma_i, \nu \rangle \models \mathsf{final}$ *iff* $i = n$.
- $\langle \sigma_i, \nu \rangle \models \mathsf{true}$ *and* $\langle \sigma_i, \nu \rangle \not\models \mathsf{false}$.
- $\langle \sigma_i, \nu \rangle \models \varphi$, *where φ is a first-order formula ($\varphi \in \mathcal{L}(S)$) iff* $\langle s_i, \nu \rangle \models \varphi$.
- $\langle \sigma_i, \nu \rangle \models \neg\varphi$ *iff* $\langle \sigma_i, \nu \rangle \not\models \varphi$.
- $\langle \sigma_i, \nu \rangle \models \psi \wedge \chi$ *iff* $\langle \sigma_i, \nu \rangle \models \psi$ *and* $\langle \sigma_i, \nu \rangle \models \chi$.
- $\langle \sigma_i, \nu \rangle \models \bigcirc\varphi$ *iff* $i < n$ *and* $\langle \sigma_{i+1}, \nu \rangle \models \varphi$.
- $\langle \sigma_i, \nu \rangle \models \psi \,\mathsf{U}\, \chi$ *iff there exists a* $j \in \{i, \ldots, n\}$ *such that* $\langle \sigma_j, \nu \rangle \models \chi$ *and for every* $k \in \{i, \ldots, j-1\}$, $\langle \sigma_k, \nu \rangle \models \psi$.
- $\langle \sigma_i, \nu \rangle \models (\forall x)\,\varphi$, *iff for every* $a \in \mathcal{D}$, $\langle \sigma_i, \nu[x \to a] \rangle \models \varphi$, *where $\nu[x \to a]$ differs from ν only in that it assigns a to the variable x.*

Standard temporal operators such as *always* ($\Box$), *eventually* ($\Diamond$), and *release* (R), typical binary connectives such as $\vee, \supset, \equiv$, and the existential quantifier $\exists$ are defined in terms of these basic elements as follows.

$$(\varphi \vee \psi) \stackrel{\text{def}}{=} \neg(\neg\varphi \wedge \neg\psi), \qquad (\varphi \supset \psi) \stackrel{\text{def}}{=} (\neg\varphi \vee \psi),$$
$$(\varphi \equiv \psi) \stackrel{\text{def}}{=} (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi), \qquad (\exists x)\,\varphi \stackrel{\text{def}}{=} \neg(\forall x)\,\neg\varphi,$$
$$(\psi \,\mathsf{R}\, \chi) \stackrel{\text{def}}{=} \neg(\neg\psi \,\mathsf{U}\, \neg\chi), \qquad \Box\varphi \stackrel{\text{def}}{=} (\mathsf{false}\,\mathsf{R}\,\varphi),$$
$$\Diamond\varphi \stackrel{\text{def}}{=} (\mathsf{true}\,\mathsf{U}\,\varphi).$$

The definitions for *valid* and *satisfiable* formula are the same as in LTL. They follow.

Definition 3.4 (Valid formula) *We say that a formula ϕ is valid, denoted by $\models \phi$, if for every computation σ, $\sigma \models \phi$.*

Definition 3.5 (Satisfiable formula) *We say that a formula ϕ is satisfiable, if for some computation σ, $\sigma \models \phi$.*

As in LTL, any f-FOLTL formula can be rewritten as formula that specifies an atemporal condition that must hold in the first state and a condition that has to be verified in the following state. Identity (1) below can be used to transform any formula to that form. That and other identities below are key to the design of the algorithm that transforms f-FOLTL formulae to automata.

Proposition 3.1 *Let φ ψ, and χ be f-*FOLTL *formulae, and assume variable x is not free in ψ. The following formulae are valid.*

1. $\psi \,\mathsf{U}\, \chi \equiv \chi \vee \psi \wedge \bigcirc(\psi \,\mathsf{U}\, \chi)$,
2. $\neg\bigcirc\varphi \equiv \mathsf{final} \vee \bigcirc\neg\varphi$,
3. $\psi \,\mathsf{U}(\exists x)\, \varphi \equiv (\exists x)\,(\psi \,\mathsf{U}\, \varphi)$,
4. $\psi \,\mathsf{R}(\forall x)\, \varphi \equiv (\forall x)\,(\psi \,\mathsf{R}\, \varphi)$,
5. $\psi \,\mathsf{R}\, \chi \equiv \chi \wedge (\mathsf{final} \vee \psi \vee \bigcirc(\psi \,\mathsf{R}\, \chi))$.

Proof: See Section A.1 (page 162). ■

Limiting f-FOLTL to finite computations results in several obvious discrepancies in the interpretation of LTL and f-FOLTL formulae. In particular, discrepancies can arise with LTL formulae whose models can only be infinite. For example, in f-FOLTL the formula $\Box(\varphi \supset \bigcirc\psi) \wedge \Box(\psi \supset \bigcirc\varphi)$ is equivalent to $\Box\neg(\varphi \vee \psi)$. This is because if φ or ψ were true in some state of a model, the model the formula would have to be an infinite sequence of states. A second example is the LTL formula $\Box p$ which in f-FOLTL is *not* equivalent to $p \wedge \bigcirc\Box p$. If it were, $\Box p$ could never be true in computations with a single state. The interpretation of the $\bigcirc$ operator, represented by identity 2 of Proposition 3.1, is also a source of discrepancies. The reader familiar with LTL, will note that identity 2 replaces LTL's equivalence $\neg\bigcirc\varphi \equiv \bigcirc\neg\varphi$. This formula does not hold in f-FOLTL because $\bigcirc\varphi$ is true in a state iff there exists a next state that satisfies φ. Since our logic refers to finite sequences of states, the last state of each model has no successor, and therefore in such states $\neg\bigcirc\varphi$ holds for any φ.

Although there are differences between LTL and f-FOLTL, their expressive power is similar when it comes to describing temporally extended goals for finite planning. Indeed, f-FOLTL has the advantage that it is tailored to refer to finite plans. As a consequence, we can express goals that cannot be expressed with LTL. Some examples follow.

Example 3.1 The following are temporal f-FOLTL goals together with their intuitive meaning.

- $\Diamond(\mathsf{final} \wedge (\exists c)\,(corridor(c) \wedge at(Robot,c)))$: In the final state, $at(Robot,c)$ for some corridor c. This is one way of encoding final-state goals in f-FOLTL.
- $\Box((closed(D_1) \wedge \bigcirc\neg closed(D_1)) \supset \bigcirc\bigcirc\bigcirc closed(D_1))$: If D_1 was closed at plan step i, and then becomes opened at plan step $i+1$, then it must be closed by plan step $i+3$, for every i.
- $(\forall r_1, r_2).\, priorityOver(r_1,r_2) \supset ((\neg delivered(r_2)\,\mathsf{U}\,delivered(r_1)) \wedge \Diamond delivered(r_2))$: If r_1 has priority over r_2 then r_2 must be delivered, but not before r_1.
- $\Diamond(p(a) \wedge \bigcirc\bigcirc\mathsf{final})$: $p(a)$ must hold true two states before the plan ends. This is an example of a goal that cannot be expressed in LTL, since it does not have the final constant.

When writing f-FOLTL goals, one has to be careful not to use formulae that require infinite plans, since they may be reduced to a contradictory formula. Indeed, the algorithm we present in the next section will automatically generate a non-accepting automaton for some of these formulae.

The algorithm we present in the next section generates an automaton that accepts models of f-FOLTL formula expressed in a syntactical form that we call *extended prenex normal form*.

Definition 3.6 (Extended Prenex Normal Form (EPNF)) *A formula is in* extended prenex normal form *(EPNF) if it is of the form* $(Q_1x_1)(Q_2x_2)\cdots(Q_nx_n)\,\varphi$*, where* $Q_i \in \{\forall,\exists\}$ *and all quantifiers that occur in* φ *quantify on first-order, atemporal, subformulae.*

Some formulae that are not in EPNF, have an EPNF equivalent. For example, it can be proven that $(\forall x)\,\Box(P(x) \supset (\exists y)\,\Diamond Q(x,y))$ is equivalent to $(\forall x)\,\Box(P(x) \supset \Diamond(\exists y)\,Q(x,y))$, which is in EPNF. However, there are formulae that do not have an EPNF equivalent, e.g. $\Box\exists x(P(x) \wedge \Diamond Q(x))$.

3.2.2 Planning Instances

To simplify the exposition of the concepts of this chapter, we represent planning instances using causal rules instead of STRIPS or ADL operators. A planning instances is a tuple $\langle \mathcal{I}, \mathcal{D}, \mathcal{G}, \mathcal{T} \rangle$, where $\mathcal{I}$ is the *initial state*, represented as a set of first-order (ground) positive facts; $\mathcal{D}$ is the *domain description*; $\mathcal{G}$ is a temporal formula describing the *goal*, and $\mathcal{T}$ is a (possibly empty) set of *derived predicate* definitions, which are predicates that are defined in terms of other fluents of the domain.

A domain description is a tuple $\mathcal{D} = \langle Objs, \mathcal{C}, \mathcal{R} \rangle$, where $Objs$ is a finite set of objects, $\mathcal{C}$ is a set of *causal rules*, and $\mathcal{R}$ a set of *action precondition rules*. Causal rules correspond to positive and negative *effect axioms* in the Situation Calculus (Pednault, 1989; McCarthy and Hayes, 1969). As such, we remark that any planning instance described in ADL (plus derived predicates) can be described in terms of causal rules (plus derived predicates) and vice versa.

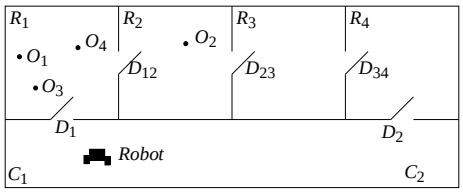


Figure 3.1: The robot domain.

We represent positive and negative causal rules by the triple $\langle a(\vec{x}), c(\vec{x}), f(\vec{x})\rangle$ and $\langle a(\vec{x}), c(\vec{x}), \neg f(\vec{x})\rangle$ respectively, where $a(\vec{x})$ is an *action term*, $f(\vec{x})$ is a *fluent term*, and $c(\vec{x})$ is a first-order formula, each of them with free variables among those in $\vec{x}$. $\langle a(\vec{x}), \Phi(\vec{x}), \ell(\vec{x})\rangle \in \mathcal{C}$ expresses that fluent literal $\ell(\vec{x})$ becomes true after performing action $a(\vec{x})$ in the current state if condition $\Phi(\vec{x})$ holds. As with ADL operators (Pednault, 1989), the condition $c(\vec{x})$, can contain quantified FO subformulae. Finally, we assume that for each action-fluent pair, there exists at most one positive and one negative causal rule in $\mathcal{C}$. Free variables in $\mathcal{C}$ are assumed to be universally quantified. The set $\mathcal{R}$ of action precondition rules consists of tuples $\langle a(\vec{x}), \pi(\vec{x})\rangle$, such that $a(\vec{x})$ is an action term, and $\pi(\vec{x})$ is a first-order condition. Intuitively $\langle a, \pi\rangle \in \mathcal{R}$ means that it is possible to execute a in a state that satisfies condition π. Free variables in $\mathcal{C}$ or $\mathcal{R}$ are assumed to be universally quantified.

Example 3.2 Consider the robot domain defined by Bacchus and Kabanza (1998). In an instance of this domain, depicted in Figure 3.1, there is a robot, some objects and six locations. Four of the locations correspond to rooms ($R_1, \ldots, R_4$), and two of them represent the corridor (C_1 and C_2). Rooms are connected by doors, which can be opened or closed. The robot can move between connected rooms, close or open doors, and grasp or drop objects. It can hold one object at a time. The causal rules for this domain are the following.

Positive	**Negative**
$\langle open(d), \mathsf{true}, opened(d)\rangle$	$\langle close(d), \mathsf{true}, \neg opened(d)\rangle$
$\langle grasp(o), \mathsf{true}, holding(o)\rangle$	$\langle grasp(o), \mathsf{true}, \neg handempty\rangle$
$\langle release(o), \mathsf{true}, handempty\rangle$	$\langle release(o), \mathsf{true}, \neg holding(o)\rangle$
$\langle move(x,y), o = robot \vee holding(o), at(o,y)\rangle$	$\langle move(x,y), o = robot \vee holding(o), \neg at(o,x)\rangle$

3.2.3 Causal Rules for Arbitrary Formulae

The causal rules of a domain describe the dynamics of individual fluents. However, to model an NFA in a planning domain, we must also know the dynamics of arbitrary complex formulae, such as for example, the causal rule for $at(o, R_1) \wedge holding(o)$.

To obtain these rules one can use regression, a well-known technique introduced by Waldinger (1977), and then extended for ADL by Pednault (1989), and further generalized by Reiter (1991). Since

regression is well-studied in the literature and is not central to this chapter, here we only show the form of causal rules for arbitrary formulae. For more details on the correctness of this approach, we refer the reader to Reiter's book (2001) or Pednault's paper (1989).

To characterize the causal rules without articulating them explicitly, we introduce below the relation `causes` that holds over the set of valid causal rules for arbitrary formulae. The definition for `causes` below only includes rules for negation and conjunction since disjunction follows from these. Here, we assume that $\alpha(\vec{x})$ is a boolean formula of fluents with free variables among the vector of variables $\vec{x}$. Furthermore, $\vec{t}$ is a vector of variables or constants.

Definition 3.7 `causes` *is the least set that satisfies the following properties:*

1. ***(base case)*** *If* $\langle a, c, (\neg) f\rangle \in \mathcal{C}$ *then* $\langle a, c, (\neg) f\rangle \in$ `causes`.
2. ***(instantiation & negation)*** *If* $(a(\vec{x}), \Phi^{+}_{a,\alpha}(\vec{x}), \alpha(\vec{x})) \in$ `causes`*, then,*

   (a) $\langle a(\vec{x}), \vec{x} = \vec{t} \wedge \Phi^{+}_{a,\alpha}(\vec{x}), \alpha(\vec{t})\rangle \in$ `causes`*, and*

   (b) $\langle a(\vec{x}), \vec{x} = \vec{t} \wedge \Phi^{+}_{a,\alpha}(\vec{x}), \neg\neg\alpha(\vec{t})\rangle \in$ `causes`.

3. ***(conjunction)*** *If the following causal rules are in* `causes`*:*

   $$\langle a(\vec{x}), \Phi^{+}_{a,\alpha}(\vec{x}), \alpha(\vec{t_1})\rangle, \qquad \langle a(\vec{x}), \Phi^{-}_{a,\alpha}(\vec{x}), \neg\alpha(\vec{t_1})\rangle,$$
   $$\langle a(\vec{x}), \Phi^{+}_{a,\beta}(\vec{x}), \beta(\vec{t_2})\rangle, \qquad \langle a(\vec{x}), \Phi^{-}_{a,\beta}(\vec{x}), \neg\beta(\vec{t_2})\rangle$$

   then the following are also in `causes`*:*

   (a) $\langle a(\vec{x}), \Phi^{+}_{a,\alpha\wedge\beta}, \alpha(\vec{t_1}) \wedge \beta(\vec{t_2})\rangle \in$ `causes`*, where*

   $$\Phi^{+}_{a,\alpha\wedge\beta} = \left(\Phi^{+}_{a,\alpha}(\vec{x}) \wedge \Phi^{+}_{a,\beta}(\vec{x})\right) \vee \left(\alpha(\vec{x}) \wedge \neg\Phi^{-}_{a,\alpha}(\vec{x}) \wedge \Phi^{+}_{a,\beta}(\vec{x})\right) \vee \left(\beta(\vec{x}) \wedge \neg\Phi^{-}_{a,\beta}(\vec{x}) \wedge \Phi^{+}_{a,\alpha}(\vec{x})\right)$$

   (b) $\langle a(\vec{x}), \Phi^{-}_{a,\alpha\wedge\beta}, \neg(\alpha(\vec{t_1}) \wedge \beta(\vec{t_2}))\rangle$*, where* $\Phi^{-}_{a,\alpha\wedge\beta} = \Phi^{-}_{a,\alpha}(\vec{x}) \vee \Phi^{-}_{a,\beta}(\vec{x})$.

Rules such as this will be extensively used to produce the translated domain. The downside of this approach is its space complexity; the size of the conditions in the causal laws can grow exponentially, as it is shown by the following proposition.

Proposition 3.2 *Let* φ *be a an atemporal formula with n binary boolean connectives such that all its atomic sub-formulae are fluent ground terms. Moreover, let* F *be the set of atomic sub-formulae of* φ*.*

Then, assuming no simplifications are made, the aggregated size of the causal rules for $(\neg)\varphi$ *is* $\Omega(4^n m)$*, where m is the size of the smallest causal rule among all fluents* $f \in F$*.*

Proof: Straightforward by solving a recursive equation for a lower bound on the size of the causal rules for φ. ■

For this reason we will also provide a more efficient translation, based on derived predicates. We will introduce this translation in the following chapters.

3.3 From f-FOLTL to Parameterized NFA

It is a well known fact that for every LTL formula φ, there exists a Büchi automaton[3] A_φ that accepts an infinite state sequence σ if and only if $\sigma \models \varphi$ (Vardi and Wolper, 1994). In this section, we provide an algorithm for the construction of parameterized finite state automata (PNFA) that accept the models of f-FOLTL formulae in EPNF. This translation step is essential to converting TEGs into standard, final-state goals.

The rest of the section starts by introducing parameterized automata. Then it describes an algorithm that accepts models of f-FOLTL formulae, and establishes it correctness. Finally, it comments on how these automata can be simplified.

3.3.1 Parameterized Finite-State Automata

Parameterized finite-state automata represent families of finite-state automata. The input to these automata are models of f-FOLTL formulae, which are either rejected or accepted. The first automaton we utilize is the parameterized, state-labeled, finite-state automaton (PSLNFA). A PSLNFA is like a nondeterministic finite-state automaton (NFA) but with two main differences. The first difference, is that its states are labeled with first-order formulae. Intuitively, whenever the automaton is in state q labeled $L(q)$, it checks that all formulae in $L(q)$ are true in the interpretation that is at the beginning of its input. The second main difference is that PSLNFA are parameterized, which means that its acceptance condition can be affected by a set of parameters. The parameters are variables that may occur free in the labels of the states. A formal definition of a PSLNFA follows.

Definition 3.8 (PSLNFA) *A parameterized state-labeled NFA (PSLNFA) is a tuple*

$$A = \langle Q, \Sigma(S,\mathcal{D}), \delta, L, \Gamma, \vec{x}, Q_0, F \rangle,$$

where Q is a finite set of states, and $Q_0 \subseteq Q$ *is a set of initial states. The alphabet* $\Sigma(S,\mathcal{D})$ *is a set of first-order* $\mathcal{L}_{FO}(S)$*-interpretations over the same domain* $\mathcal{D}$*, such that they assign the same denotation*

[3] A Büchi automaton is an extension of a finite state automaton to infinite inputs.

to all function symbols in S; $\delta \subseteq Q \times Q$ is a transition relation; $F \subseteq Q$ is the set of final states, $\vec{x}$ is a string of variables; $\Gamma \in \{\forall, \exists\}^*$ is a string of quantifiers such that $|\Gamma| = |\vec{x}|$, and the labeling function $L : Q \to 2^{\mathcal{L}_{FO}(S)}$ is such that if $\varphi \in L(q)$ then all the free variables of φ are in $\vec{x}$.

A PSLNFA with no quantifiers accepts a string of interpretations $s_0 \dots s_n$ iff there is a path $q_0 \dots q_n$ from an initial automaton state to a final automaton state such that labels of the states traversed are true in the corresponding interpretation (i.e., all formulas in $L(q_i)$ are true in s_i). When adding quantifiers, the free variables in the labels are interpreted based on the quantifiers in Γ. For example, if a PSLNFA contains a single parameter x, and its quantifier is a $\forall$, then the PSLNFA accepts $s_0 \dots s_n$ if for all ways of interpreting x, there is a path to an accepting state with the condition above.

To give a formal definition of the language accepted by a PSLNFA we define a few more concepts. For any PSLNFA $A = \langle Q, \Sigma(S, \mathcal{D}), \delta, L, \Gamma, \vec{x}, Q_0, F \rangle$, we define an automaton augmented with function ν that maps variables of the language to elements of the domain $\mathcal{D}$. This augmented automaton is denoted by $A \cdot \nu$, and formally corresponds to a tuple that contains ν as a new element.

Definition 3.9 (Run of a Quantifier-Free, Augmented PSLNFA) *A* run *of an augmented automaton with no quantifiers $A = \langle Q, \Sigma(S, \mathcal{D}), \delta, L, \varepsilon, \varepsilon, Q_0, F, \nu \rangle$ over a string $s_0 s_1 \cdots s_n \in \Sigma(S, D)^*$ is a string of automaton states $\rho = q_0 q_1 \cdots q_n$ such that $(q_j, q_{j+1}) \in \delta$, and $\langle s_i, \nu \rangle \models L(q_i)$, for all $i \in \{0, 1, \dots, n\}$, and all $j \in \{0, 1, \dots, n-1\}$.*

A string will be accepted by an augmented automaton with no quantifiers if there is a run for it that ends in a final state. Formally,

Definition 3.10 (Strings Accepted by a Quantifier-Free Augmented PSLNFA) *A string of interpretations $\sigma \in \Sigma(S, D)^*$ is* accepted *by an augmented automaton with no quantifiers A iff there exists a run of A, $\rho = q_0 q_1 \cdots q_n$ on σ, such that $q_n \in F$.*

Now we are ready to define when a string is accepted by a regular augmented PSLNFA. The acceptance condition is strongly related to the definition of truth of a first-order formula. Intuitively, for PSLNFA $A \cdot \nu$ with an initial quantifier $\forall$ (respectively $\exists$), we will say that a string σ is accepted iff all (respectively, some) of the augmented automaton $A \cdot \nu'$ accepts the string, where ν' extends ν with an assignment for a new variable. Formally,

Definition 3.11 (String Accepted by a PSLNFA) *String $\sigma \in \Sigma(S, D)^*$ is* accepted *by an augmented PSLNFA with quantifiers $A = \langle Q, \Sigma(S, \mathcal{D}), \delta, L, V\Gamma, x\vec{x}, Q_0, F, \nu \rangle$, where V is either $\forall$ or $\exists$ and x is a variable, iff when $V = \forall$ (respectively, when $V = \exists$) σ is accepted by automaton*

$$A' = \langle Q, \Sigma(S, \mathcal{D}), \delta, L, \Gamma, \vec{x}, Q_0, F, \nu[x \to a] \rangle,$$

for all (respectively, for some) $a \in \mathcal{D}$.

Finally, we are now able to define the language accepted by a PSLNFA.

Definition 3.12 (Language Accepted by a PSLNFA) *A PSLNFA* $A = \langle Q, \Sigma(S,\mathcal{D}), \delta, L, \Gamma, \vec{x}, Q_0, F\rangle$ *accepts the set of all* $\sigma \in \Sigma(S,D)^*$*, such that* σ *is accepted by all augmented automatons* $A \cdot \nu$*, for every* ν *that assigns elements in* $\mathcal{D}$ *to variables in* $\vec{x}$*.*

Example Consider the PSLNFA $A = \langle\{q_0,q_1\},\Sigma,\delta,L,\Gamma,xy,\{q_0\},\{q_1\}\rangle$, where $\delta = \{(q_0,q_1),(q_1,q_1)\}$, and $L(q_0) = P(x)$ and $L(q_1) = Q(x,y)$. A accepts the models of $(\forall x).(\forall y).P(x) \wedge \bigcirc\Box Q(x,y)$ if $\Gamma = \forall\forall$ and accepts the models of $(\forall x).(\exists y).P(x) \wedge \bigcirc\Box Q(x,y)$ in case $\Gamma = \forall\exists$. Figure 3.2 shows a graphical representation of a PSLNFA that can accept the models of either $(\forall x)\Diamond P(x)$ or $(\exists x)\Diamond P(x)$.

3.3.2 The algorithm

The translation algorithm is a modification of the one proposed by Gerth, Peled, Vardi, and Wolper (1995). In contrast to their algorithm, ours generates a PSLNFA instead of a Büchi automaton.

To represent a node of the automaton, the algorithm uses Gerth *et al.*'s data structure *Node*, which is a tuple $\langle Name, Incoming, New, Old, Next\rangle$. The field *Name* contains the name of the node; *Incoming* is the list of node names with an outgoing edge leading to *Node*; *New* contains first-order formulae that must hold at the current state but that have not been processed by the algorithm; *Old* contains the formulae that must hold in the nodes that have been processed by the algorithm; *Next* contains temporal formulae that have to be true in the immediate successors of *Node*.

In the following, suppose we want to build a PSLNFA for sentence φ in EPNF. We denote the string of quantifiers and variables at the beginning of φ by $\mathsf{QPrefix}(\varphi)$. To generate the PSLNFA, we strip $\mathsf{QPrefix}(\varphi)$ from φ and then leave the formula just in terms of the temporal operators U and R, and the binary boolean operators $\wedge$ and $\vee$. We then push all $\neg$'s inside such that they occur only in front of first-order formulae. The resulting formula, say, φ' is the input for the procedure we describe below. Note that the construction will start with a single node that contains φ' in its *New* field.

When processing node N, the algorithm checks whether there are pending formulae in *New*. If there are none, then the node can be added to the *NodeSet*. Two cases can hold:

1. If there is already a node in *NodeSet* with the same fields *Old* and *Next*, then its *Incoming* list is updated by adding those nodes in N's incoming list. (Line 4).

2. If there is no such node, then N is added to *NodeSet*. Then, a new node is created for processing if $\mathsf{final} \notin Old$. This node contains N in its incoming list, and the field *New* set to N's *Next* field. The fields *Next* and *Old* of the new node are empty. (Lines 5–12).

   Intuitively, in this case we are creating a new node, successor to the current node, intended to verify the formulae in the *Next* set. Notice that the new node will only be created if $\mathsf{final} \notin Old$, since that is the only case in which a node can have a successor.

Otherwise, if *New* is not empty, formula η is removed from *New* and added to *Old*. Then,

1. In case η is a literal, or of the form $(\forall x)\,\phi(x)$, or $(\exists x)\,\phi(x)$, then if $\neg\eta$ is in *Old*, the node is discarded (a contradiction has occurred). Otherwise, η is added to *Old* and the node continues to be processed.

2. Otherwise:

   (a) If $\eta = \varphi \wedge \psi$, both φ, and ψ are added to *New*.

   (b) If $\eta = \bigcirc\psi$, then ψ is added to *Next*.

   (c) If η is one of $\varphi \vee \psi$, $\varphi \mathsf{U} \psi$, or $\varphi \mathsf{R} \psi$, then N is split into two nodes. The set $\texttt{New1}(\eta)$ and $\texttt{New2}(\eta)$ are added, respectively, to the *New* field of the first and second nodes. These functions are defined as follows:

η	$\texttt{New1}(\eta)$	$\texttt{New2}(\eta)$
$\varphi \vee \psi$	$\{\varphi\}$	$\{\psi\}$
$\varphi \mathsf{U} \psi$	$\{\varphi, \bigcirc(\varphi \mathsf{U} \psi)\}$	$\{\psi\}$
$\varphi \mathsf{R} \psi$	$\{\psi, \mathsf{final} \vee \bigcirc(\varphi \mathsf{R} \psi)\}$	$\{\varphi, \psi\}$

   The intuition of the split lies in standard f-FOLTL equivalences. For example, $\varphi \mathsf{U} \psi$ is equivalent to $\psi \vee (\varphi \wedge \bigcirc(\varphi \mathsf{U} \psi))$, thus one node verifies the condition ψ, whereas the other verifies $\varphi \wedge \bigcirc(\varphi \mathsf{U} \psi)$.

Definition 3.13 ($\Delta^-(q)$) *Let $\Delta(q)$ be the value of the Old field for node q, when node q has been processed. We define $\Delta^-(q)$ as the set containing all the literals in $\Delta(q)$ or formulae of the form $(Qx)\,\varphi$, where φ is a first-order (atemporal) formula.*

For an EPNF formula φ, we define PSLNFA $A_\varphi = \langle Q, \Sigma(S, \mathcal{D}), \delta, L, \Gamma, \vec{x}, Q_0, F \rangle$, where

- $Q = \{n \,|\, n \in \mathit{NodeSet}\}$,
- $Q_0 = \{q \in Q \,|\, \mathit{Init} \in \mathit{Incoming}(q)\}$.
- $\vec{x}$ is the maximal subsequence of variables in $\mathsf{QPrefix}(\varphi)$, and Γ is the maximal subsequence of quantifiers in $\mathsf{QPrefix}(\varphi)$
- δ is such that $\delta(q, q')$ iff q and q' are connected in the graph (i.e., $q \in \mathit{Incoming}(q')$).
- $F = \{q \in Q \,|\, \mathit{Next}(q) = \emptyset \text{ and } \neg\mathsf{final} \notin \Delta^-(q)\}$.
- $L(q)$ is equal to $\Delta^-(q) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$.

Algorithm 3.1 Converts an f-FOLTL formula φ into a graph used to define A_φ.

function EXPAND(*Node*,*NodeSet*)
  if $New(Node) = \emptyset$ **then**
    if $\exists N \in NodeSet$ and $Old(N) = Old(Node)$ and $Next(N) = Next(Node)$ **then**
      $Incoming(N) \leftarrow Incoming(N) \cup Incoming(Node)$
      return *NodeSet*
    else if final $\notin Old(Node)$ **then**
      return EXPAND($[Name \leftarrow Father \leftarrow$ newname(),
          $Incoming \leftarrow Name(Node)$,
          $New \leftarrow Next(Node), Old \leftarrow \emptyset$
          $Next \leftarrow \emptyset], \{Node\} \cup NodeSet$)
    else if $Next(Node) = \emptyset$ **then return** $\{Node\} \cup NodeSet$
    else return *NodeSet* ▷ *Node* is discarded
    end if
  else
    choose $\eta \in New(Node)$
    $New(Node) \leftarrow New(Node) \setminus \{\eta\}$
    if $\eta \neq True$ and $\eta \neq False$ **then**
      $Old(Node) \leftarrow Old(Node) \cup \{\eta\}$
    end if
    if η is a literal, $(Qx)\,\varphi$, *True* or *False* **then**
      if $\eta = False$ or $\neg\eta \in Old(Node)$ **then**
        return $(NodeSet)$ ▷ *Node* is discarded
      else
        return EXPAND$(Node, NodeSet)$
      end if
    else if $\eta = \bigcirc\varphi$ **then**
      $Next(Node) \leftarrow Next(Node) \cup \{\varphi\}$
      return EXPAND$(Node, NodeSet)$
    else if $\eta = \varphi \wedge \psi$ **then**
      $New(Node) \leftarrow New(Node) \cup (\{\varphi, \psi\} \setminus Old(Node))$
      return EXPAND$(Node, NodeSet)$
    else if $\eta = \varphi \vee \psi$ or $\varphi \mathsf{R} \psi$ or $\varphi \mathsf{U} \psi$ **then**
      $Node1 \leftarrow$ SplitNode$(Node, \texttt{New1}(\eta))$
      $Node2 \leftarrow$ SplitNode$(Node, \texttt{New2}(\eta))$
      return EXPAND(Node2,EXPAND(Node1,NodeSet))
    end if
  end if
end function
function SPLITNODE(*Node*,ϕ)
  $NewNode \leftarrow [Name \leftarrow$ newname(), $Father \leftarrow Name(Node)$
  $Incoming \leftarrow Incoming(Node), New \leftarrow New(Node) \cup \phi, \quad Old \leftarrow Old(Node), Next \leftarrow Next(Node)]$
  return *NewNode*
end function
function GENGRAPH(φ)
  EXPAND$([Name \leftarrow Father \leftarrow$ newname(), $Incoming \leftarrow \{Init\}, New \leftarrow \{\varphi\}, Old \leftarrow \emptyset], \emptyset)$
end function

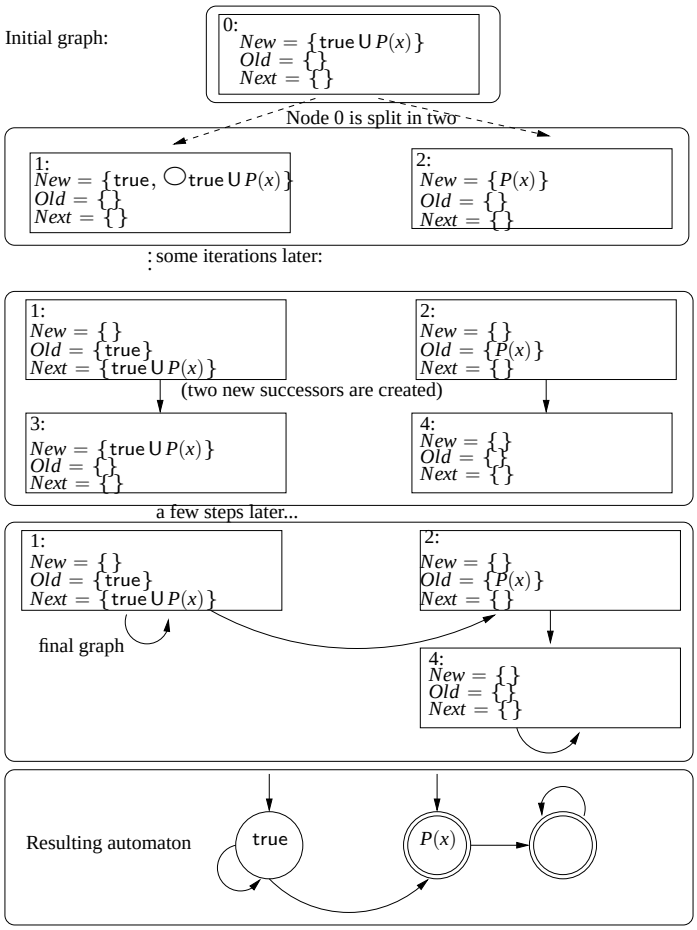


Figure 3.2: Algorithm execution for formula $(\forall x)\Diamond P(x)$.

Figure 3.2 shows an example of the generation of a PSLNFA for $(\forall x)\Diamond A(x)$.

This theorem states the correctness of the algorithm.

Theorem 3.1 *Let A_ψ be the automaton constructed by our algorithm from an f-*FOLTL *formula ψ in EPNF. Then A_ψ accepts exactly the models of ψ.*

Proof: See Section A.2 (page 163). ■

An immediate consequence of this theorem is that our algorithm generates non-accepting automata for temporal formulae that are only satisfied by infinite models, and that thus are unsatisfiable f-FOLTL formulae. Sometimes this would be reflected by the fact that the automaton does not have accepting states at all (this happens for $\varphi \wedge \Box(\varphi \supset \bigcirc\psi) \wedge \Box(\psi \supset \bigcirc\varphi)$, or by the fact that some state visited by all paths to an accepting state is labeled with an inconsistent first-order formulae (this is the case of $\Diamond P(a) \wedge (\forall x)\Box(P(x) \supset \bigcirc P(x))$). In the former case we are able to recognize that the goal is intrinsically

unachievable by just looking at the automaton, whereas in the latter we cannot do it in general, since checking whether the labeling formulae are consistent is undecidable.

Simplifying PSLNFAs into PNFAs

The algorithm presented above often produces automata that are much bigger than the optimal. To simplify the automata, we have used a modification of the algorithm proposed by Etessami and Holzmann (2000). This algorithm uses a simulation technique to simplify the automaton. In experiments conducted by Fritz (2003), it was shown to be slightly better than LTL2AUT (Daniele, Giunchiglia, and Vardi, 1999) at simplifying Büchi automata.

To apply the algorithm directly, we need an automaton representation in which transitions rather than states are labelled with formulae. To that end, we introduce parameterized NFAs (PNFAs). Intuitively, a PNFA is like a PSLNFA but such that *transitions*—not states—are labeled with first-order formulae. Formally, a PNFA is a tuple $A = \langle Q, \Sigma(S,\mathcal{D}), \delta, \Gamma, \vec{x}, Q_0, F\rangle$, where Q, Q_0, F, Γ, $\vec{x}$, and $\Sigma(S,\mathcal{D})$ are defined as in PSLNFAs. Finally, the labeled transition relation δ is a subset of $Q \times 2^{\mathcal{L}_{FO}(S)} \times Q$.

As before, given an assignment of variables to domain variables ν, we can define an augmented version of A denoted by $A \cdot \nu$. A run of $A \cdot \nu$ over the string of states $\sigma = s_1 \cdots s_n \in \Sigma(S,\mathcal{D})^*$ is a sequence $\rho = q_0 q_1 \cdots q_n$ where $q_0 \in Q_0$, and for some label L such that $(q_i, L, q_{i+1}) \in \delta$, $\langle s_{i+1}, \nu\rangle \models L$, for all $i \in \{0,\ldots,n-1\}$. Run ρ is *accepting* if $q_n \in F$. Finally, the acceptance for PNFAs is defined analogously to that of PSLNFAs, and therefore we omit it here.

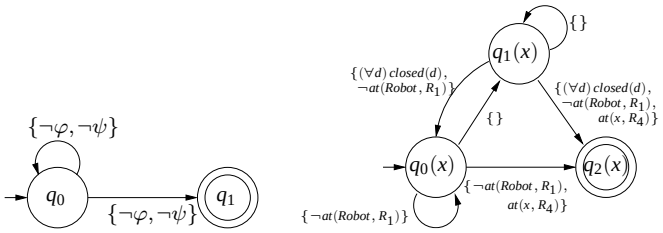
It is straightforward to convert a PSLNFA to an equivalent PNFA by adding one initial state and copying labels of states to any incoming transition. Figure 3.3 shows examples of PNFAs generated by our implementation for some f-FOLTL formulae. The automaton for formula (b) is parameterized on variable *x*, which is indicated beside the state name.

Size complexity of the NFA

In theory, the resulting automaton can be exponential in the size of formula in the worst case. Simplifications reduce the number of states of the PNFA significantly.

Proposition 3.3 *Let φ be in negated normal form, then the number of states of A_φ is $2^{O(|\varphi|)}$.*

Proof: Note that the Algorithm 3.1 generates a new node if there is no previously existing node with identical *Old* and *Next* fields. In the worst case it will generate all plausible nodes, which is bound by the total number of possible combinations for *Old* and *Next*. This number is upper-bounded by $2^{sub(\varphi)} \times 2^{sub(\varphi)}$, where $sub(\varphi)$ is the number of subformulae of φ. The proof is concluded by observing that $sub(\varphi) = O(|\varphi|)$. ■



(a) (b)

Figure 3.3: Simplified PNFA (a) $\Box(\varphi \supset \bigcirc\psi) \wedge \Box(\psi \supset \bigcirc\varphi)$, and (b) $\Box(at(Robot,R_1) \supset \bigcirc\Diamond(\forall d)\,closed(d)) \wedge (\forall x)\Diamond\Box at(x,R_4)$.

Unfortunately, the upper bound above is tight. There are simple cases where our proposed translation blows up.

Proposition 3.4 *Any PNFA for formula $\Diamond p_1 \wedge \Diamond p_2 \wedge \ldots \wedge \Diamond p_n$, where $p_1, p_2, \ldots, p_n$ are propositions, has at least 2^n states.*

Proof: See Section A.3 (page 166). ■

Intuitively, each state of the PNFA for the above mentioned formula keeps track of a particular combination of propositions that has been true in the input read so far. Nevertheless, in Section 3.4.3 we describe techniques that will *not* blow up the planning domain when transforming formulae like the one in Proposition 3.4. Also, it is critical to note that in practice, the number of states of NFAs for natural goals were generally equivalent to the size of our formulae (see Section 3.5).

3.4 Compiling PNFAs into a Planning Instance

We are now ready to show how the PNFA can be encoded in a planning instance. This will be essential to transform TEGs into classical final-state goals.

During the execution a plan $a_1 a_2 \cdots a_n$, a set of planning states $\sigma = s_0 s_1 \cdots s_n$ is generated. In what follows we make no distinction between a planning state (which are sets of ground first-order facts) and a first-order interpretation. Thus, if s is a planning state, we say that an atomic ground fact $P(\vec{c})$ is true in s (i.e., $s \models P(\vec{c})$) if and only if $P(\vec{c}) \in s$. This definition extends trivially to non-atomic formulae.

In the planning domain, each state of the automaton is represented by a fluent. More formally, for each state q of the automaton A we add to the domain a new fluent $E_q(\vec{x})$, where $\vec{x}$ is the vector of variables used in the definition of A. The translation is such that if a sequence of actions $a_1 a_2 \cdots a_n$ is performed in state s_0, generating the succession of states $\sigma = s_0 s_1 \cdots s_n$, then $E_q(\vec{c})$ is true in s_n, for a vector of constants $\vec{c}$, if and only if there is a run ρ of $A_\varphi \cdot \nu$ on σ that ends in state q, where ν assigns the variables in $\vec{x}$ to constants $\vec{c}$.

Once the PNFA is modelled within the domain, the temporal goal in the newly generated domain is reduced to a property of the final state alone. Intuitively, this property corresponds to the accepting condition of the automaton.

To represent the dynamics of the states of the automaton, there are two alternatives. The first is to modify the domain's *causal rules* to give an account of their change. The second, is to define them as *derived predicates* or *axioms*. The derived predicates approach we introduce here prove to be more efficient, both in theory and in practice.

Henceforth, we assume the following:

- We start with a planning instance $\langle \mathcal{I}, \mathcal{D}, \mathcal{G}, \mathcal{T} \rangle$, where $\mathcal{G}$ is a temporal formula in f-FOLTL.
- Temporal goal $\mathcal{G}$ is translated to the PNFA $A_{\mathcal{G}} = (Q, \Sigma, \delta, \Gamma, \vec{x}, Q_0, F)$, with $\Gamma = V_1 \cdots V_n$ and $\vec{x} = x_1 \cdots x_n$.
- To simplify notation, we denote by $\texttt{pred}(q)$ the set of predecessors of q. E.g., in Fig. 3.3(b), $\texttt{pred}(q_0) = \{q_0, q_1\}$.
- We define $\lambda_{p,q}(\vec{x})$ as the formula $\bigvee_{(q,L,p)\in\delta} \bigwedge L$. E.g., in Fig. 3.3(b), $\lambda_{q_1,q_0} = (\forall d)\, closed(d) \wedge \neg at(Robot, R_1)$. Note that $\vec{x}$ corresponds to the variable vector in $A_{\mathcal{G}}$, and therefore $\vec{x}$ are all the variables that may appear free in $\lambda_{p,q}(\vec{x})$.
- For first-order formulae φ, we denote its grounded version by $\mathsf{ground}(\varphi)$. This formula is equivalent to φ, but that has no quantifiers. Note that it is possible to compute this formula since planning domains we are dealing with have a finite number of objects. ground is defined as follows,

$$\mathsf{ground}(\varphi) \stackrel{\text{def}}{=} \begin{cases} \varphi & \text{if } \varphi \text{ is an atomic proposition} \\ \neg\mathsf{ground}(\psi) & \text{if } \varphi = \neg\psi \\ \mathsf{ground}(\psi) \wedge \mathsf{ground}(\chi) & \text{if } \varphi = \psi \wedge \chi \\ \bigwedge_{a \in Objs} \mathsf{ground}(\psi(a)) & \text{if } \varphi = (\forall x)\, \psi(x) \end{cases}$$

3.4.1 Translating PNFA to Causal Rules

Recall that we have translated our TEG into a PNFA and to encode this PNFA in the planning domain, we have introduced fluents E_q, one for each state q of the automaton. The final step is defining the dynamics of this domain. We propose two methods to do that.

In the first translation, we encode the dynamics of the fluent E_q as causal rules. For each fluent E_q we generate a new set of causal rules. The resulting new rules are added to the set $\mathcal{C}'$, which is initialized to $\emptyset$. In the second, we define the dynamics of E_q axiomatically, through the so-called PDDL derived predicates.

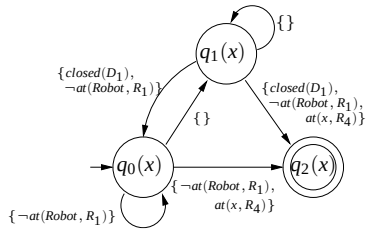


Figure 3.4: A PNFA for $\Box(at(Robot,R_1) \supset \bigcirc\Diamond closed(D_1)) \wedge (\forall x)\Diamond\Box at(x,R_4)$.

We later show that the second translation is much more efficient than the first. The introduction of the first one is justified for a pragmatic reason, since not many state-of-the-art planners are able to handle derived predicates.

New Causal Rules

To understand the intuition behind the translation, consider the NFA shown in Figure 3.3(b). Suppose E_{q_2} is false in a state s_i. After performing action a_i, fluent E_{q_2} must become true in the resulting state, s_{i+1}, iff either E_{q_0} was true in s_i and $\neg at(Robot,R_1) \wedge at(O_1,R_4)$ is true in s_{i+1} or E_{q_1} was true in s_i and $\neg at(Robot,R_1) \wedge (\forall d)\, closed(d) \wedge at(O_1,R_4)$ is true in s_{i+1}. Note that $\neg at(Robot,R_1) \wedge (\forall d)\, closed(d) \wedge at(O_1,R_4)$ can be true in s_{i+1} because a made the property true, or because it was true in s_i and a did not make it false.

To write the positive causal rule for $E_{q_2}(\vec{x})$ on action a, we must only refer to the state prior to the execution of a. To do so, we appeal to regression. For each action a, the positive action rule $\langle a, \Phi^+_{a,E_q}(\vec{x}), E_q(\vec{x})\rangle$ is added to $\mathcal{C}'$, where $\Phi^+_{a,E_q}(\vec{x})$ stands for:

$$\bigvee_{p \in Pred(q)\setminus\{q\}} E_p(\vec{x}) \wedge (\Phi^+_{a,\mathsf{ground}(\lambda_{p,q}(\vec{x}))} \vee \lambda_{p,q}(\vec{x}) \wedge \neg\Phi^-_{a,\mathsf{ground}(\lambda_{p,q}(\vec{x}))}))). \tag{3.1}$$

Note that $\Phi_{a,\mathsf{ground}(\lambda_{p,q})}$ is a condition obtained by regression.

For the negative case, consider state q_0 of the automaton. If E_{q_0} is true in some state s_i, then when getting to state s_{i+1} after performing a, it will become false if $\neg at(Robot,R_1)$ holds in s_{i+1} *and* it does not happen that in E_{q_1} is true in s_i and $\neg at(Robot,R_1) \wedge closed(D_1)$ is true in s_{i+1}.

Again, we need to appeal to regression. For each action a, the positive action rule $\langle a, \Phi^-_{a,E_q}, \neg E_q\rangle$ is added to $\mathcal{C}'$, where $\Phi^-_{a,E_q}(\vec{x})$ stands for:

$$\neg\Phi^+_{a,E_q}(\vec{x}) \wedge \neg(\Phi^+_{a,\mathsf{ground}(\lambda_{q,q}(\vec{x}))} \vee \lambda_{q,q}(\vec{x}) \wedge \neg\Phi^-_{a,\mathsf{ground}(\lambda_{q,q}(\vec{x}))}). \tag{3.2}$$

Note that $\lambda_{q,q}(\vec{x})$ is false if there is no self transition in q.

Example 3.2 (cont.) In the robots domain, consider the automaton constructed for formula $\Box(at(Robot,R_1) \supset \bigcirc\Diamond closed(D_1)) \wedge (\forall x)\Diamond\Box at(x,R_4)$ shown in Figure 3.4. We would add the following positive causal

rule for fluent E_{q_2} and action $close(x)$.

$$\langle close(x), E_{q_1}(x) \wedge [at(O_1,R_4) \wedge \neg at(Robot,R_1) \wedge closed(D_1) \vee$$
$$at(O_1,R_4) \wedge \neg at(Robot,R_1) \wedge x = D_1)] \vee$$
$$E_{q_0}(x) \wedge at(O_1,R_4) \wedge \neg at(Robot,R_1), E_{q_2}(x)\rangle$$

New Initial State

The original initial state must also be modified, since it now must include which fluents E_q are initially true. The new set of facts $\mathcal{I}'$ is the following:

$$\mathcal{I}' = \{E_q(c_1,\ldots,c_n) \,|\, (c_1,\ldots,c_n) \in Objs^n, p \in Q_0, \text{and for some } L,\ q,\ (p,L,q) \in \delta, \mathcal{I} \models \lambda_{p,q}(c_1,\ldots,c_n)\}.$$

I.e., are the facts $E_q(c_1,\ldots,c_n)$ such that q is reachable for some initial state p through a transition whose label is a fact that is true in $\mathcal{I}$.

New Goal & Planning Instance

Intuitively, the automaton $A_\mathcal{G}$ accepts iff the temporally extended goal $\mathcal{G}$ is satisfied. Therefore, the new goal, $\mathcal{G}' = \mathsf{QPrefix}(\mathcal{G})$. $\bigvee_{p\in F} E_p(x_1,\ldots,x_n)$, is defined according to the acceptance condition of the NFA, i.e. the goal is achieved if $A_\mathcal{G}$ is in some final state. Note that $\mathcal{G}'$ is a non-temporal goal.

The final planning instance L' is $\langle \mathcal{I} \cup \mathcal{I}', \mathcal{C} \cup \mathcal{C}', \mathcal{R}, \mathcal{G}', \mathcal{T}\rangle$.

Size Complexity

Since we have generated a standard planning instance, the complexity of decision problem associated is still PSPACE-hard. However, we the size of the new problem is worst-case exponential in the size of the original problem. This is stated by the following proposition.

Proposition 3.5 *The size of $\mathcal{C}'$ is wost-case $n|Q|2^{O(\ell)}$ where ℓ is the maximum size of a grounded transition in $A_\mathcal{G}$, and n is the number of action terms in the domain.*

Proof: For each of the $|Q|$ predicates we need n new rules. From Proposition 3.2, each of them is worst-case exponential on the size of the (grounded) transition formula. ■

3.4.2 Translation to Derived Predicates (axioms)

In this translation we propose to write a derived predicate definition for $E_q(\vec{x})$. However, as we saw previously, the truth value of $E_q(\vec{x})$ in s_{i+1} depends on whether some fluents $E_p(\vec{x})$ hold true in the previous state, where p is a state of the automaton. Therefore, we need a way to represent in state s_{i+1} what fluents E_p were true in the previous state.

Thus, for each state q of the automaton we use an auxiliary fluent $Prev_q(\vec{x})$ which is true in a plan state s iff E_q was true in the previous state. The dynamics of fluent $Prev_q(\vec{x})$ is described by the following causal rules, which are added to $\mathcal{C}'$:

$$\langle a, E_q(\vec{x}), Prev_q(\vec{x})\rangle, \qquad \langle a, \neg E_q(\vec{x}), \neg Prev_q(\vec{x})\rangle,$$

for each action a. The following definitions are also added to $\mathcal{T}'$:

$$E_q(\vec{x}) \stackrel{\text{def}}{=} \bigvee_{p \in \texttt{pred}(q)} Prev_p(\vec{x}) \wedge \lambda_{p,q}(\vec{x}),$$

New Initial State

The new initial state must specify which fluents of the form $Prev_q$ are true. These are precisely those facts that correspond to the initial state of the automaton.

$$\mathcal{I}' = \{Prev_q(c_1,\ldots,c_n) \,|\, q \in Q_0, (c_1,\ldots,c_n) \in Objs^n\}.$$

New Goal & Planning Instance

The new goal is defined by $\mathcal{G}' = (V_1 x_1)\cdots(V_n x_n) \bigvee_{p\in F} E_p$, and the new planning instance is $\langle \mathcal{I}\cup\mathcal{I}', \mathcal{C}\cup\mathcal{C}', \mathcal{R}, \mathcal{G}', \mathcal{T}\cup\mathcal{T}'\rangle$.

Size Complexity

Planning with the new translated theory is theoretically as hard as planning with the original theory. The amount of additional effort required to update newly created fluents is reflected in the size of $\mathcal{T}'$.

Proposition 3.6 *The size of $\mathcal{T}'$ is $O(n|Q|\ell)$ where ℓ is the maximum size of a transition in $A_{\mathcal{G}}$, and n is the number of action terms in the domain. The size of $\mathcal{C}'$ is only $O(n|Q|)$.*

3.4.3 Avoiding Blowups: Multiple Goals and Formula Splitting

In the previous section we saw that the size of the resulting translation depends on the number of states in the automaton, $|Q|$, and, in the case of using the regression approach, it is worst-case exponential in the size of the transitions. Previously, we also saw that $|Q|$ is worst-case exponential in the size of the temporal formula. This means that we could be generating quite big translations even if we choose to use derived predicates.

Below we present two techniques that will reduce the size of the translation. The first one aims at reducing total number of states of the automata, while the seconds aims at reducing the size of the transitions, an issue that is critical when using regression. These techniques are not guaranteed to always reduce significantly the size of the resulting translation.

Multiple Goals

Fortunately, there is a way to sometimes reduce this size complexity by regarding a formula as specifying a goal composed of multiple individual goals. Consider for example the formula $\varphi = \Diamond p_1 \wedge \ldots \wedge \Diamond p_n$, which we know has an exponential NFA. We know that φ will be satisfied if each of the conjuncts $\Diamond p_i$ is satisfied. If instead of generating a unique NFA for φ we generated a *different* NFA *for each* $\Diamond p_i$, then we could just plan for a goal equivalent to the conjunction of the acceptance conditions of each of those automata. For this particular φ this means that the number of states in the new planning instance is linear in n instead of exponential.

If the TEG, without its quantifier prefix, corresponds to a formula in which the top-level operators are boolean, then we consider each of the (temporal) subformulae as an independent subgoal, and therefore we build an automaton for each of them.

Formally, let φ be a TEG with its quantifier prefix removed. Let function $\Upsilon(\varphi) = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ correspond to the set of all subformulae of φ whose top-level operator is a temporal one, and such that they are maximal under subformulae inclusion (i.e., no pair of different elements φ_i and φ_j in $\Upsilon(\varphi)$ are subformulae of each other). For each of the $\varphi_i \in \Upsilon(\varphi)$ we construct a PNFA, and compute its accepting condition, G_i, expressed in terms of its accepting predicates. The final (classical) goal, corresponds to a formula like φ, but in which all $\varphi_i \in \Upsilon(\varphi)$ are replaced by G_i.

Example 3.3 Let $\varphi \stackrel{\text{def}}{=} \Box(p \supset \Diamond q) \vee (\Diamond r \wedge \Diamond s)$. Here $\Upsilon(\varphi) = \{\Box(p \supset \Diamond q), \Diamond r, \Diamond s\}$. The final (classical) goal condition corresponds to $G_1 \vee (G_2 \wedge G_3)$, where G_1, G_2 and G_3 correspond to the accepting conditions of automata for $\Box(p \supset \Diamond q)$, $\Diamond r$, and $\Diamond s$.

Formula Splitting

On the other hand a formula transformation can be used to reduce the size of the transition formulae. Consider for example the propositional formula $\alpha \stackrel{\text{def}}{=} \Box \bigwedge_{d \in D} closed(d)$. The automaton generated for this formula has transitions of size $|D|$, and therefore the causal rules have size exponential in $|D|$. In this case, however, we could use the fact that α is equivalent to $\bigwedge_{d \in D} \Box closed(d)$, and use the method described above to generate $|D|$ automata with a single proposition in their transitions. Again, in this case we go from an exponential translation to a polynomial one. The transformation we have done to formula α is what we call *formula splitting*.

Splitting can be generalized to any combination of boolean formulae. In our implementation, before generating the automata, we preprocess the TEG formula using the follwing f-FOLTL equivalences:

$$\phi \mathsf{U} (\psi \vee \chi) \equiv (\phi \mathsf{U} \psi) \vee (\psi \mathsf{U} \chi),$$
$$\phi \mathsf{R} (\psi \wedge \chi) \equiv (\phi \mathsf{R} \psi) \wedge (\psi \mathsf{R} \chi),$$

and other similar equivalences that hold for the temporal connectives R and $\bigcirc$, effectively "pulling" binary connectives up in the formulae. With this technique, we generate more automata but avoid the risk of exponential explosion.

The formal implementation is given by Algorithm 3.2. The SPLIT function calls repeatedly the BREAKUP function until a fixed point is reached. BREAKUP simply applies one an f-FOLTL identity generating an equivalent formula.

Algorithm 3.2 A simple algorithm for splitting formulae to avoid blowups

```
1: function BREAKUP(f-FOLTL formula φ)
2:     if φ unifies with ○(ψ ∧ χ) then
3:         return ○(BREAKUP(ψ)∧BREAKUP(χ))
4:     else if φ unifies with ψ U(χ ∨ ζ) then
5:         return BREAKUP(ψ)UBREAKUP(χ) ∨ BREAKUP(ψ)UBREAKUP(ζ)
6:     else if φ unifies with ψ R(χ ∧ ζ) then
7:         return BREAKUP(ψ)UBREAKUP(χ) ∧ BREAKUP(ψ)UBREAKUP(ζ)
8:     end if
9: end function
10: function SPLIT(f-FOLTL formula φ)
11:     ψ ← φ stripped from QPrefix(φ)
12:     repeat
13:         ψ′ ← ψ
14:         ψ ←BREAKUP(ψ)
15:     until ψ′ = ψ
16: end function
```

3.4.4 Search Space Pruning by Progression

As previously noted, planners for TEGs such as TLPLAN are able to prune the search space by progressing temporal formulae representing the goal. A state *s* is pruned by progression if the progressed temporal goal in *s* is equivalent to false. Intuitively, this means that there is no possible sequence of actions that when executed in *s* would lead to the satisfaction of the goal.

Using our approach we can also prune the search space in a similar way. We illustrate the intuition in the propositional case. Suppose we have constructed an NFA for the propositional TEG $\mathcal{G}$. Since our NFAs have no non-final states that do not lead to a final state, if at some state during the plan all fluents E_q are false for every $q \in Q$, then this means that the goal will never be satisfied. We can also do this in the first-order case by considering the quantifiers of the TEG.

In the planning domain the pruning can be achieved in two ways. One way is to add

$$\mathsf{QPrefix}(\varphi) \bigvee_{q \in Q} E_q(\vec{x})$$

as a state constraint (or *safety constraint*). The other way is to add this condition to all of the action's

Prb.	Comp.	No.	CR+FF		DP+FF$_{\mathcal{X}}$	
	CR/DP	Sts.	t	ℓ	t	ℓ
1	.02/.02	2	.02	6	.02	6
2	.02/.01	2	.02	8	.01	8
3	.09/.06	15	.04	10	.04	10
4	.06/.07	5	.03	6	.02	6
5	.07/.03	6	.04	15	.03	15
6	.49/.39	37	.19	16	.16	16
7	.05/.03	6	.05	9	.11	10
8	.07/.06	15	.05	10	.04	12
9	.01/.02	4	.03	18	.03	18
10	.04/.05	6	.07	32	.05	15
11	.08/.04	5	.06	22	.03	20
12	.09/.02	5	.50	25	.03	24
13	.09/.05	6	m	–	.04	28
14	.32/.05	5	m	–	.10	33
15	.07/.03	5	.11	31	.09	34
16	.09/.04	10	m	–	.07	46

Table 3.1: A comparison between the two translation approaches for 16 problems on the Robots domain.

preconditions (Bacchus and Ady, 1999; Rintanen, 2000). This second approach however, implies regressing the precondition, so it is prone to the worse-case exponential blowup discussed above.

This means that we are able to add certain types of TDCK to our planning domains by simply adding the TDCK to the goal. Currently, though, our logic does not have the Goal modality that is used in TLPLAN, which enables it to tailor the control depending on the goal.

3.5 Implementation and Experiments

We implemented a compiler that takes a planning domain and a TEG in EPNF f-FOLTL as input and generates a classical planning problem as described in Section 4. Furthermore, the program can convert the new problem into PDDL, thereby enabling its use with a wide variety of planners.

It is hard to perform an accurate experimental analysis of our approach for two reasons. First, there are no standard benchmark problems for planning with TEGs. Second, none of the planners for TEGs is heuristic, so it is not hard to contrive problems easily solvable by our approach but completely out of the reach of non-heuristic planners.

The rest of the section is divided in two parts. First, it provides an evaluation of the relative performance of the two translations we have proposed. Then it provides analysis of the performance of our approach relative to that of existing planners.

Prb.	DP+FF$_{\mathcal{X}}$		TPBA/$_{\text{dfs+c}}$		TPBA/$_{\text{dfs}}$		TPBA/$_{\text{bfs+c}}$		TPBA/$_{\text{bfs}}$	
	t	ℓ	t	ℓ	t	ℓ	t	ℓ	t	ℓ
1	.00	2	.06	2	0.3	2	0.24	2	0.44	2
2	.01	5	.51	15	30	563	0.96	5	44.42	5
3	.01	6	.58	17	29.56	563	1.3	5	47.91	5
4	.02	7	1.20	25	m	–	3.29	7	m	–
5	.01	13	1.53	34	m	–	11.66	10	m	–
6	.01	16	1.68	38	m	–	28.87	12	m	–
7	.02	17	2.00	45	m	–	82.57	15	m	–
8	.02	17	2.13	49	m	–	35.69	17	m	–
9	.03	21	2.50	52	m	–	13.37	20	m	–
10	.07	41	7.18	91	m	–	126.25	35	m	–
11	.09	46	8.66	101	m	–	m	–	m	–
12	.10	49	10.06	113	m	–	m	–	m	–
13	.28	67	19.89	131	m	–	m	–	m	–
14	2.45	74	28.28	236	m	–	m	–	m	–
15	4.54	115	43.07	300	m	–	m	–	m	–

Table 3.2: Our approach compared to search control with Büchi automata

Domain	*Problems solved*		*Speedup (s)*					*Length ratio (r)*		
	FF$_{\mathcal{X}}$	TLPLAN	$s < 2$	$2 \leq s < 10$	$10 \leq s < 100$	$100 \leq s < 1000$	$s \geq 1000$	$r = 1$	$1 \leq r < 1.3$	$r \geq 1.3$
ZenoTravel (25)	21(84%)	9(36%)	0	1(11%)	2(22%)	5(56%)	1(11%)	8(89%)	1(11%)	0
Logistics (23)	23(100%)	17(74%)	1(6%)	4(24%)	4(24%)	6(35%)	2(12%)	14(82%)	2(12%)	1(6%)
Robot (16)	16(100%)	9(56%)	0	4(44%)	3(33%)	2(22%)	0	5(56%)	4(44%)	0%

Table 3.3: Performance of our approach compared to TLPLAN in 3 benchmark domains. *Speedup* and the *length ratio* are shown for instances that were solved by both planners. *Speedup* (resp. *length ratio*) is the time taken (resp. plan length obtained) by TLPLAN over that of our approach.

3.5.1 Axioms versus Causal Rules

We have seen in theory that both of our translations have an exponential worst-case, and that the translation to axioms is more compact. In this section we analyze how this is reflected in the performance of real planning systems.

We designed and ran a suite of problems in the robot domain (as per Fig. 3.1) to test the relative effectiveness of the two translations. In each experiment, we compiled the planning problem to PDDL. To evaluate the translation to causal rules (CR), we used FF as our heuristic planning engine (CR+FF). For the translation to derived predicates (DP), we used FF$_{\mathcal{X}}$ (DP+FF$_{\mathcal{X}}$), an extension of FF proposed by Thiébaux, Hoffmann, and Nebel (2005) that supports derived predicates.

Table 3.1 presents results obtained for various temporal goals by both of our translations. The second and third columns show statistics about the translation. The second column shows the time taken in each translation, and the third shows the number of states of the automata representing the goal. The rest of the columns show the time (t) and length (ℓ) of the plans for each approach. The character 'm' stands

for *ran out of memory*.

Although the relative performance in many cases is comparable, the derived predicates approach is never inferior to the causal rules approach, and sometimes it is clearly superior.

The causal rule approach may generate problems that cannot be handled by the FF planner, even when the number of states in the automata is quite low. That is the case of goal number 14, which corresponds to the formula: $\Diamond[(AllIn(R_4) \vee AllIn(R_3)) \wedge \bigcirc AllClosed)] \wedge \Diamond\Box at(O_1, C_1)$, where *AllClosed* stands for a formula where all doors are closed, and $AllIn(r)$ stands for "all objects are in *r*." Although the automaton for this goal is relatively simple, the grounded formulae in the transitions (which are needed by the causal rules approach), are quite big. This produces very large conditions in conditional effects, causing FF to run out of memory in the preprocessing phase, in which this planner converts ADL operators into STRIPS operators.

Since we have shown that the causal rule approach is not superior to the derived predicates approach, in the next subsection we focus our attention only on the derived predicates approach.

3.5.2 Comparison to State of the Art

We have compared the performance of our translation in conjunction with $FF_{\mathcal{X}}$ against TLPLAN and the planner presented by Kabanza and Thiébaux (2005) (henceforth, TPBA), which uses Büchi automata to control search. The TPBA planner is not heuristic and is implemented in Scheme. It offers four templates to write automata. We conducted experiments in the robots domain for goals that fit into these templates. We have used one of them, which is of the form $\Diamond(p_1 \wedge \bigcirc(\Diamond p_2 \wedge \ldots \wedge \bigcirc\Diamond p_n)\ldots)$.[4] Results are shown in Table 3.2

TPBA is significantly outperformed by our approach, even in the presence of extra control information added by hand (this is indicated by the '+c' in the table). In dfs mode, TPBA is able to solve every problem but more slowly and with inferior quality. In the bfs mode with no control information, TPBA fails for goal 4, which is "O_1 must eventually be at R_2, then at R_4, then at C_1, then at R_3, and finally at C_2". On the other hand, TPBA fails in bfs mode with control information for goal 10, which is defined as "eventually O_1 at R_2, then eventually all objects in R_4, and finally all objects in C_1." The control information added by hand in this case is "do not close any doors."

On the other hand, Table 3.3 presents a comparison of our approach and TLPLAN in three domains. For each domain, we designed a set of reasonably natural TEGs. Both *ZenoTravel* (a travel agency domain) and *Logistics* (a package delivery domain) are benchmark domains from past IPC. To get a feeling for the types of goals we used, here is an example of a goal in the *ZenoTravel* domain: "*persons P_1 and P_2 want to meet in some city and then eventually be at C_1 and C_2.*" Our third test domain, the *Robot* domain (Bacchus and Kabanza, 1998) describes a robot that moves between rooms and carries

[4] Three more are available. One is for classical goals, another is for cyclic (infinite) goals, and the third is very similar to the one we are using.

objects. An example of a goal in the robot domain is: *"open all the doors, then deliver objects to the rooms, and then close all doors."*

Since most of the goals were unsolvable by TLPLAN (exceeding the 1GB RAM limit), we needed to add extra TDCK to TLPLAN so that it could show more of its potential. We conclude that our approach significantly outperforms TLPLAN. This can be seen in the *speedup* metric in the table, where a significant percentage of the problems are solved over two orders of magnitude faster. Note that in some cases, the plans that are returned are slightly longer than those obtained by TLPLAN. This is usually the case with heuristic planners, where there is a tradeoff between optimality and speed. Some plans are not solved by $FF_{\mathcal{X}}$ in the ZenoTravel domain, which is due to the presence of universally quantified disjunctive goals.

The translation times for each of these problems was very low; in most cases it was less than 15% of the planning time. Furthermore, the ratio $|A_\varphi|/|\varphi|$, where A_φ is the number of states of $|A_\varphi|$, and $|\varphi|$ is the size of the TEG φ never exceeds 1.0, which illustrates that our automata translation does not blow up easily for natural TEGs.

The results shown, although good, are not surprising. We have compared our heuristic approach to the blind-search approach (plus pruning) of TLPLAN. Consequently, these results were expected. TLPLAN is particularly good when used with classical goals and a fair amount of hand-coded TDCK. Our approach has the advantage that it is able to guide the search effectively towards the satisfaction of a TEG with no need for hand-coded TDCK.

3.6 Discussion

There are two decisions at the core of our approach that deserve further discussion. The first, has to do with the decision to choose a reformulation approach, and the second with the choice of language for TEGs. We discuss both decisions below.

3.6.1 Why a Reformulation Approach?

Instead of designing a specific heuristic for TEGs, we chose a reformulation approach, which, as we have seen, may blow up the representation exponentially. Why is a reformulation approach justifiable when it is conceivable that specific heuristics could be adapted to plan with TEGs?

There are three reasons why we think the reformulation approach has merit on its own. First, as we have seen, we generate PDDL output, which can be used by *any* PDDL-compliant planner. This is important because it means that potentially any advance in classical planning can be leveraged for planning with TEGs.

Second, a reformulation approach serves as a useful benchmark for future comparison. Since TEGs are very relevant, we expect future work that adapts classical approaches to TEGs. Those extensions

however, will only be of value if they can be proven to be superior to the original algorithm when applied to the translated problem. Thus, we think our reformulation approach services the planning community by providing a uniform *baseline* for future experimental comparison.

Third, using the reformulation approach, it is possible to gain insights about designing specific heuristics for TEGs. As an example, consider that we wanted to adapt the relaxed plan heuristic in FF for TEGs. To anyone deciding to take such an approach we would recommend to replicate somehow the information that is provided by the extra automata predicates during the relaxed plan expansion, rather than to adapt the progression algorithm to evaluate the formula in the relaxed states. Why? The answer is a pragmatic one. Updating the truth values of predicates is quite an easy task. Progressing a formula, on the other hand, might need some effort, especially regarding simplification, that the designer of the heuristic might not want to pay. Our automata basically encode *all possible ways* in which a formula could be progressed. We pay a price for computing this representation *only once* however. Such a price would be paid multiple times (maybe more than one could afford) if progression is computed during heuristic computation.

By observing in which cases an algorithm *A* does not do well using the translated domain, one could adapt the heuristic in *A* to do better in these cases. It is thus conceivable and quite possible that a TEG adaptation of an algorithm originally designed for classical goals may outperform the original algorithm used in conjunction with our reformulation.

3.6.2 Why Not LTL and Büchi Automata?

A fundamental design decision of this work is the use of a finite logic, f-FOLTL, over the standard LTL logic. As we argued above, the main motivation is to provide a language for goals that is more compatible with classical planning technology, in which plans returned are finite.

The use of f-FOLTL has several practical advantages. One advantage is that formulae that would require infinite plans are not allowed by the logic, sometimes even generating automata that accept the empty language. This allows most planners to immediately realize that the goal is not achievable.

Another advantage of f-FOLTL over LTL, is that there is a very clear relationship between acceptance condition of automata for f-FOLTL, and the representation of this condition in the planning domain. Indeed, being at an accepting state of a PNFA is equivalent to accepting, and therefore equivalent to satisfying the f-FOLTL formula. The acceptance condition can be represented directly from the automaton, without any extra information.

With an LTL approach, through Büchi automata (BA), there is not always a clear relationship between being in an accepting state and satisfying the LTL formula. The main reason is that the acceptance condition for a BA requires visiting an accepting state infinitely often. To clarify this, consider the two Büchi automata shown in Figure 3.5. The automaton shown in (a) has only final states. Clearly in this case it is not possible to interpret being in an accepting state as satisfying the goal, as we do with PNFA.

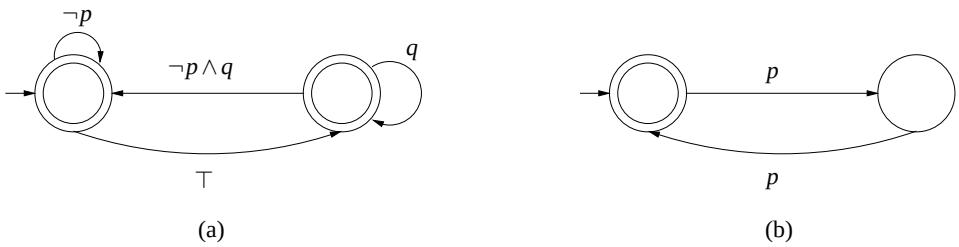


Figure 3.5: Büchi automata for (a) $\Box(p \supset \bigcirc q)$ and (b) $\Box$p. The automaton (a) is generated by LTL2BA (Gastin and Oddoux, 2001), whereas (b) was built by hand.

Otherwise we could accept as valid an empty plan starting from an initial planning state where p is true, and q is false. On the other hand, in general we cannot expect to interpret not being in an accepting state as rejection. For the contrived automata for $\Box p$ shown in Figure 3.5(b), we would not accept an empty plan as a solution if the initial state already satisfies p. Although such an automaton is not generated by standard LTL-to-BA algorithms, we cannot guarantee that similar situations do not occur for other formulae.

We think that an approach that uses BA is indeed feasible if we have additional information as to what is exactly being checked in each state. This information is usually not available in the graphical representation of the BA but it is usually available in the internal data-structures used to construct the automaton. For example, the algorithm to construct BAs by Gerth *et al.* (1995) that we have adapted, the $Next(q)$ field contains the formula that needs to be checked in the next state while we are in automaton state q. $Next(q)$ can be used to determine whether or not we should accept a finite plan. The method requires evaluating the formula as if the current state repeated for ever.[5] As Cresswell and Coddington (2004) have shown, given an LTL formula φ, it is possible to construct a non-temporal formula that evaluates to true in a state s if and only if φ is true in a model that contains an infinite repetition of s. Such a formula could be somehow encoded in the planning problem to correctly determine when a plan has been found.

Nevertheless, the drawback of the approach sketched above is that information such as the $Next(q)$ formula, might not be available from off-the-shelf LTL-to-BA software. This is because typically BAs are simplified, which usually implies post-processing the automaton. A special modification to the simplifying algorithms might also be needed in order to keep information such as $Next(q)$ even after simplification. Arguably, such modifications are not straightforward.

[5]The repetition of the final state is what Bacchus and Kabanza (1998) call the *idling* of the final state. This is the standard way in which one usually determines whether or not an LTL formula is satisfied by a finite plan.

3.7 Summary and Related Work

In this chapter we proposed a method for reformulating planning instances with first-order TEGs into classical planning instances. With this reformulation in hand, we exploited domain-independent heuristic search to determine a plan. Our compiler generates PDDL so it is equally amenable to use with any PDDL-compliant classical planner.

There are many advantages to the quantifiers in our f-FOLTL language. In addition to providing a richer language for goal specification, f-FOLTL also results in more efficient goal processing. Propositionalizing out quantifiers, as must be done in previous approaches to this problem, increases the size of the reformulation as a function of the size of the domain and arity of predicates in the TEG. In particular, a propositional encoding requires grounding both the initial state of the automata and their transitions, making the compilation specific to the instance of the problem.

We tested our approach on more than 60 problems over 3 standard benchmark domains, comparing our results to TLPLAN. Using our method, the FF$_{\mathcal{X}}$ planner often produced orders of magnitude speedup compared to TLPLAN, solving some planning problems TLPLAN was unable to solve. Since FF$_{\mathcal{X}}$ propositionalizes its domains, it does not fully exploit the strength of our first-order goal encoding.

There are several pieces of related work. Rintanen (2000) proposed a reformulation of a subset of LTL into a set of ADL operators, which is restricted to a very limited set of TEGs. Pistore and colleagues (e.g. dal Lago, Pistore, and Traverso, 2002) used automata to encode goals for planning with model checkers. Their approach uses different goal languages and is not heuristic.

Cresswell and Coddington (2004) briefly outline a means of compiling LTL formulae to PDDL. They translate LTL to deterministic finite state machines (FSM) using progression (Bacchus and Kabanza, 1998), and then translate the FSM into an ADL-only domain. The accepting condition must be determined by simulating an infinite repetition of the last state. Further, the use of deterministic automata makes it very prone to exponential blowup with even simple goals. The authors' code was unavailable for comparison. They report that their technique is no more efficient than TLPLAN (Cresswell and Coddington, 2004), so we infer that our method is superior.

Kabanza and Thiébaux's work (2005) is distinct because they are able to generate infinite and cyclic plans. They compile infinite propositional LTL into a Büchi automaton. Then they use the automaton to guide planning by following a path in its graph from initial to final state, backtracking as necessary. The planner is more prone to get lost and the restriction to one automaton makes it vulnerable to blowup. In a recent poster publication (Baier and McIlraith, 2006c), we have presented a similar approach for propositional TEGs. Besides the expressiveness and efficiency issues related to propositionalizing TEGs, the reformulation presented generates only ADL operators, which, as shown here, are less efficient both in theory and in practice.

Finally, Edelkamp (2006a) provides a reformulation of PDDL3[6] into PDDL2.2 by encoding propositionalized LTL hard constraints and preferences into Büchi automata. The approach cannot be used directly to provide heuristic search guidance to achieve TEGs because the acceptance condition of a Büchi automata requires visiting final states an infinite number of times.

[6]PDDL3 supports a subset of LTL first-order temporally extended goals and preferences. It will be introduced in more detail in Section 4.2.3.

Chapter 4

Planning with Temporally Extended Preferences

4.1 Introduction

As we have seen in Chapter 2, classical planning requires a planner to find a plan that achieves a specified goal. In practice, however, not every plan that achieves the goal is equally desirable. Moreover, many applications requires returning plans that satisfy rich user preferences.

When we use the term rich user preferences, we refer to a range of possible properties that a user would potentially like their plans to optimize. Given some task to be achieved, users may have preferences over what goals to achieve, and under what circumstances. They may also have preferences over *how* goals are achieved – properties of the world that are to be achieved, maintained or avoided during plan execution, and/or adherence to a particular way of doing some or all of the tasks at hand. Interestingly, with the exception of Markov Decision Processes (MDPs), nontrivial user preferences have only recently been integrated into AI automated planning.

Planning with preferences involves not only finding a plan that achieves the goal, it requires finding one that achieves the goal while also optimizing the user's preferences. Unfortunately, finding an optimal plan can be computationally expensive. In such cases, we would at least like the planner to direct its search towards a reasonably preferred plan.

Planning with preferences is motivated by many applications. Indeed, since preferences play a significant role in human decision making, it is not hard to argue that most real-world planning applications will require some kind of preference reasoning. To mention a few, consider the Robocup@Home Sce-

nario[1], in which a robot achieves a variety of goals in a domestic environment. Such a robot would be required to constantly plan to achieve hard goals but would also certainly be required to take into account the preferences of their users. There exist many applications in software environments too. Requirements Engineering could be also viewed as a planning problem as the objective is to optimize a function in which the company's or the user's preferences play a significant role (e.g. Hui, Liaskos, and Mylopoulos, 2003). Finally, component software composition and WSC are other compelling applications.

In this chapter we provide a technique for planning with a rich class of user preferences. Most notably this class includes *temporally extended preferences*. The difference between a TEP and a so-called *simple* preference is that a simple preference expresses some desired property of the final state achieved by the plan, while a TEP expresses a desired property of the sequence of states traversed by the plan. For example, a preference that a shift worker work no more than 2 overtime shifts in a week is a temporally extended preference. It expresses a condition on a sequence of daily schedules that might be constructed in a plan. Planning with TEPs has been the subject of recent research (e.g. Delgrande, Schaub, and Tompits, 2007; Son and Pontelli, 2006; Bienvenu, Fritz, and McIlraith, 2006). It was also a theme of the 5th International Planning Competition (IPC-5).

The technique we propose in this chapter is able to plan with a class of preferences that includes those that can be specified in the Planning Domain Definition Language PDDL3 (Gerevini *et al.*, 2009). PDDL3 was specifically designed for IPC-5. It extends PDDL2.2 to include, among other things, facilities for expressing both temporally extended and simple preferences, where the temporally extended preferences are described by a subset of LTL. It also supports quantifying the value of achieving different preferences through the specification of a metric function. The metric function assigns to each plan a value that is dependent of the specific preferences the plan satisfies. The aim in solving a PDDL3 planning instance is to generate a plan that satisfies the hard goals and constraints while achieving the best possible metric value, optimizing this value if possible or at least returning a high value plan if optimization is infeasible.

Our technique is a two-step approach. The first step exploits the compilation technique we have presented in the previous chapter to convert planning problems with TEPs to equivalent problems containing only simple preferences defined over an extended planning domain. The second step solves the reformulated instance with a specialized solver that we have developed.

4.1.1 Contributions of this Chapter

The main contributions of this chapter follow.

- We use the reformulation approach of the previous chapter to show how temporally extended

[1] `http://www.ai.rug.nl/robocupathome/`

preferences can be transformed into simple preferences (i.e., preferences that only refer to the final state). Although this contribution is a straightforward application of our existing method, it is important because allows *any* planner for simple preferences to plan for TEPs. Also, it enables the use of distance-bases heuristics for TEGs.

- We develop a set of new heuristics, and a search algorithm that can exploit these heuristics to guide the planner towards preferred plans. Many of our heuristics are extracted from a *relaxed plan graph*. Previous heuristics for classical planning, however, are not well suited to planning with preferences. The heuristics we present here are specifically designed to address the tradeoffs that arise when planning to achieve preferences.

  Our search algorithm is also very different from previous algorithms used in planning. We prove that it has a number of attractive properties, including the ability to find optimal plans without having to resort to admissible heuristics. This is important because admissible heuristics generally lead to unacceptable search performance. Our method is also able to find optimal plans without requiring a restriction on plan length or make-span. This is important because such restrictions do not generally allow the planner to find a globally optimal plan. In addition, the search algorithm is incremental in that it finds a sequence of plans each one improving on the previous. This is important because in practice it is often necessary to trade off computation time with plan quality. The first plans in this sequence of plans can often be generated fairly quickly and provide the user with at least a working plan if they must act immediately. If more time is available, the algorithm can continue to search for a better plan. The incremental search process also employs a pruning technique to make each incremental search more efficient. The heuristics and search algorithm presented here can easily be employed in other planning systems.

- Our third and final contribution is that we have brought all of these ideas together into a working planning system called HPLAN-P. Our planner is built as an extension of the TLPLAN system (Bacchus and Kabanza, 1998). The basic TLPLAN system uses LTL formulae to express *domain control knowledge*; thus, LTL formulae serve to prune the search space. However, TLPLAN has no mechanism for providing heuristic guidance to the search. In contrast, our implementation extends TLPLAN with a heuristic search mechanism that guides the planner towards plans that satisfy TEPs, while still pruning those partial plans that violate hard constraints. We also exploit TLPLAN's ability to evaluate quantified formulae to avoid having to convert the preference statements (many of which are quantified) into a collection of ground instances. This is important because grounding the preferences can often yield intractably large domain descriptions. We use our implementation to evaluate the performance of our algorithm and to analyze the relative performance of different heuristics on problems from both the IPC-5 *Simple* and *Qualitative Preferences* tracks. We observe that planning performance is improved when using the heuristics

we propose. We also show that pruning is a technique that is sometimes be critical to finding good-quality plans.

4.1.2 Outline

In the rest of the chapter we first provide some necessary background. This includes a formal definition of preference-based planning and a brief description of the features of PDDL3 that our approach can handle. In Section 4.3 we describe the first part of our approach—a method for compiling a domain with temporally extended preferences into one that is solely in terms of simple (i.e., final state) preferences. Section 4.4 describes the heuristics and search algorithm we have developed. It also presents a number of formal properties of the algorithm, including characterizing various conditions under which the algorithm is guaranteed to return optimal plans. Section 4.5 presents an extensive empirical evaluation of the technique, including an analysis of the effectiveness of various combinations of the heuristics presented in Section 4.4. Section 4.7 summarizes our contributions and discusses related work after which we provide some final conclusions.

4.2 Background

For the rest of this chapter, we assume familiarity with STRIPS and ADL planning (described in Sections 2.1.1 and 2.1.2). We also assume familiarity with planning as heuristic search (cf. Section 2.2). Section 4.2.1 describes a variation of the well-known approach to computing domain-independent heuristics based on the computation of relaxed plans that is used by our planner to compute heuristics. As opposed to most well-known approaches, our method is able to handle ADL domains directly without having to pre-compile the domain into a STRIPS domain. Then, Section 4.2.2 defines formally the preference-based planning problem. Section 4.2.3 describes the planning domain definition language PDDL3, a recent version of PDDL that enables the definition of hard constraints, preferences, and metric functions.

4.2.1 Relaxed Plans for Function-Free ADL Domains

To compute heuristics for function-free ADL domains one can first transform the domain to STRIPS, using a well-known procedure described by Gazen and Knoblock (1997), and then compute the heuristic as usual. This is the approach taken by some systems (e.g. FF) but unfortunately this procedure can lead to a considerable blow up in the size of the original instance.

Our planner handles ADL domains, but takes a different approach. In particular, it computes the relaxed planning graph directly from the ADL instance, using an approach similar to that taken by the MARVIN planning system (Coles and Smith, 2007). To effectively handle relaxed ADL domains

(in which effects can be conditioned on negative facts), the relaxed states represent both the facts that become *true* and the facts that become *false* after executing a set of actions. To that end, the relaxed states are divided into two parts: a positive part, that represents added facts, and a negative part, that represents deleted facts.

When computing a relaxed graph for a state s, the set of relaxed states is a sequence of pairs of fact sets $(F_0^+,F_0^-),\ldots,(F_n^+,F_n^-)$, with $F_0^+ = s$ and $F_0^- = s^c$, where s^c is the set of facts not in s (i.e., the complement of s). Furthermore, if action a appears in the action layer at depth n, all facts that are added by a are included in the positive relaxed state at depth F_{k+1}^+, whereas facts that are deleted by a are added to F_{k+1}^-. Moreover, all facts in layer k are copied to layer $k+1$ (i.e. $F_n^+ \subseteq F_{k+1}^+$ and $F_k^- \subseteq F_{k+1}^-$).

Special care has to be taken in the evaluation of preconditions and conditions in conditional effects for actions, because negations could appear anywhere in those conditions. To evaluate a formula in a relaxed state, we evaluate its *negation normal form* (NNF) instead. In NNF, all negations appear right in front of atomic formulae. A formula can easily be converted to NNF by pushing negations in using the standard rules $\neg\exists.f \equiv \forall.\neg f$, $\neg\forall.f \equiv \exists.\neg f$, $\neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2$, $\neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2$, and $\neg\neg f \equiv f$.

Now assume we want to determine whether or not the formula ϕ is true in the relaxed state (F_k^+,F_k^-) in the graph with relaxed states $(F_0^+,F_0^-)\cdots(F_k^+,F_k^-)\cdots(F_n^+,F_n^-)$. Furthermore, let ϕ' be the NNF of ϕ. To evaluate ϕ we instead evaluate ϕ' recursively in the standard way, interpreting quantifiers and boolean binary operators as usual. When evaluating a positive fact f, we return the truth value of $f \in F_k^+$. On the other hand, when evaluating a negative fact $\neg f$, we return the truth value of $f \in F_k^-$. In short, $\neg f$ is true at depth k if f was deleted by an action or was already false in the initial state. More formally,

Definition 4.1 (Evaluation of an NNF formula in a relaxed state) *Let the relaxed planning graph constructed from the initial state s in a problem where the set of objects of the problem is Objs be* $(F_0^+,F_0^-)\cdots(F_k^+,F_k^-)$. *The following cases define when* ϕ *evaluates to true at level k of the relaxed graph, which is denoted as* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$.

- *if* ϕ *is an atomic formula then* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$ *iff* $\phi \in F_k^+$.
- *if* $\phi = \neg f$, *where* f *is an atomic formula, then* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$ *iff* $\phi \in F_k^-$
- *if* $\phi = \psi \wedge \xi$, *then* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$ *iff* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \psi$ *and* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \xi$.
- *if* $\phi = \psi \vee \xi$, *then* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$ *iff* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \psi$ *or* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \xi$.
- *if* $\phi = \forall x.\psi$, *then* $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \phi$ *iff for every* $o \in Objs$ $(F_k^+,F_k^-) \models_{\overline{\text{rg}}} \psi(x/o)$, *where* $\psi(x/o)$ *is the formula* ψ *with all free instances of x replaced by o.*[2]

[2] In our implementation, bounded quantification is used so that this condition can be checked more efficiently. In particular,

- *if* $\phi = \exists x.\psi$, *for some* $o \in Objs$ $(F_k^+, F_k^-) \models_{\text{rg}} \psi(x/o)$.

The standard relaxed plan extraction has to be modified slightly for the ADL case. Now, because actions have conditional effects, whenever a fact f is made true by action a there is a particular set of facts that is responsible for its addition, i.e. those that made both the precondition of a and the condition in its conditional effect true. When recursing from a subgoal f we add as new subgoals all those facts responsible for the addition of f (which could be in either part of the relaxed state).

As is the case with STRIPS relaxed planning graphs, whenever a fact f is reachable from a state by performing a certain sequence of legal actions, then f eventually appears in a fact layer of the graph. The same happens in these relaxed graphs. This is proven in the following proposition.

Proposition 4.1 *Let s be a planning state,* $R = (F_0^+, F_0^-)(F_1^+, F_1^-) \cdots (F_m^+, F_m^-)$ *be the relaxed planning graph constructed from s up to a fixed point, and* ϕ *be an NNF formula. If* ϕ *is true after performing a legal sequence of actions* $a_1 \cdots a_n$ *in s, then there exists some* $k \leq m$ *such that* $(F_k^+, F_k^-) \models_{\text{rg}} \phi$.

Proof: See Appendix B. ■

This proposition verifies that the relaxed planning graph is in fact a relaxation of the problem. In particular, it says that if the goal is not reachable in the relaxed planning graph then it is not achievable by a real plan.

Besides being a desirable property, this reachability result is key to some interesting properties of our search algorithm. In particular, as we see later, it is essential to proving that some of the bounding functions we employ will never prune an optimal solution (under certain reasonable assumptions).

4.2.2 Preference-based Planning

We now introduce the preference-based planning formulation following Baier and McIlraith (2008).

An instance of the PBP problem is a pair $(I, \preceq)$, where I is a standard planning instance. Furthermore, $\preceq$ is a transitive and reflexive relation in $\mathcal{P} \times \mathcal{P}$, where $\mathcal{P}$ contains precisely all plans for I. The $\preceq$ relation is the formal mechanism for comparing two plans for I. Intuitively $p_1 \preceq p_2$ stands for "p_1 is *at least as preferred as* plan p_2." Moreover, we use $p_1 \prec p_2$ to abbreviate that $p_1 \preceq p_2$ and $p_2 \not\preceq p_1$. Thus, $p_1 \prec p_2$ holds true if and only if p_1 is strictly preferred to p_2.

Definition 4.2 (Preferece-based Planning) *Given an instance* $N = (I, \preceq)$, *the preference-based planning problem consists of finding any plan in the set*

$$\Lambda_N = \{p \in \mathcal{P} \mid \textit{there is no } p' \in \mathcal{P} \textit{ such that } p' \prec p\}.$$

this means that not every object in *Objs* need be checked.

1.	$s_0 s_1 \cdots s_n \models (\texttt{always}\ \phi)$	iff	$\forall i : 0 \leq i \leq n,\ s_i \models \phi$
2.	$s_0 s_1 \cdots s_n \models (\texttt{sometime}\ \phi)$	iff	$\exists i : 0 \leq i \leq n,\ s_i \models \phi$
3.	$s_0 s_1 \cdots s_n \models (\texttt{at end}\ \phi)$	iff	$s_n \models \phi$
4.	$s_0 s_1 \cdots s_n \models (\texttt{sometime-after}\ \phi\ \psi)$	iff	$\forall i$ if $s_i \models \phi$ then $\exists j : i \leq j \leq n,\ s_j \models \psi$
5.	$s_0 s_1 \cdots s_n \models (\texttt{sometime-before}\ \phi\ \psi)$	iff	$\forall i$ if $s_i \models \phi$ then $\exists j : 0 \leq j < i,\ s_j \models \psi$
6.	$s_0 s_1 \cdots s_n \models (\texttt{at-most-once}\ \phi)$	iff	$\forall i : 0 < i \leq n$, if $S_i \models \phi$ then $\exists j : j \geq i,\ \forall k : k > j,\ s_k \models \neg\phi$

Figure 4.1: Semantics of PDDL3's temporally extended formulae that do not mention explicit time. The trajectory $s_0 s_1 \cdots s_n$ represents the sequence of states that results from the execution a sequence of actions $a_1 \cdots a_n$.

Intuitively, the set Λ_N contains all the optimal plans for an instance I with respect to $\preceq$. Observe that now, as opposed to classical planning, we are interested in any plan that is *optimal* based on $\preceq$.

Below, we define PDDL3, which defines the $\preceq$ relation in a quantitative way. At the end of the following section we define precisely the PBP problem in PDDL3.

4.2.3 Brief Description of PDDL3

PDDL3 was introduced by Gerevini *et al.* (2009) for the 5th International Planning Competition. It extends PDDL2.2 by enabling the specification of *preferences* and *hard constraints*. It also provides a way of defining a *metric function* that defines the quality of a plan dependent on the satisfaction of the preferences.

The current version of our planner handles the non-temporal and non-numeric subset of PDDL3, which was the language used for the *Qualitative Preferences* track in IPC-5. In this subset, temporal features of the language such as durative actions and timed fluents are not supported. Moreover, preference formulae that mention explicit times (e.g., using operators such as `within` and `always-within`) are not supported. Numeric functions (PDDL fluents) are not supported either. The rest of this section briefly describes the new elements introduced in PDDL3 that we do support.

Temporally Extended Preferences and Constraints

PDDL3 specifies TEPs and temporally extended hard constraints in a subset of a quantified LTL (Pnueli, 1977). These LTL formulae are interpreted over *trajectories*, which in the non-temporal subset of PDDL3 are sequences of states that result from the execution of a legal sequence of actions. Figure 4.1 shows the semantics of LTL-based operators that can be used in temporally extended formulae. The first two operators are standard in LTL; the remaining ones are abbreviations that can be defined in terms of standard LTL operators.

Temporally Extended Preferences and Constraints

Preferences and constraints (which can be viewed as being preferences that must be satisfied) are declared using the `:constraints` construct. Each preference is given a name in its declaration, to allow for later reference. By way of illustration, the following PDDL3 code defines two preferences and one hard constraint.

```
(:constraints
 (and
  (preference cautious
    (forall (?o - heavy-object)
      (sometime-after (holding ?o)
                      (at recharging-station-1))))
  (forall (?l - light)
    (preference p-light (sometime (turn-off ?l))))
  (always (forall ?x - explosive) (not (holding ?x)))))
```

The `cautious` preference suggests that the agent be at a recharging station sometime after it has held a heavy object, whereas `p-light` suggests that the agent eventually turn all the lights off. Finally, the (unnamed) hard constraint establishes that an explosive object cannot be held by the agent at any point in a valid plan.

When a preference is *externally* universally quantified, it defines a family of preferences, containing an individual preference for each binding of the variables in the quantifier. Therefore, preference `p-light` defines an individual preference for each object of type `light` in the domain. Preferences that are not quantified externally, like `cautious`, can be seen as defining a family containing a single preference.

Temporal operators cannot be nested in PDDL3. Our approach can however handle the more general case of nested temporal operators.

Precondition Preferences

Precondition preferences are atemporal formulae expressing conditions that should ideally hold in the state in which the action is performed. They are defined as part of the action's precondition. For example, the preference labeled `econ` below specifies a preference for picking up objects that are not heavy.

```
(:action pickup :parameters (?b - block)
 (:precondition (and (clear ?b)
                     (preference econ (not (heavy ?b)))))
 (:effect (holding ?b)))
```

Precondition preferences behave something like conditional action costs. They are violated each time the action is executed in a state where the condition does not hold. In the above example, `econ`

will be violated every time a heavy block is picked up in the plan. Therefore these preferences can be violated a number of times.

Simple Preferences

Simple preferences are atemporal formulae that express a preference for certain conditions to hold in the final state of the plan. They are declared as part of the goal. For example, the following PDDL3 code:

```
(:goal (and (delivered pck1 depot1)
            (preference truck (at truck depot1))))
```

specifies both a hard goal (`pck1` must be delivered at `depot1`) and a simple preference (that `truck` is at `depot1`). Simple preferences can also be externally quantified, in which case they again represent a family of individual preferences.

Metric Function

The metric function defines the quality of a plan, generally depending on the preferences that have been achieved by the plan. To this end, the PDDL3 expression `(is-violated name)`, returns the number of individual preferences in the `name` family of preferences that have been violated by the plan. When `name` refers to a precondition preference, the expression returns the *number of times* this precondition preference was violated during the execution of the plan.

The quality metric can also depend on the function `total-time`, which, in the non-temporal subset of PDDL3, returns the plan length, and the actual duration of the plan in more expressive settings. Finally, it is also possible to define whether we want to maximize or minimize the metric, and how we want to weigh its different components. For example, the PDDL3 metric function:

```
(:metric minimize (+ (total-time)
                     (* 40 (is-violated econ))
                     (* 20 (is-violated truck))))
```

specifies that it is twice as important to satisfy preference `econ` as to satisfy preference `truck`, and that it is less important, but still useful, to find a short plan.

In this chapter we focus on metric functions that mention only `total-time` or `is-violated` functions, since we do not allow function symbols in the planning domain.

4.3 Preprocessing PDDL3

As described in the previous section, PDDL3 supports the definition of temporally extended preferences in a subset of LTL. A brute force method for generating a preferred plan would be to generate all

plans that realize the goal and then to rank them with respect to the PDDL3 metric function. However, evaluating plans once they have been generated is not efficient because there could be many plans that achieve the goal. Instead, we need to be able to provide heuristic guidance to the planner to direct it towards the generation of *high-quality* plans. This involves estimating the merit of partial plans by estimating which of the TEPs could potentially be satisfied by one of its extensions (and thus estimating the metric value that could potentially be achieved by some extension). With such heuristic information the planner could then direct the search effort towards growing the most promising partial plans.

To actively guide the search towards plans that satisfy the problem's TEPs we develop a two-part approach. The first component of our approach is to exploit the techniques presented in Chapter 3 to convert a planning domain containing TEPs into one containing an equivalent set of simple (final-state) preferences. Simple preferences are quite similar to standard goals (they express soft goals), and thus this conversion enables the second part of our approach, which is to extend existing heuristic approaches for classical goals to obtain heuristics suitable for guiding the planner toward the achievement of this new set of simple preferences. The development and evaluation of these new heuristics for simple preferences is one of the main contributions of our work and is described in the next section. That section also presents a new search strategy that is effective in exploiting these heuristics.

In this section we describe the first part of our approach: how the techniques of Chapter 3 can be exploited to compile a planning domain containing TEPs into a domain containing only simple preferences. Besides the conversion of TEPs we also describe how we deal with the other features of PDDL3 that we support (i.e., those described in the previous section).

4.3.1 Temporally Extended Preferences and Constraints

In Chapter 3 we presented a technique that can construct an automaton A_φ from a temporally extended formula φ. The automaton A_φ has the property that it accepts a sequence of states (e.g., a sequence of states generated by a plan) if and only if that sequence of states satisfies the original formula φ. The technique works for a rich subset of first-order linear temporal logic formulas that includes all of PDDL3's TEPs. It also includes TEPs in which the temporal operators are nested, which is not allowed in PDDL3. To encode PDDL3 preference formulae, each preference formula is represented as an automaton. Reaching an accepting condition of the automaton corresponds to satisfying the associated preference formula.

The techniques presented in Chapter 3 were aimed at planning with temporally extended goals, not preferences. Up to the construction of the automata for each temporally extended formula, our approach is identical to that taken in Chapter 3. However, Chapter 3 proposes the use of derived predicates or regression to embed the automata in the planning domain. In this Chapter, we have chosen a different approach that is more compatible with the underlying TLPLAN system we employed in our implementation. In the rest of the section, we give some more details on the construction of automata and the way

we embed these automata into a planning domain.

From PDDL3 to PNFA

The compilation process first constructs a parameterized nondeterministic finite-state automaton (PNFA) A_φ for each temporally extended preference or hard constraint expressed as an LTL formula φ.

PDDL3 preferences, however, are not written using standard LTL operators. Thus we first transform each PDDL3 operator into a standard formula. Below give a declarative definition of the ToLTL operator, which transforms PDDL3 temporally extended formulas into f-FOLTL.

- $\mathsf{ToLTL}(Qx\varphi) = Qx\,\mathsf{ToLTL}(\varphi)$, for $Q \in \{\forall, \exists\}$.
- $\mathsf{ToLTL}(\neg\varphi) = \neg\mathsf{ToLTL}(\varphi)$.
- $\mathsf{ToLTL}(\varphi * \psi) = \mathsf{ToLTL}(\varphi) * \mathsf{ToLTL}(\psi)$, for any Boolean connective $*$.
- $\mathsf{ToLTL}(\texttt{always}(\varphi)) = \Box\mathsf{ToLTL}(\varphi)$.
- $\mathsf{ToLTL}(\texttt{sometime}(\varphi)) = \Diamond\mathsf{ToLTL}(\varphi)$.
- $\mathsf{ToLTL}(\texttt{at-end}(\varphi)) = \Box\Diamond\mathsf{ToLTL}(\varphi)$.
- $\mathsf{ToLTL}(\texttt{at-most-once}(\varphi)) = \Box(\varphi' \supset \varphi'\,\mathsf{U}(\mathsf{final} \vee \Box\neg\varphi'))$, where $\varphi' = \mathsf{ToLTL}(\varphi)$.
- $\mathsf{ToLTL}(\texttt{sometime-before}(\varphi, \psi)) = \Box\neg\varphi' \vee (\neg\varphi' \wedge \neg\psi')\,\mathsf{U}(\psi' \wedge \neg\varphi' \wedge \bigcirc\Diamond\varphi')$, where $\mathsf{ToLTL}(\varphi) = \varphi'$, and $\mathsf{ToLTL}(\psi) = \psi'$.
- $\mathsf{ToLTL}(\texttt{sometime-after}(\varphi, \psi)) = \Box(\varphi' \supset \Diamond\psi')$, where $\mathsf{ToLTL}(\varphi) = \varphi'$, and $\mathsf{ToLTL}(\psi) = \psi'$.
- $\mathsf{ToLTL}(\varphi) = \varphi$ if φ does not if none of the previous expansions apply.

Our transformation generates a formula that is equivalent to the original one, as shown by the following result.

Proposition 4.2 *Let φ be a PDDL3 formula, and σ be the states generated by a plan. Then $\sigma \models \varphi$ iff $Int(\sigma) \models \mathsf{ToLTL}(\varphi)$, where $Int(\sigma)$ represent the obvious map of a sequence of planning states into a first-order computation (defined in Definition 3.2 ,p. 24).*

Proof: Straightforward from the PDDL3 and f-FOLTL semantics. ■

With our preferences represented in f-FOLTL, we run the algorithm of Chapter 3 to obtain a PNFA for PDDL3 temporally extended preferences. Figure 4.2 shows two examples of PNFA constructed for PDDL3 formulae.

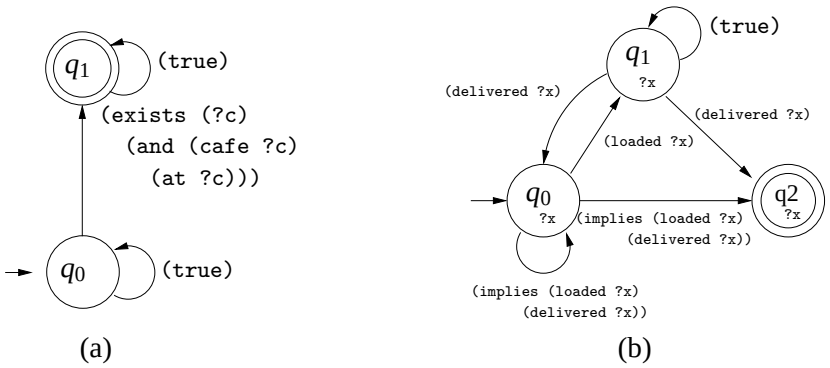


Figure 4.2: PNFA for (a) `(sometime (exists (?c) (and (cafe ?c) (at ?c))))`, and (b) `(forall (?x) (sometime-after (loaded ?x) (delivered ?x)))`. In both PNFA q_0 is the initial state and the accepting states are indicated by a double circle border.

A PNFA is useful for computing heuristics because it effectively represents all the different paths to the goal that can achieve a certain property; its states intuitively "monitor" the progress towards satisfying the original temporal formula. Therefore, while expanding a relaxed graph for computing heuristics, one is implicitly considering all possible (relaxed) ways of satisfying the property.

Representing the PNFA Within the Planning Problem

After the PNFA has been constructed it must be embedded within the planning domain. This is accomplished by extending the original planning problem with additional predicates that represent the state of the automaton in each plan state. If the planning domain has multiple TEPs (as is usually the case), a PNFA is constructed for each TEP formula and then embedded within the planning domain with automaton-specific automata-state predicates. That is, the final planning problem will contain distinct sets of automata-state predicates, one for each embedded automaton.

To represent an automaton within the domain, we define a predicate specifying the automaton's current set of states. When the automaton is parameterized, the predicate has arguments, representing the current set of automaton states for a particular *tuple of objects*. In our example, the fact `(aut-state q0 A)` represents that object `A` is in automaton state `q0`. Moreover, for each automaton we define an *accepting predicate*. The accepting predicate is true of a tuple of objects if the plan has satisfied the temporal formula for the tuple.

Rather than modify the domain's actions so that the automata state can be properly updated as actions are executed (as was in Chapter 3) we instead modified the underlying TLPLAN system so that after every action it would automatically apply a specified set of *automata updates*. Automata updates work like pseudo-actions that are performed automatically while a new successor is generated. When generating the successor to s after performing action a, the planner builds the new state s' by

adding and deleting the effects of a. When this is finished, it processes the automata updates over s', generating a new successor s''. The state s'' is then regarded as the actual successor of s after performing a. The compilation process can then avoid changes to the domain's actions and instead insert all of the conditions needed to transition the automata state in one self-contained addition to the domain specification.

Syntactically, the automata updates are encoded in the domain as first-order formulae that contain the `add` and `del` keywords, just like regular TLPLAN action effect specifications. For the automata of Figure 4.2(b), the update would include rules such as:

```
(forall (?x) (implies (and (aut-state q0 ?x) (loaded ?x))
                      (add (aut-state q1 ?x))))
```

That is, an object `?x` moves from state `q0` to `q1` whenever `(loaded ?x)` is true.

Analogously, we define an update for the accepting predicate, which is performed immediately after the automata update—if the automaton reaches an accepting state then we add the accepting predicate to the world state.

In addition to specifying how the automata states are updated, we also need to specify what objects are in what automata states in the initial state of the problem. This means we must augment the problem's initial state by adding a collection of automata facts. Given the original initial state and an automaton, the planner computes the states that every relevant tuple of objects can be in after the automaton has inputed the problem's initial state, and then adds the corresponding facts to the new problem. In our example, the initial state of the new compiled problem contains facts stating that both A and B are in states q_0 and q_2.

If the temporally extended formula originally described a hard constraint, the accepting condition of the automaton can be treated as an additional mandatory goal. During search we also use TLPLAN's ability to incrementally check temporal constraints to prune from the search space those plans that have already violated the constraint.

4.3.2 Precondition Preferences

Precondition preferences are very different from TEPs: they are atemporal, and are associated with the execution of actions. If a precondition preference `p` is violated n times during the plan, then the PDDL3 function `(is-violated p)` returns n.

Therefore, the compiled problem contains a *new* domain function `is-violated-counter-p`, for each precondition preference family `p`. This function keeps track of how many times the preference has been violated. It is initialized to zero and is (conditionally) incremented whenever its associated action is performed in a state that violates the atemporal preference formula. In the case where the preference is quantified, the function is parameterized, which allows us to compute the number of times different objects have violated the preference.

For example, consider the PDDL3 `pickup` action given above. In the compiled domain, the original declaration is replaced by:

```
(:action pickup :parameters (?b - block)
 (:precondition (clear ?b))
 (:effect (and (when (heavy ?b)
                 (increase (is-violated-counter-econ) 1)))
               (holding ?b))) ;; add (holding ?b)
```

4.3.3 Simple Preferences

As with TEPs, we add new *accepting predicates* to the compiled domain, one for each simple preference. We also define updates, analogous to the automata updates for these accepting predicates. Accepting predicates become true iff the preference is satisfied. Moreover, if the preference is quantified, these accepting predicates are parameterized: they can be true of some tuples of objects and at the same time be false for other tuples.

4.3.4 Metric Function

For each preference family *name*, we define a new *domain* function `is-violated-`*name*. The return values of these functions are defined in terms of the accepting predicates (for temporally extended and simple preferences) and in terms of the violation counters (for precondition preferences). If preference `p` is quantified, then the `is-violated-p` function counts the number of object tuples that fail to satisfy the preference.

By way of illustration, the TLPLAN code that is generated for the preference p-light defined in Section 4.2.3 is:

```
(def-defined-function (is-violated-p-light)
 (local-vars ?x)                                  ;; ?x is a local variable
    (and (:= ?x 0)                                ;; ?x initialized to 0
      (forall  (?l) (light ?l)
      (implies (not (preference_p-light_satisfied ?l))
        (:= ?x (+ ?x 1))))                        ;; increase ?x by 1 if
                                                  ;; preference not satisfied
      (:= is-violated-p-light ?x)))               ;; return total sum
```

where `preference_p-light_satisfied` is the accepting predicate defined for preference p-light. Note our translation avoids grounding by using quantification to refer to all objects of type `light`.

If the original metric function contains the PDDL3 function `(total-time)`, we replace its occurrence by the TLPLAN function `(plan-length)`, which counts the number of actions in the plan. Thus, actions are implicitly associated a unitary duration.

The metric function in the resulting instance is defined just as in the PDDL3 definition but by making reference to these new functions. If the objective was to maximize the function we invert the sign of the function body. Therefore, we henceforth assume that the metric is always to be minimized.

In the remainder of the chapter, we use the notation `is-violated`(p,N) to refer to the value of `is-violated-p` in a search node N. We will sometimes refer to the metric function as M, and we will use $M(N)$ to denote the value of the metric in search node N.

4.4 Planning with Preferences via Heuristic Search

As we have discussed earlier in this document, forward-chaining search guided by heuristics has proved to be a powerful and useful paradigm for solving planning problems. As shown above, the automata encoding of temporally extended preferences allows us to automatically augment the domain with additional predicates that serve to keep track of the partial plans' progress towards achieving the TEPs. The central advantage of this approach is that it converts the planning domain to one with simple preferences. In particular, now the achievement of a TEP is marked by the achievement of an accepting predicate for the TEP, which is syntactically identical to a standard goal predicate.

This means that, in the converted domain, standard techniques for computing heuristic distances to goal predicates can be utilized to obtain heuristic distances to TEP accepting predicates. For example, the standard technique based on a relaxed planning graph (Hoffmann and Nebel, 2001), which approximates the distance to each goal and each TEP accepting predicate can be used to heuristically guide a forward-chaining search.

Nevertheless, although the standard methods can be fairly easily modified in this manner, our aim here is to develop a search strategy that is more suitable to the problem of planning with TEPs. In particular, our approach aims to provide a search algorithm with three main features. First, the planner should find good plans, which optimize a supplied metric function. Second, it should be able to generate optimal plans, or at least be able to generate an improvement over an existing plan. Finally, since in some contexts it might be very hard to achieve an optimal plan—and hence a great deal of search effort could be required—we want the algorithm to find at least one plan as quickly as possible.

Heuristic search with non-admissible heuristics, like the relaxed goal distances employed in planners like FF can be very effective at quickly finding a plan. However, they offer no assurances about the quality of the plan they find. On the other hand, if an admissible heuristic is used, the plan found is guaranteed to be optimal (assuming the heuristic is admissible with respect to the supplied plan metric). Unfortunately, admissible heuristics typically perform poorly in practice (Bonet and Geffner, 2001). Hence, with an admissible heuristic the plan often fails to find any plan. This is typically unacceptable in practice.

In this section we develop a heuristic search technique that exploits the special structure of the

translated planning domains in order to (a) find a plan fairly rapidly using a non-admissible heuristic and (b) generate a sequence of improved plans that, under some fairly general conditions, terminates with an optimal plan by using a bounding technique. In particular, our search technique allows one to generate better plans—or even optimal plans—if one has sufficient computational resources available. It also allows one to improve on an existing plan and sometimes prove a plan to be optimal.

In the rest of the section we begin by describing a set of different heuristic functions that can serve to guide the search towards satisfying goals and preferences. Then, we describe our search algorithm and analyze some of its properties.

4.4.1 Heuristics Functions for Planning with Preferences

Our algorithm performs a forward search in the space of states guided by heuristics. Most of the heuristic functions given below are computed at a search node N by constructing a relaxed graph as described in Section 4.2.1. The graph is expanded from the planning state corresponding to N and is grown until all *goal* facts and all *preference* facts (i.e., instances of the accepting predicates) appear in the relaxed state or a fixed point is reached. The goal facts correspond to the hard goals, and the preference facts correspond to instantiations of the accepting predicates for the converted TEPs.

Since in our compiled domain we need to update the automata predicates, the procedure in Section 4.2.1 is modified to apply automata updates in action layers after all regular actions have been performed. On the other hand, because our new compiled domain has functions, in addition we modify the procedure in Section 4.2.1 to *ignore* all effects that directly affect the value of a function. This means that in the relaxed worlds, all preference counters will have the same value as in the initial state s. Note that since preference counters do not appear in the conditions of conditional effects or in the preconditions of actions, Proposition 4.1 continues to hold for relational facts; in particular, it holds for accepting predicates.

Below we describe a suite of heuristics that can be computed from the relaxed graph and can be used for planning with preferences. They are designed to guide the search towards (1) satisfying the goal, and (2) satisfying highly valued preferences, i.e., those preferences that are given a higher weight in the metric function. However, highly valued preferences can be very hard to achieve and hence guiding the planner towards the achievement of such preferences might yield unacceptable performance. To avoid this problem, our approach tries to account for the difficulty of satisfying preferences as well as their value, ultimately attempting to achieve a tradeoff between these two factors.

Goal Distance Function (G)

This function returns an estimate of the number of actions needed to achieve the goal (planning problems often contain a hard “must achieve” goal as well as a collection of preferences). G is the same as the

heuristic used by the FF planner but modified for the ADL case. The value returned by G is the number of actions contained in a relaxed plan that achieves the goal.

Preference Distance Function (P)

This function is a measure of how hard it is to reach the various preference facts. It is based on a heuristic proposed by Zhu and Givan (2005) for conjunctive hard goals, but adapted to the case of preferences. Let $\mathcal{P}$ be the set of preference facts that appear in the relaxed graph, and let $d(f)$ be the depth at which f first appears during the construction of the graph. Then $P(N) = \sum_{f \in \mathcal{P}} d(f)^k$, for some parameter k. Notice that unreachable preference facts (i.e., those not appearing in the relaxed graph) do not affect P's value.

Optimistic Metric Function (O)

The O function is an estimate of the metric value achievable from a search node N in the search space. O does not require constructing the relaxed planning graph. Rather, we compute it by assuming (1) no further precondition preferences will be violated in the future, (2) TEPs that are violated and that can be proved to be unachievable from N are regarded as false, (3) all remaining preferences are regarded as satisfied, and that (4) the value of $(\texttt{total-time})$ is evaluated to the length of the plan corresponding to N. To prove that a TEP p is unachievable from N, O uses a sufficient condition. It checks whether or not the automaton for p is currently in a state from which there is no path to an accepting state. Examples of LTL formulae that can be detected by this technique as always being falsified in the future are those of the form $(\texttt{always}\ \varphi)$. Indeed, as soon as φ becomes false, from no state in the automaton's current set of states will it be possible to reach an accepting state.

Although O clearly underestimates the set of preferences that can be violated by any plan extending N it is not necessarily a lower bound on the metric value of any plan extending N. It will be a lower bound when the metric function is non-decreasing in the number of violated preferences. As we will see later, lower bounds for the metric function can be used to soundly prune the search space and speed up search.

Definition 4.3 (NDVPL metric functions) *Let $\mathcal{I}$ be a (preprocessed) PDDL3 planning instance, let the set Γ contain its preferences, and let $\texttt{length}(N)$ be the length of the sequence of action that generated N. A metric function M is* non-decreasing in the number of violated preferences and in plan length *(NDVPL) iff for any two nodes N and N' it holds that:*

1. *If $length(N) \geq length(N')$, and for every $p \in \Gamma$, $\texttt{is-violated}(p,N) \geq \texttt{is-violated}(p,N')$, then $M(N) \geq M(N')$, and*

2. *If* $(\texttt{total-time})$ *appears in* M*, and* $length(N) > length(N')$*, and for every* $p \in \Gamma$, $\texttt{is-violated}(p,N) \geq \texttt{is-violated}(p,N')$*, then* $M(N) > M(N')$.

NDVPL metrics are natural when the objective of the problem is to minimize the metric function (as in our preprocessed instances). Problems with NDVPL metrics are those in which violating preferences never improves the metric of the plan. Furthermore, adding more actions to a plan that fail to satisfy any new preferences can never improve its metric. Below, in Remark 4.1, we see that *additive* metrics, which were the only metrics used in IPC-5, satisfy this condition.

Proposition 4.3 *If the metric function is NDVPL, then* $O(N)$ *is guaranteed to be a lower bound on the metric value of any plan extending* N.

Proof: The optimistic metric only regards as violated those preferences that are provably violated in every successor of N (i.e., in every state reachable from N by some sequence of actions). It regards as satisfied all remaining preferences. That is, O is evaluating the metric in a hypothetical node N_O such that for any node N' reachable from N and for every $p \in \Gamma$ $\texttt{is-violated}(p,N_O) \leq \texttt{is-violated}(p,N')$. Furthermore, because O evaluates the plan length to that of N, our hypothetical node is such that $\texttt{length}(N_O) = \texttt{length}(N)$ and hence we have $\texttt{length}(N_O) \leq \texttt{length}(N')$. Since the metric function is NDVPL, it follows from Definition 4.3 that for every successor N' of N, $M(N_O) \leq M(N')$. It follows that $O(N)$ returns a lower bound on the metric value of any plan extending N. ■

The O function is a variant of the *"optimistic weight"* heuristic in the PPLAN planner (Bienvenu *et al.*, 2006). PPLAN *progresses* LTL preferences (as defined by Bacchus and Kabanza (1998)) through every node of the search space. The optimistic weight assumes as falsified only those LTL preferences that have progressed to false.

Best Relaxed Metric Function (B)

The B function is another estimate of the metric value achievable by extending a node N. It utilizes the relaxed planning graph grown from the state corresponding to N to obtain its estimate. In particular, we evaluate the metric function in each of the relaxed worlds of the planning graph and take B to be the minimum among these values. The metric function evaluated in a relaxed world w, $M(w)$, evaluates the `is-violated` functions directly on w, and evaluates $(\texttt{total-time})$ as the length of the sequence of actions that corresponds to N.

For the case of NDVPL metric functions, B is similar to O, but can return tighter estimates. Indeed, note that the last layer of the relaxed graph contains a superset of the preference facts that can be made true by some successor to the current state. Also, because the counters for precondition preferences are not updated while expanding the graph, the value of the `is-violated` functions for precondition preferences is constant over the relaxed states. This represents the implicit assumption that no further

precondition preferences will be violated. The metric value of the relaxed worlds does not increase (and sometimes actually decreases), since the number of preference facts increases in deeper relaxed worlds. As a result, the metric of the deepest relaxed world is the one that will be returned by B. This value corresponds to evaluating the metric function in a relaxed state where: (1) `is-violated` functions for precondition preferences are identical to the ones in N, (2) preference facts that do not appear in the relaxed graph are regarded as violated, and (3) all remaining preferences are regarded as satisfied. This condition (2) is stronger than condition (2) in the definition of O above. Indeed, no preference that is detected as unsatisfiable by the method described for O can appear in the relaxed graph, since there is no path to an accepting state of that preference. Hence, no action can ever add the accepting predicate for the preference.

By using the relaxed graph, B can sometimes detect preferences that are not satisfiable by any successor of N but that cannot be spotted by O's method. For example, consider we have a preference $\varphi =$ `(sometime f)`, and consider further that fact `f` is not reachable from the current state. The myopic O function would regard this preference as satisfiable, because it is always possible to reach the final state of the automaton for formula φ (the automaton for `f` looks like the one in Figure 4.2(a)). On the other hand, `f` might not appear in the relaxed graph—because `f` is unreachable from the current state—and therefore B would regard φ as unsatisfiable.

These observations lead to the conclusion that $B(N)$ will also be a lower bound on the metric value of any successor of N under the NDVPL condition.

Proposition 4.4 *If the metric function is NDVPL, then $B(N)$ is guaranteed to be a lower bound on the metric value of any plan extending N.*

Proof: Proposition 4.1 implies that all preference facts that could ever be achieved by some successors of N will eventually appear in the deepest relaxed world. Because the metric is NDVPL, this implies that the metric value of the deepest relaxed world is also the minimum, and therefore such a value will be returned by the B function. Now we can apply the same argument as in the proof for Proposition 4.3, since the returned metric value corresponds to evaluating the metric in a hypothetical node in which all `is-violated` counters are lower or equal than those of any plan extending N. ■

Discounted Metric Function ($D(r)$)

The D function is a weighting of the metric function evaluated in the relaxed worlds. Assume $w_0, w_1, \ldots, w_n$ are the relaxed worlds in the relaxed planning graph, where w_i is at depth i and the $w_0 = (s, s^c)$, i.e., the positive and negative facts of the state where $D(r)$ is being evaluated. Then the discounted metric, $D(r)$, is:

$$D(r) = M(w_0) + \sum_{i=0}^{n-1} (M(w_{i+1}) - M(w_i)) r^i, \tag{4.1}$$

where $M(w_i)$ is the metric function evaluated in the relaxed world w_i and r is a discount factor ($0 \leq r \leq 1$).

The D function is optimistic with respect to preferences that appear earlier in the relaxed graph (i.e., preferences that seem easy) and pessimistic with respect to preferences that appear later (preferences that seem hard). Intuitively, the D function estimates the metric value of plans extending the current state by "believing" more in the satisfaction of preferences that appear to be easier. Observe that $M(w_{i+1}) - M(w_i)$ is the amount of metric value *gained* when passing from relaxed world w_i to w_{i+1}. This amount is then multiplied by r^i, which decreases as i increases. Observe also that, although the metric gains are discounted, preferences that are weighted higher in the PDDL3 metric will also have a higher impact on the value of D. That is, D achieves the desired tradeoff between the ease of achieving a preference and the value of achieving it.

A computational advantage of the D function is that it is easy to compute. As opposed to other approaches, this heuristic never needs to make an explicit selection of the preferences to be pursued by the planner.

Finally, observe that when r is close to 1, the effect of discounting is low, and when it is close to 0, the metric is quickly discounted. When r is close to 0 the D function is myopic in the sense that it discounts heavily those preferences that appear deeper in the graph.

Algorithm 4.1 HPLAN-P's search algorithm

```
1: function SEARCH-HPLAN-P(initial state init, goal formula goal, a set of hard constraints hConstraints,
   metric function METRICFN, heuristic function USERHEURISTIC)
2:     frontier ← INITFRONTIER(init)                          ▷ initialize search frontier
3:     closed ← ∅
4:     bestMetric ←worst case upper bound
5:     HEURISTICFN ← G
6:     while frontier is not empty do
7:         current ← Best element from frontier according to HEURISTICFN
8:         if ¬CLOSED?(current, closed) and current satisfies hConstraints then
9:             if METRICBOUNDFN(current) < bestMetric then          ▷ pruning by bounding
10:                if current satisfies goal and its metric is < bestMetric then
11:                    Output plan for current
12:                    if this is first plan found then
13:                        HEURISTICFN ← USERHEURISTICFN
14:                        frontier ← INITFRONTIER(init)            ▷ search restarted
15:                        Reinitialize closed List
16:                    end if
17:                    bestMetric ← METRICFN(current)
18:                end if
19:                succ ← successors of current
20:                frontier ← merge succ into frontier
21:                closed ← closed ∪ {current}
22:            end if
23:        end if
24:    end while
25: end function
```

4.4.2 The Planning Algorithm

Our planning algorithm searches for a plan in a series of *episodes*. The purpose of each of these episodes is to find a plan for the goal that has a better value than the best found so far. In each planning episode a best-first search for a plan is initiated using some of the heuristics proposed above. The episode ends as soon as it finds a plan whose quality is better than that of the plan found in the previous episode. The search terminates when the search frontier is empty. The algorithm is shown as Algorithm 4.1.

When search is started (i.e., no plan has been found), the algorithm uses the goal distance function (G) as its heuristic in a standard best-first search. The other heuristics are ignored in this first planning episode. This is motivated by the fact that the goal is a hard condition that must be satisfied. In some problems the other heuristics (that guide the planner towards achieving a preferred plan) can conflict with achieving the goal, or might cause the search to become too difficult.

After finding the first plan, the algorithm restarts the search from scratch, but this time it uses some combination of the above heuristics to guide the planner towards a preferred plan. Let USERHEURISTIC() denote this combination. USERHEURISTIC() could be any combination of the above heuristic functions. Nevertheless, in this chapter we consider only a small subset of all possible combinations. In particular, we consider only *prioritized* sequences of heuristics, where the lower priority heuristics are used only to break ties in the higher priority heuristics.

Since achieving the goal remains mandatory, USERHEURISTIC() always uses G as the first priority, together with some of the other heuristics at a lower priority. For example, consider the prioritization sequence $GD(0.3)O$. When comparing two states of the frontier, the planner first looks at the G function. The best state is the one with lower G value (i.e., lower distance to the goal). However, if there is a tie, then it uses $D(0.3)$ (the best state being the one with a smaller value). Finally, if there is still a tie, it uses the O function to break it. In Section 4.5, we investigate the effectiveness of several such prioritized heuristics sequences.

Pruning the Search Space

Once we have completed the first planning episode (using G) we want to ensure that each subsequent planning episode yields a better plan. Whenever a plan is found, it will only be returned if its metric is lower than that of the last plan found (line 10).

Moreover, in each episode we can use the metric value of the previously found plan to prune the search space, and thus improve search performance. In each planning episode, the algorithm prunes from the search space any node N that we estimate cannot reach a better plan than the best plan found so far. This estimate is provided by the function METRICBOUNDFN(), which is given as an argument to the search algorithm. METRICBOUNDFN(N) must compute or estimate a lowerbound on the metric of any plan extending N.

Pruning is realized by the algorithm in line 9, when the condition in the *if* becomes false. As the value of *bestMetric* gets updated (line 17), the pruning constraint imposes a tighter bound causing more partial plans to be rejected.

The *O* and *B* heuristic functions defined above are well-suited to be used as METRICBOUNDFN(). Indeed, we tried both of them in our experiments. On the other hand, it is also simple to "turn-off" pruning by simply passing a null function as METRICBOUNDFN().

Discarding Nodes in Closed List

Under certain conditions, our algorithm will also prune nodes that revisit a plan state that has appeared in a previously expanded node. This is done for efficiency, and allows the algorithm to avoid considering plans with cycles.

The algorithm keeps a list of nodes that have already been expanded in the variable *closed*, just as in standard best-first search. Furthermore, when *current* is extracted from the search frontier, its state is checked against the set of closed nodes (line 8). If there exists a node in the closed list with the same state and a better or equal heuristic value (i.e., CLOSED?$(current, closed)$ is true), then the node *current* will be pruned from the search space.

Note that for two states to be identical in the compiled planning instance every boolean predicate has to coincide and, moreover, values assigned to each ground function also have to coincide. In particular, this means that `is-violated` counters in two identical states are also identical, i.e., the preferences are equally satisfied. Nevertheless, two search nodes with identical states can still be assigned different heuristic values. Given the way we have defined USERHEURISTIC(), different heuristic values will be assigned to nodes with identical states only when the metric function depends on (`total-time`). If the (`total-time`) function appears positively in the metric (i.e., the metric is such that for otherwise equally preferred plans, longer ones are never preferred to shorter ones), then discarding of nodes cannot prune any node that leads to an optimal plan. We discuss this further in the next section.

Finally, note that the cycles we are eliminating are those that occur in the compiled instance, *not* those occurring in the original instance. Indeed, in the original instance there might be LTL preferences that can be satisfied by visiting the same state twice. For example consider the preference: *eventually turn the light switch on and sometime after turn it off*. Any plan that contains the action *turn-on* immediately followed by *turn-off* satisfies the preference but also visits the same state twice. In our compiled domains however such a plan will not produce a cycle, and therefore will not be pruned. This is because the set of current states of the preference's automaton—represented by the automata domain predicates—changes when performing those actions; indeed it changes from a non-accepting state to an accepting state.

4.4.3 Properties of the Algorithm

In this section we show that under certain conditions our search algorithm is guaranteed to return *optimal* and *k-optimal* plans. We will prove this result without imposing any restriction on the USERHEURISTIC() function. In particular, we can still ensure optimality even if this function is inadmissible. In planning this is important, as inadmissible heuristics are typically required for adequate search performance.

The first requirement in our proofs is that the pruning performed by the algorithm is *sound*.

Definition 4.4 (Sound Pruning) *The pruning performed by Algorithm 4.1 is* sound *iff whenever a node N is pruned (line 9) the metric value of any plan extending N exceeds the current bound bestMetric.*

When Algorithm 4.1 uses sound pruning, no state will be incorrectly pruned from the search space. That is, node N is not pruned from the search space if some plan extending it can achieve a metric-value superior to the current bound. To guarantee that the algorithm performs sound pruning it suffices to provide a lowerbound function as input to the algorithm.

Theorem 4.1 *If* METRICBOUNDFN(N) *is a lower bound on the metric value of any plan extending N, then Algorithm 4.1 performs sound pruning.*

Proof: If node N is not in closed and is pruned from the search space then (a) METRICBOUNDFN$(N) \geq$ *bestMetric*. If METRICBOUNDFN() is a lower bound on the metric value of any plan extending N, then (b) METRICBOUNDFN$(N) \leq M(N_p)$ for any solution node N_p extending N. By putting (a) and (b) together we obtain that if N is not in closed and it is pruned, then $M(N_p) \geq$ *bestMetric*, for every solution node N_p extending N, i.e., pruning is sound. ■

As proven previously in Section 4.4.1, if the metric function is NDVPL, O and B will both be lower bound functions, and therefore provide sound pruning. Notice also that "turning off" pruning by having METRICBOUNDFN() return a value that is always less than *bestMetric*, also provides sound pruning.

The second requirement for optimality has to do with the discarding of closed nodes performed in line 8. To preserve optimality, the algorithm must not remove a node that can lead to a plan that is more preferred than any plan that can be achieved by extending nodes that are not discarded. Formally,

Definition 4.5 (Discarding of Closed Nodes Preserves Optimality) *The discarding of nodes by Algorithm 4.1 preserves optimality iff for any node N that is discarded in line 8, there is either already an optimal node (i.e., plan) N_O in the closed list or there exists a node N in frontier that can be extended to a plan with optimal quality.*

The condition defined above holds when using NDVPL metrics under fairly general conditions. In particular, it holds for any NDVPL metric that is independent of (`total-time`). It also holds if the

NDVPL metric depends on (`total-time`), and *O* or *B* is used as a first tie breaker after *G* or *P* in USERHEURISTIC(). Finally, it will hold if *D* is used as the first tie breaker for NDVPL metric functions that are *additive on total-time*.

Definition 4.6 (Additive on total-time (ATT)) *A metric function M is additive on total time (ATT) iff it is such that* $M(N) = M_P(N) + M_T(N)$*, where* $M_P(N)$ *is an expression that does not mention the function* (`total-time`)*, and* $M_T(N)$ *is an expression whose only plan-dependent function is* (`total-time`).

Intuitively, an ATT metric is a sum of a function that only depends on the `is-violated` functions, and a function that includes (`total-time`) but does not include any `is-violated` functions. Now we are ready to state our result formally.

Theorem 4.2 *The discarding of nodes done by Algorithm 4.1 preserves optimality if the Algorithm performs sound pruning, the metric function M is NDVPL and:*

1. *M is independent of* (`total-time`)*, or*
2. *M is dependent on* (`total-time`) *and O or B are used as the first tie breaker in* USERHEURISTIC() *after G or P, or*
3. *M is ATT and D is used as the first tie breaker in* USERHEURISTIC() *after G or P.*

Proof: See Appendix B. ■

An important fact about sound pruning is that it never prunes optimal plans from the search space, unless another optimal plan has already been found. An important consequence of this fact, is that the search algorithm will be able to find optimal plans under fairly general conditions. Our first result says that, under sound pruning, optimality is guaranteed when the algorithm terminates.

Theorem 4.3 *Assume Algorithm 4.1 performs sound pruning, and that its node discarding preserves optimality. If it terminates, the last plan returned, if any, is optimal.*

Proof: Each planning episode has returned a better plan, and the algorithm stops only when the final planning episode has rejected all possible plans. Since the algorithm never prunes or discards a node that can be extended to an optimal unless an optimal plan has already been found then no plan better than the last one returned exists. ■

Theorem 4.3 still does not guarantee that an optimal solution will be found because the algorithm might never terminate. To guarantee this we must impose further conditions that restrict the explored search space to be finite. Once we have these conditions, optimality is easy to prove since the search must eventually terminate.

Theorem 4.4 *Assume the following conditions hold:*

1. *The initial value of bestMetric (worst case upper bound) in Algorithm 4.1 is finite;*
2. *The set of cycle-free nodes N such that* METRICBOUNDFN(N) *is less than the initial value of bestMetric is finite;*
3. *Algorithm 4.1 performs sound pruning;*
4. *Node discarding in Algorithm 4.1 preserves optimality.*

Then Algorithm 4.1 is guaranteed to find an optimal plan, if one exists.

Proof: Each planning episode only examines nodes with estimated metric value—given by METRICBOUNDFN—that is less than *bestMetric*. By assumption 2, this is a finite set of nodes, so each episode must complete and the algorithm must eventually terminate. Now the result follows from Theorem 4.3. ■

In Theorem 4.4, condition 1 is satisfied by any implementation of the algorithm that uses a sufficiently large number for the initial value of *bestMetric*. Moreover, Theorem 4.1 shows how condition 3 can be satisfied, and Theorem 4.2 shows how condition 4 can be satisfied. Condition 2, however, can sometimes be falsified by a PDDL3 instance. In particular, the metric function can be defined in such a way that its value *improves* as the number of violated precondition preferences increases. Under such a metric function the plans' metric values might improve without bound as the plan length increases. This would mean that the number of plans with metric value less than the intitial bound, *bestMetric*, becomes unbounded, and condition 2 will be violated. We can avoid cases like this when the metric function is *bounded on precondition preferences*.

Definition 4.7 (BPP metrics) *Let the individual precondition preferences for a planning instance P be* Γ*, and let U denote the initial value of bestMetric. A metric function is* bounded on precondition preferences *(BPP) if there exists a value* r_i *for each precondition preference* $p_i \in \Gamma$ *such that in every node N with* METRICBOUNDFN$(N) < U$*,* p_i *is never violated more than* r_i *times.*

BPP metrics are such that the `is-violated` functions are always smaller than a fixed bound in every node with metric value lower than U. This property guarantees that there are only a finite number of plans with value less than U, and ultimately enables us to prove another optimality result:

Corollary 4.1 *Assume that the metric function for planning instance P is BPP and assume conditions 1, 3, and 4 in Theorem 4.4 hold. Then Algorithm 4.1 finds an optimal plan for P.*

Proof: We need only prove that the set of nodes N with METRICBOUNDFN$(N) <$ *bestMetric* is finite. This will satisfy condition 2 and allow us to apply Theorem 4.4. The BPP condition ensures that each

precondition function p_i in N can only have a value in the range 0–r_i (for some fixed value r_i). Since the precondition functions are the only functions in the planning instance (the remaining elements of the state are boolean predicates), this means that only a finite number of different states can have this property. ■

Note that the NDVPL property, which we could use to satisfy condition (4) in Theorem 4.4, *does not* imply necessarily the BPP property. As an example suppose a domain where `precPref` is a precondition preference, and `goalPref1` and `goalPref2` are final-state preferences. Assume we are using the B function as METRICBOUNDFN and that the metric for a node N is defined as:

$$M(N) = \texttt{is-violated}(\texttt{goalPref1}, N) * \texttt{is-violated}(\texttt{precPref}, N) + \texttt{is-violated}(\texttt{goalPref2}, N). \tag{4.2}$$

M is clearly NDVPL since it cannot decrease as plans violate more preferences. However, M does not necessarily *increase* as more preferences are violated, which can lead to situations in which we have an infinite set of goal nodes with the same metric value. Indeed, assume `goalPref2` is an unreachable preference that cannot be detected by the relaxed graph (i.e., it is such that it won't be detected by our B bounding function). Moreover, assume the planner has found a node that satisfies `goalPref1`. Assuming `precPref` can be violated by some action in the planning instance, there might be infinite plans that could be generated that violate `precPref` repeatedly while still satisfying `goalPref1`. Because the `is-violated` functions are represented within the state, those plans cannot be eliminated by the algorithm since they will not produce cycles.

The BPP and NDVPL properties are quite natural conditions on the metric function. Indeed, it is reasonable to assume that violated preferences are undesirable. Hence, a plan should become (arbitrarily) worse as the number of preferences it violates becomes (arbitrarily) larger. Such a property is sufficient to guarantee both the NDVPL and the BPP conditions. The *additive* family of metric functions satisfies both conditions, and it is defined as follows.

Definition 4.8 (Additive metric function) *A PDDL3 metric function is* additive*, if it has the form* $M = \sum_{i=0}^{n} c_i \times \texttt{is-violated}(p_i)$*, where* $c_i \geq 0$*.*

Remark 4.1 *Additive metric functions satisfy the NDVPL condition and satisfy the BPP condition when* METRICBOUNDFN *is either B or O.*

Additive metric functions were used in all of the problems in the qualitative preference track of IPC-5. Therefore, our algorithm—when using O or B for pruning—is guaranteed to find an optimal solution for these problems, given sufficient time and memory. In practice, however, due to restrictions of time and memory, the algorithm finds the optimal solution only in the most simple problems. On the other larger problems it returned the best plan its completed planning episodes found in the time alloted.

k-Optimality

Instead of searching for an optimal plan among the set of all valid plans, one might be interested in restricting attention to a subset of the valid plans. For example, there might be resource usage limitations that might further constrain the set of plans that one is willing to accept. This might be the case when a shift worker cannot be asked to work more than one overtime shift in three days, or a plane cannot log more than a certain number of continuous kilometers. If the set of plans one is interested in can be characterized by a temporally extended property, it suffices to add such a property to the set of hard constraints. The optimality results presented above, will allow the planner to find the optimal plan from among the restricted set of plans, regardless of the property used.

For some interesting properties, however, we can find optimal plans under weaker conditions on the metric function than those required in the general case above. This is the case, for example, when we are interested in plans whose length is bounded by a certain value.

Several existing preference planners are able to find plans that are optimal among the set of plans with restricted length or makespan. For example, PPLAN (Bienvenu *et al.*, 2006) when given a bound k is able to find an optimal plan among those with length k or less. Similarly, both the system by Brafman and Chernyavsky (2005) and SATPLAN-P (Giunchiglia and Maratea, 2007) return optimal plans among those plans of makespan n, where n is a parameter. It should be noted, however, that such plans need not be globally optimal. That is, there could be plans of longer length or makespan that have higher value than the plan returned by these systems. Our algorithm, on the other hand, can return the globally optimal plan under conditions described above. If we are interested, however, in plans of restricted length then our algorithm can return k-optimal plans under weaker conditions.

Definition 4.9 (k-optimal plan) *A plan is* k-optimal *iff it is the optimal among the set of plans of length* $i \leq k$.

To achieve k-optimality, we force the algorithm to search in the space of plans whose length is smaller than or equal to k, by imposing an additional hard constraint that restricts the length of the plan.

Theorem 4.5 *Assume Algorithm 4.1 uses sound pruning, and that the set of initial hard constraints contains the formula* $(\texttt{total-time}) \leq k$. *Then, the returned plan (if any) is* k*-optimal.*

Proof: Since the space of plans of length up to k is finite, each planning episode will terminate with an improved plan (if any exists). Because of sound pruning, no node can be wrongly pruned from the search space. Hence, the last returned plan (if any) is optimal. ■

Note that this result does not require restrictions on the metric function such as condition 2 in Theorem 4.4. Thus, this result is satisfied by a broader family of metric functions than those that satisfy Theorem 4.4; for example, it is satisfied when using NDVPL metrics such as the one in Equation 4.2.

4.5 Implementation and Evaluation

We have implemented our ideas in the planner HPLAN-P. HPLAN-P consists of two modules. The first is a preprocessor that reads PDDL3 problems and generates a planning problem with only simple preferences expressed as a TLPLAN domain. The second module is a modified version of TLPLAN that is able to compute the heuristic functions and implements the algorithm of Section 4.4.

Recall that two of the key elements in our algorithm are the iterative pruning strategy and the heuristics used for planning. In the following subsections we evaluate the effectiveness of our planner in obtaining good quality plans using several combinations of the heuristics. As a testbed, we use the problems of the qualitative preferences track of IPC-5, all of which contain TEPs. The IPC-5 domains are composed of two transportation domains: TPP and trucks, a production domain: openstacks, a domain which involves moving objects by using machines under several restrictions: storage, and finally, rovers, which models a rover that must move and collect experiments (for more details, we refer the reader to the IPC-5 booklet (Dimopolus, Gerevini, Haslum, and Saetti, 2006)). Each domain consists of 20 problems. The problems in the trucks, openstacks, and rovers domains have hard goals and preferences. The remaining problems have only preferences. Preferences in these domains impose interesting restrictions on plans, and usually there is no plan that can achieve them all.

At the end of the section, we compare our planner against the other planners that participated in IPC-5. The results are based on the data available from IPC-5 (Gerevini, Dimopoulos, Haslum, and Saetti, 2006) and our own experiments.

4.5.1 The Effect of Iterative Pruning

To evaluate the effectiveness of iterative pruning we compared the performance of three pruning functions: the optimistic metric (O), the best relaxed metric (B), and no pruning at all. From our experiments, we conclude that most of the time pruning can only produce better results than no pruning, and that, overall, pruning with B usually produces better results than pruning with O.

To compare the different strategies, we ran all IPC-5 problems with O and no pruning, with a 30-minute timeout. The heuristics used in these experiments were the four top-performing strategies on each domain, under pruning with B.

The impact of pruning varies across different domains. In three of the domains, the impact of pruning is little. In the storage and TPP domains, pruning has no effect, in practice. In the rovers domain, the impact is slim: O performs as good as B does, and no pruning, on average, produces solutions with a 0.05% increase on the metric. An increased impact is observed in the trucks domain, where the top-performing heuristics improve the metric of the first plan found by 30.60% under B pruning, while under O pruning the metric is improved by 28.02% on average, and under no pruning by 21.33% on average. Finally, the greatest impact can be observed on the openstacks domain. Here,

B produces 13.63% improvement on average, while both no pruning and pruning with *O* produce only 1.62% improvement.

In general, pruning has a noticeable impact when, during search, it can be frequently proven that certain preferences will *not* be satisfied. In the case of the openstacks domain for example, most preferences require certain products (which are associated with *orders*) to be *delivered*. On the other hand, the goal usually requires a number of orders to be *shipped*. To ship an order one is required to start the order, and then ship it. However, to deliver a product associated with order *o*, one needs to *make* the product after *o* has been started and before the *o* has been shipped. Thus, whenever an order *o* is shipped, the *B* function automatically regards as unsatisfiable all preferences that involved the delivery of an unmade product associated with *o*. This occurs frequently in the search for plans for this domain. The initial solution, which ignores preferences, produces a plan with no *make-product* actions. As the search progresses, states that finish an order early are constantly pruned away, which in turn favours adding *make-product* actions.

A side effect of pruning is that it can sometimes prove (when the conditions of Theorem 4.3 are met) that an optimal solution has been found. Indeed, the algorithm stops on most of the simplest problems across all domains (therefore, proving it has found an optimal plan). If no pruning was used the search would generally never terminate.

4.5.2 Performance of Heuristics

To determine the effectiveness of various prioritized heuristic sequences (Section 4.4.1) we compared 42 heuristic sequences using *B* as a pruning function, allowing the planner to run for 15 minutes over each of the 80 IPC-5 problem instances. All the heuristics had *G* as the highest priority (therefore, we omit *G* from their names). Specifically, we experimented with *O*, *B*, *OP*, *PO*, *BP*, *PB*, and $BD(r)$, $D(r)B$, $OD(r)$, $D(r)O$ for $r \in \{0, 0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9, 1\}$.

In general, we say that a heuristic is better than another if it produces plans with better quality, where quality is measured by the metric of the plans. To evaluate how good a heuristic is, we measure the percent improvement of the metric of the last plan found with respect to the metric of the first plan found. Thus, if the first plan found has metric 100, and the last has metric 20, the percent improvement is 80%. Since a first plan is always found using *G*, its metric value is always the same, regardless of the heuristic we choose. Hence this measure can be used to objectively compare performance.

Table 4.1 shows the best and worst performing heuristics in each of the domains tested. In many domains, several heuristics yield very similar performance. Moreover, we conclude that the heuristic functions that use the relaxed graph are key to good performance. In all problems, save TPP, the heuristics that used the relaxed graph had the best performance. The case of TPP is pathological in the qualitative preference track. However, upon looking at the actual plans traversed during the search we observed that it is not the case that *O* is a *good* heuristic for this problem, indeed *O* is almost totally blind

Domain	1 Plan	>1 Plan	Best heuristics	Worst heuristics
openstacks	18	14	BP[13.77], DO(1)[13.63], DB(1)[13.63], BD(1)[13.63], B[13.63]	D(0)B[7.56], for $r \in \{0.01, 0.05, 0.1\}$: DO($r$)[7.63] and DB($r$)[7.63]
trucks	5	4	D(0)O[30.68], OD(0)[30.68]	PB[5.35], OP[5.35], PO[5.35], O[12.02]
storage	16	9	BO[37], OB[37], B[37], O[37], BD(0.05)[35.62], OD(0.05)[35.55], BD(0)[35.42]	PO[21.04], PB[21.04], BP[24.18], OP[24.18]
rovers	11	9	D(0.1)O[17.15], D(0.1)B[17.15], D(0.3)B[16.91], D(0.3)O[16.91], O(0.01)D[16.47], O(0.05)D[16.47]	BP[6.97], OP[7.16], B[10.85], OB[10.85], BO[10.85], O[10.85]
TPP	20	20	O[40.32], BO[32.02], B[32.02], OB[33.97]	for $r \leq 0.9$: BD(r)[9.03], OD(0.9)[10.98]

Table 4.1: Performance of different heuristics in the problems of the *Qualitative Preferences* track of IPC-5. The second column shows the number of problems where at least one plan was found. The third, shows how many of these plans were subsequently improved upon by the planner. The average percent metric improvement wrt. the first plan found is shown in square brackets.

since in most states O is equal to 0. Rather, it turns out that heuristics based on the relaxed graph are *poor* in this domain, misguiding the search. In Section 4.6, we explain scenarios in which our heuristics can perform badly, and give more details on why TPP is one of these cases.

4.5.3 Comparison to Other Approaches

We entered HPLAN-P in the IPC-5 *Qualitative Preferences* track (Gerevini *et al.*, 2006), achieving 2nd place behind SGPlan_5 (Hsu *et al.*, 2007). Despite HPLAN-P's distinguished standing, SGPlan_5's performance was superior to HPLAN-P's, sometimes finding better quality plans, but generally solving more problems and solving them faster. SGPlan_5's superior performance was not unique to the preferences tracks. SGPlan_5 dominated all 6 tracks of the IPC-5 *satisficing planner* competition. As such, we conjecture that their superior performance can be attributed to the partitioning techniques they use, which are not specific to planning with preferences, and that these techniques could be combined with those of HPLAN-P. This is supported by the fact that HPLAN-P has similar or better performance than SGPlan_5 on simple planning instances, as we see in experiments shown at the end of this section.

HPLAN-P consistently performed better than MIPS-BDD (Edelkamp, Jabbar, and Naizih, 2006) and MIPS-XXL (Edelkamp, 2006b); HPLAN-P can usually find plans of better quality and solve many more problems. MIPS-BDD and MIPS-XXL use related techniques, based on propositional Büchi automata, to

handle LTL preferences. We think that part of our superior performance can be explained because our compilation does not ground LTL formulae, avoiding blowups, and also because the heuristics are easy to compute. For example, MIPS-XXL and MIPS-BDD were only able to solve the first two problems (the smallest) of the openstacks domain, whereas HPLAN-P could quickly find plans for almost all of them. In this domain the number of preferences was typically high (the third instance already contains around 120 preferences). On the other hand, something similar occurs in the storage domains. In this domain, though, there are many fewer preferences, but these are quantified. More details can be found on the results of IPC-5 (Gerevini *et al.*, 2006).

While we did not enter the *Simple Preferences* track, experiments performed after the competition indicate that HPLAN-P would have done well in this track. To perform a comparison, we ran our planner for 15 minutes[3] on the first 20 instances[4] of each domain. In Table 4.2, we show the performance of HPLAN-P's best heuristics compared to all other participants, in those domains on which all four planners solved at least one problem. HPLAN-P was able to solve 20 problems in all domains, except trucks, where it could only solve the 5 simpler instances (see Table 4.3 for details on the trucks domain). In the table, #S is the number of problems solved by each approach, and *Ratio* is the average ratio between the metric value obtained by the particular planner and the metric obtained by our planner. Thus, values over 1 indicate that our planner is finding better plans, whereas values under 1 indicate the opposite. The results for HPLAN-P were obtained on an Intel(R) Xeon(TM) CPU 2.66GHz machine running Linux, with a timeout of 15min. Results for other planners were extracted from the IPC-5 official results, which were generated on a Linux Intel(R) Xeon(TM) CPU 3.00GHz machine, with a 30 min. timeout. Memory was limited to 1GB for all processes.

We conclude that SGPlan_5 typically outperforms HPLAN-P. SGPlan_5, on average, obtains plans that are no more than 25% better in terms of metric value than those obtained by HPLAN-P. Moreover, in the most simple instances usually HPLAN-P does equally well or better than SGPlan_5 (see Table 4.3). HPLAN-P can solve more instances than those solved by *Yochan*$^{\mathcal{PS}}$, MIPS-XXL and MIPS-BDD. Furthermore, it outperforms *Yochan*$^{\mathcal{PS}}$ and MIPS-XXL in terms of achieved plan quality. HPLAN-P's performance is comparable to that of MIPS-BDD in those problems that can be solved by both planners. Finally, we again observed that the best-performing heuristics in domains other than TPP are those that use the relaxed graph, and, in particular, the *D* heuristic.

We ran a final comparison between SGPlan_5 and HPLAN-P on the openstacks-nce domain (Haslum, 2007). openstacks-nce is a re-formulation of the original openstacks simple-preferences domain that does not include actions with conditional effects. These two domains are essentially equivalent in the sense that plans in one domain have a corresponding plan with equal quality in the other. The results are shown in Table 4.4. We observe that HPLAN-P consistently outperforms SGPlan_5 across all instances

[3] In IPC-5, planners where given 30 min. on a similar machine.

[4] Only the pathways domain has more than 20 problems.

Domain	HPLAN-P		SGPlan$_5$		*Yochan*$^{\mathcal{PS}}$		MIPS-BDD		MIPS-XXL	
	#S	Ratio	#S	Ratio	#S	Ratio	#S	Ratio	#S	Ratio
TPP	20	1	20	.78–.8	11	1.02–1.07	9	0.94–0.99	9	1.68–1.78
openstacks	20	1	20	.89–.92	*	*	2	2.5	18	6.45–6.81
storage	20	1	20	.74–.76	5	3.86–3.95	4	1	4	15.41
pathways	20	1	20	.77	4	1.02	10	0.79	16	1.19–1.21

Table 4.2: Relative performance of HPLAN-P's best heuristics for simple preferences, compared to other IPC-5 participants. *Ratio* compares the performance of the particular planner and HPLAN-P's. Ratio > 1 means HPLAN-P is superior, and Ratio < 1 means otherwise. #S is the number of problems solved. "*" means the planner did not compete in the domain.

of this domain, obtaining plans that are usually at least 50% better in quality. We also observe that the performance of HPLAN-P is consistent across the two formulations, which is not the case with SGPlan$_5$.

4.6 Discussion

In previous sections, we proposed a collection of heuristics that can be used in planning with TEPs and simple preferences in conjunction with our incremental search algorithm. In our experimental evaluation we saw that in most domains the heuristics that utilize the relaxed planning graph are those that provide the best performance. Given the limited number of domains in which we have had the opportunity to test the planner, it is hard—and might be even be impossible—to conclude which is the best combination of heuristics to use. It is even hard to give a justified recipe for their use. However, some situations in which our heuristics perform poorly can be identified and analyzed. Below we describe two reasons for potential poor performance.

The first reason for potentially poor performance is due to our choice of using prioritized sequences of heuristics. We have chosen the goal distance G to appear as the first priority to guide the planner towards satisfying the must-achieve goals for a pragmatic reason: the goal is the most important thing to achieve. However, this design decision sometimes makes the search algorithm focus excessively on goal achievement to the detriment of preference satisfaction. This issue becomes particularly relevant when there are interactions between the goal and the preferences. Consider, for example, a situation in which a preference p can *only* be achieved *after* achieving the goal. Furthermore, assume the goal g is the conjunction $f_1 \wedge f_2$, and assume that prior to achieving p one has to make f_2 false. In cases like this, after the algorithm finds a plan for the goal, it can hardly find a plan that also satisfies p. When extending any plan for g, the planner will always choose an action that does not invalidate the subgoal f_2 over an action that invalidates f_2, if such an action is available. This is because the goal distance (G) of any search node in which f_2 is false is strictly greater than the goal distance in which both f_1 and f_2 are true. As a consequence, the algorithm will have trouble achieving p, and actually will only achieve

Instance	*Yochan*^{PS}	MIPS-BDD	MIPS-XXL	SGPlan$_5$	HPLAN-P			
					O	OD(r=0.5)	OD(r=0)	OD(r=1)
TPP-01	22	16	16	16	16	16	16	16
TPP-02	36	24	24	24	24	24	24	24
TPP-03	24	29	29	29	29	29	29	29
TPP-04	45	35	35	35	39	35	35	42
TPP-05	103	89	223	79	103	79	87	105
TPP-06	133	110	275	101	120	118	114	120
TPP-07	124	126	322	100	124	135	135	135
openstacks-01	*	12	63	13	6	6	6	6
openstacks-02	*	12	63	16	4	4	4	4
openstacks-03	*	ns	88	12	36	30	36	30
openstacks-04	*	ns	98	26	47	44	45	49
openstacks-05	*	ns	133	36	25	21	25	21
openstacks-06	*	ns	133	33	21	18	21	18
openstacks-07	*	ns	285	67	87	74	87	74
trucks-01	0	0	0	1	0	0	0	0
trucks-02	3	0	0	0	0	0	0	0
trucks-03	0	0	0	0	0	0	0	0
trucks-04	0	0	ns	0	3	1	3	4
trucks-05	1	ns	ns	0	0	0	0	0
storage-01	6	3	18	5	3	3	3	3
storage-02	11	5	37	8	5	5	5	5
storage-03	49	6	158	14	6	6	6	6
storage-04	51	9	197	17	9	9	9	9
storage-05	165	ns	ns	87	97	130	130	97
storage-06	ns	ns	ns	124	161	195	195	161
storage-07	ns	ns	ns	160	274	281	307	274
pathways-01	2	2	3	2	2	2	2	2
pathways-02	3	3	5	3	3	4	4	4
pathways-03	3	3	4.7	3	3	3.7	3.7	3.7
pathways-04	3	2	3	2	2	2	2	2
pathways-05	ns	7	10.2	6.5	8.5	9	10.2	10.2
pathways-06	ns	8	12.9	10	12.9	12.9	12.9	12.9
pathways-07	ns	11	12.5	8	12.5	12.5	12.5	12.5

Table 4.3: Plan quality (metric) of three of HPLAN-P's heuristics compared to the IPC-5 *Simple Preferences* participants on the simpler, non-metric problems. "ns" means that the instance what not solved by the planner. "*" means the planner did not compete in the domain.

Instance	openstacks-nce					openstacks				
	$SGPlan_5$	HPLAN-P				$SGPlan_5$	HPLAN-P			
		O	OD(.5)	OD(0)	OD(1)		O	OD(.5)	OD(0)	OD(1)
01	70	11	11	11	11	13	6	6	6	6
02	70	7	11	7	11	16	4	4	4	4
03	90	38	42	37	41	12	36	30	36	30
04	100	48	49	46	49	26	47	44	45	49
05	140	48	48	48	48	36	25	21	25	21
06	140	35	41	34	41	33	21	18	21	18
07	300	98	98	98	98	67	87	74	87	74
08	620	140	152	148	148	123	86	78	86	78
09	620	154	155	154	154	121	109	123	109	123
10	120	30	25	30	20	20	19	11	10	13
11	120	36	26	36	22	21	19	22	23	12
12	153	80	81	80	73	23	52	45	45	51
13	223	190	172	181	174	48	171	167	167	167
14	65	47	22	47	24	6	32	23	21	21
15	210	125	123	125	126	0	74	67	67	67
16	210	133	133	133	133	0	74	63	67	63
17	450	224	255	269	254	0	209	179	179	180
18	930	588	558	929	557	0	557	464	464	493
19	1581	1581	1581	1581	1581	254	1581	1581	1581	1581
20	1348	1348	1348	1348	1348	424	1348	1348	1348	1348
	openstacks-nce					openstacks				

Table 4.4: Metric values obtained by four of HPLAN-P's heuristics and $SGPlan_5$ on the openstacks and openstacks-nce(Haslum, 2007) domains.

p when extending a plan for g when *no actions* that invalidate f_2 are available. Unfortunately the only way of getting into such a situation implies exhausting the search space of plans that extend a plan for g without invalidating g.

The second source for poor performance is the loss of structure in which we incur by computing our heuristic in a planning instance in which the action's deletes (i.e., negative effects) are ignored. The inaccurate reachability information provided by this relaxation might significantly affect the performance of all our heuristics based on the relaxed planning graph (i.e., P, B, and D). Consider for example an instance in which there are no hard goals and there are two preferences, p_1 and p_2. Assume further that p_2 is a preference that is rather easy to achieve from any state but that has to be violated in order to achieve p_1. Assume that we are in a state in which p_2 is satisfied but p_1 is not, and in which we need to perform at least three actions to achieve both p_1 and p_2. Let those actions be a, b, and c, such that a makes p_2 false and p_1 true, and finally action b followed by c reestablish p_1, as shown in Figure 4.3. Moreover, assume that action e is applicable in s, and that it leads to s_2—a state from which p_1 and p_2 can be reached by the same sequence of three actions. Because the D heuristic is computed on the delete relaxation, D will always prefer to expand s_2 instead of s_1. A relaxed solution on s_2 may achieve both preferences at depth 1, since the preference p_2 is already satisfied at depth 0. On the other hand, a relaxed solution on s_1 may achieve both preferences at depth 2, since in s_1 two actions are needed to reestablish p_2. Once the algorithm expands s_2, there could be another action applicable in s_2, analogous to e, that would steer the search away from s_3.

It is precisely a situation similar to that described above that makes the heuristics based on the relaxed graph (especially D and P), perform poorly in the TPP domain. TPP is a transportation problem in which trucks can move between markets and depots transporting goods. A good can be put into the truck by performing a *load* followed by a *store*. Stored goods can be unloaded from the truck performing an *unload*. Once in a market, one has to *buy* an object before it becomes ready to load. In problems of the TPP domain there is a preference that states that any good must be eventually loaded on some truck (p_1). On the other hand, there is a preference that states that all trucks should be unloaded at the end of the plan (p_2). Once we have considered moving a truck to a market and bought a certain good, say `good1`, our plan prefix has achieved p_2 but not p_1. A reasonable course of action to achieve both preferences would be to *load* `good1` on the truck, followed by a *store*, and followed by an *unload*. However, the state that results from performing a *load* is never preferred by the planner, since just like in Figure 4.3, a *load* invalidates p_2 while making p_1 true. Instead, an action that preserves the p_2 property (e.g., a *buy* of another good) is always preferred. This leads the planner to consider all possible combinations of sequences that *buy* a good before considering a *load*. Even worse, after performing all possible buys, for a similar reason the search prefers to use other truck to move to another market to keep on buying products.

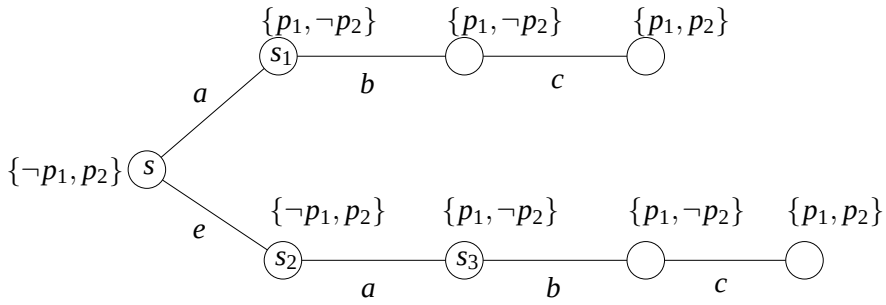


Figure 4.3: A situation in which our *D* heuristics prefers a node that does not lead to the quick satisfaction of both p_1 and p_2.

4.7 Related Work

There is a significant amount of work on planning with preferences that is related, in varying degrees, to the method we have presented here. We organize this work into two groups: first, planners that are able to plan with preferences in non-PDDL3 preference languages or using soft goals; second, work that focuses on the PDDL3 language. In the rest of the section we review the literature in these two categories.

4.7.1 Other Preference Languages

PPLAN (Bienvenu *et al.*, 2006) is a plannning system that exploits progression to plan directly with TEPs using heuristic search. In contrast to HPLAN-P, which is incremental, PPLAN always returns an optimal plan whose length is bounded by a plan-length parameter (i.e., it is *k*-optimal). Unfortunately, PPLAN uses an admissible heuristic that is far less informative than the heuristics proposed here. As such, it is far less efficient. The heuristic in PPLAN is similar to our *O* heuristic, and thus does not provide an estimate of the cost to achieving unsatisfied preferences. PPLAN was developed prior to the definition of PDDL3 and exploits its own *qualitative* preference language, *LPP*, to define preferences. *LPP* supports rich TEPs, including nested LTL formulae (unlike PDDL3) and rather than specifying a metric objective function, the *LPP* objective is expressed as a logical formula. PPLAN's *LPP* language is an extension and improvement over the *PP* language proposed by Son and Pontelli (2004).

The HPLAN-QP planner (Baier and McIlraith, 2007) was proposed as an answer to some of the shortcomings of PPLAN. It is an extension to the HPLAN-P system, allowing planning for *qualitative* TEPs guided by heuristics similar to those that have been proposed in this chapter. The preference language is based on *LPP*, the language used by PPLAN. HPLAN-QP guides the search actively towards satisfaction of preferences (unlike PPLAN), and like HPLAN-P, guarantees optimality of the last plan found given sufficient resources.

Also related is the work on *partial satisfaction planning problems* (PSPs) (over-subscription planning) (van den Briel, Nigenda, Do, and Kambhampati, 2004; Smith, 2004). PSPs can be understood as a planning problem with no hard goals but rather a collection of soft goals each with an associated utility; actions also have costs associated with them. Some existing planners for PSPs (Sanchez and Kambhampati, 2005; Do, Benton, van den Briel, and Kambhampati, 2007) are also incremental and use pruning techniques. However in general, they do not offer any optimality guarantees. Recently, Benton, van den Briel, and Kambhampati (2007) developed an incremental planner, BBOP-LP, that uses branch-and-bound pruning for PSP planning, similar to our approach. BBOP-LP is able to offer optimality guarantees given sufficient resources. However, in contrast to HPLAN-P, it uses very different techniques for obtaining the heuristics. To compute heuristics it first relaxes the original planning problem and creates an integer programming (IP) model of this new problem. It then computes heuristics from a linear-programming relaxation of the IP model. Lastly, Feldmann, Brewka, and Wenzel (2006) propose a planner for PSPs that iteratively invokes METRIC-FF to find better plans.

Bonet and Geffner (2006) have proposed a framework for planning with action costs and costs/rewards associated with fluents. Their cost model can represent PSPs as well as the simple preferences subset of PDDL3. They propose admissible heuristics and an optimal algorithm for planning under this model. Heuristics are obtained by compiling a relaxed instance of the problem to d-DNNF, while the algorithm is a modification of A^*. The approach does not scale very well for large planning instances, in part because of its need to employ an admissible heuristic.

Finally, there has been work that casts the preference-based planning problem as an answer set programming problem (ASP), as a constraint satisfaction problem (CSP), and as a satisfiability (SAT) instance. The paper by Son and Pontelli (2004) proposed one of the first languages for preference-based planning, *PP*, and cast the planning problem as an optimization of an ASP problem. Their *PP* language includes TEPs expressed in LTL. Brafman and Chernyavsky (2005) proposed a CSP approach to planning with final-state qualitative preferences specified using TCP-nets. Additionally, Giunchiglia and Maratea (2007) proposed a compilation of preference-based planning problems into SAT. None of these approaches exploits heuristic search and thus are fundamentally different form the approach proposed here. The latter two approaches guide the search for a solution by imposing a variable/value ordering that will attempt to produce preferred solutions first. Because these works are recasting the problem into a different formalism, they explore a very different search space than our approach. Note also that the conversion to ASP, CSP or SAT requires assuming a fixed bound on plan length limiting the approach to at best finding k-optimal plans.

4.7.2 IPC-5 competitors

Most related to our work are the approaches taken by the planners that competed in IPC-5, both because they used the PDDL3 language and because many used some form of heuristic search. *Yochan*$^{\mathcal{PS}}$

(Benton, Kambhampati, and Do, 2006) is a heuristic planner for simple preferences based on the Sapa^{ps} system (van den Briel *et al.*, 2004). Our approach is similar to theirs in the sense that both use a relaxed graph to obtain a heuristic estimate. *Yochan*$^{\mathcal{PS}}$ is also an incremental planner, employing heuristics geared towards classical goals. However, to compute its heuristic for a given state, it explicitly selects a subset of preferences to achieve from that state and then treats this subset as a classical goal. This process can be very costly in the presence of many preferences.

MIPS-XXL (Edelkamp *et al.*, 2006) and MIPS-BDD (Edelkamp, 2006b) both use Büchi automata to plan with temporally extended preferences. While the approach to compiling away the TEPs also constructs an automata (as in our approach), their translation process generates grounded preference formulae. This makes the translation algorithm prone to unmanageable blow-up. Further, the search techniques used in both of these planners are quite different from those we exploit. MIPS-XXL iteratively invokes a modified METRIC-FF (Hoffmann, 2003) forcing plans to have decreasing metric values. MIPS-BDD, on the other hand, performs a cost-optimal breath-first search that does not employ a heuristic.

Finally, the winner of the preferences tracks at IPC-5, SGPlan$_5$ (Hsu *et al.*, 2007), uses a completely different approach. It partitions the planning problem into several subproblems. It then uses a modified version of FF to solve those subproblems and finally integrates these sub-solutions into a solution for the entire problem. During the integration process it attempts to minimize the metric function. SGPlan$_5$ is not incremental, and seems to suffer from some non-robustness in its performance as shown by the results given in Table 4.4 (where its performance on an reformulated but equivalent domain changes quite dramatically).

4.8 Conclusions and Future Research

In this chapter we have presented a new technique for planning with preferences that can deal with simple preferences, temporally extended preferences, and hard constraints. The core of the technique, our new set of heuristics and incremental search algorithm, are both amenable to integration with a variety of classical and simple-preference planners. The compilation technique for converting TEPs to simple preferences can also be made to work with other planners, although the method of embedding the constructed automata we utilize here might need some modification, dependent on the facilities available in that planner. Our method of embedding the constructed automata utilized TLPLAN's ability to deal with numeric functions and quantification. In particular, TLPLAN's ability to handle quantification allowed us to utilize the parameterized representation of the preferences generated by the compilation, leading to a considerably more compact domain encoding.

We have presented a number of different heuristics for planning with preferences. These heuristics have the feature that some of them account for the value that could be achieved from unsatisfied pref-

erences, while others account for the difficulty of actually achieving these preferences. Our method for combining these different types of guidance is quite simple (tie-breaking), and more sophisticated combinations of these or related heuristics could be investigated. More generally, the question of identifying the domain features for which particular heuristics are most suitable is an interesting direction for future work.

We have also presented an incremental best-first search planning algorithm. A key feature of this algorithm is that it can use heuristic bounding functions to prune the search space during its incremental planning episodes. We have proved that under some fairly natural conditions our algorithm can generate optimal plans. It is worth noting that these conditions do not require the algorithm to utilize admissible heuristics. Nor do they require imposing a priori restrictions on the plan size (length or makespan) which would allow the algorithm to only achieve *k*-optimality rather than global optimality.

The algorithm can also employ different heuristics in each incremental planning episode, something we exploit during the very first planning episode by ignoring the preferences and only asking the planner to search for a plan achieving the goals. The motivation for this is that we want at least one working plan in hand before trying to find a more preferred plan. In our experiments, however, the remaining planning episodes are all executed with one fixed heuristic. More flexible schedules of heuristics could be investigated in future work.

We have implemented our method by extending the TLPLAN planning system and have performed extensive experiments on the IPC-5 problems to evaluate the effectiveness of our heuristic functions and search algorithm. While no heuristic dominated all test cases, several clearly provided superior guidance towards good solutions. In particular, those that use the relaxed graph in some way proved to be the most effective in almost all domains. Experiments also confirmed the essential role of pruning when solving large problems. HPLAN-P scales better than many other approaches to planning with preferences, and we attribute much of this superior performance to the fact that we do not ground our planning problems.

Although the proposed heuristics perform reasonably well in many of the benchmarks we have tested, we have identified cases in which they perform poorly. In some cases, computing heuristics over the delete relaxation can provide bad guidance in the presence of preferences. The resolution of some of the issues we have raised above open interesting avenues for future research.

The ideas presented in this chapter have been used to build other planning systems. As we mentioned above, Baier and McIlraith (2007) have extended HPLAN-P to plan for a *qualitative* preference language LPP. Recently, Sohrabi, Baier, and McIlraith (2009), have used the reformulation technique presented in this chapter to build a heuristic preference-based Hierarchical Task Network (HTN) (Erol, Hendler, and Nau, 1994) planner.

Chapter 5

Golog Domain Control Knowledge in State-of-the-Art Planners

5.1 Introduction

In previous chapters we have focused our attention on the problems of planning with temporally extended goals and temporally extended preferences. Our goal has been the exploitation of state-of-the-art techniques to achieve effective planning for these compelling non-classical planning tasks.

Another compelling planning technique is the use of DCK. DCK imposes domain-specific constraints on the definition of a valid plan. As such, it can be used to impose restrictions on the course of action that achieves the goal. While DCK sometimes reflects a user's desire to achieve the goal a particular way, it is most often constructed to aid in plan generation by reducing the plan search space. Moreover, if well-crafted, DCK can eliminate those parts of the search space that necessitate backtracking. In such cases, DCK together with blind search can yield valid plans significantly faster than state-of-the-art planners that do not exploit DCK. Indeed most planners that exploit DCK, such as TLPLAN (Bacchus and Kabanza, 1998) or TALPLAN (Kvarnström and Doherty, 2000), do little more than blind depth-first search with cycle checking in a DCK-pruned search space. Since most DCK reduces the search space but still requires a planner to backtrack to find a valid plan, it should prove beneficial to exploit better search techniques.

In this chapter we explore ways in which state-of-the-art planning techniques and existing state-of-the-art planners can be used in conjunction with DCK, with particular focus on *procedural* DCK. Procedural DCK (as used in HTN (Nau, Cao, Lotem, and Muñoz-Avila, 1999) or Golog (Levesque *et al.*, 1997)) is action-centric. It is much like a programming language, and often times like a plan skeleton or template. It can (conditionally) constrain the order in which domain actions should appear

in a plan.

As a simple example of procedural DCK, consider the `trucks` domain of the 5th International Planning Competition, where the goal is to deliver packages between certain locations using a limited-capacity truck. When a package reaches its destination it must be delivered to the customer. We can write simple and natural procedural DCK that significantly improves the efficiency of plan generation for instance:

> *Repeat the following until all packages have been delivered: Unload everything from the truck, and, if there is any package in the current location whose destination is the current location, deliver it. After that, if any of the local packages have destinations elsewhere, load them on the truck while there is space. Drive to the destination of any of the loaded packages. If there are no packages loaded on the truck, but there remain packages at locations other than their destinations, drive to one of these locations.*

Although procedural DCK is interesting in its own right as a planning tool, there are other very interesting applications, not strictly considered as"pure" planning applications, in which procedural control plays a fundamental role. A relevant application is agent programming; in particular, cognitive robotics (Levesque and Lakemeyer, 2007). Here the objective is to program agents with *flexible* programs. Programs are flexible in the sense that the agent may adjust, complete, or customize its execution based on its current goals, its knowledge and beliefs, and the state of the environment. Golog is one of the prominent languages used by this community, and has been used in some notable applications such as the Minerva Museum Tour Robot (Thrun, Bennewitz, Burgard, Cremers, Dellaert, Fox, Hähnel, Rosenberg, Roy, Schulte, and Schulz, 1999), Robocup (Ferrein, Fritz, and Lakemeyer, 2005), and recently also Robocup@Home.[1] By reformulating a problem with procedural Golog DCK into a classical planning problem we bring planning advances closer to the area of cognitive robotics, and thus potentially improve the performance of a broad range of applications. Claßen, Eyerich, Lakemeyer, and Nebel (2007) have made steps in connecting Golog and state-of-the-art planners but, as we discuss later in the chapter, their work is quite different from ours.

WSC is also another motivation for dealing with this problem. As seen in the first chapter, applications such as WSC require the plans that are complex structures (with loops and conditional constructs), there are many other applications. Notwithstanding, current planning technology has not reached the point at which complex plans with loops can be generated for a broad set of planning domains (for an up-to-date report, see e.g. Levesque, 2005). It has therefore been argued that a reasonable solution for problems such as WSC is the generation of plans by computing an execution of a human-generated procedure (or program) (McIlraith *et al.*, 2001), which essentially acts as DCK. These programs, however, contain significant non-determinism, which has to be resolved by the planner. Resolving these non-

[1] Personal communication with Gerhard Lakemeyer.

deterministic segments can be quite challenging. Thus, for those tasks, we are interested in leveraging the power of state-of-the-art techniques.

5.1.1 Contributions

The main contributions of this chapter follow.

1. As we mentioned above, DCK that has been used in planning is either state-centric (e.g. LTL), or based on HTNs. HTNs do not provide programming language constructs.

   We propose a language for DCK based on Golog that includes typical programming languages constructs such as conditionals and iteration as well as nondeterministic choice of actions in places where control is not germane. We argue that these action-centric constructs provide a natural language for specifying DCK for planning. We contrast them with DCK specifications based on LTL which are state-centric and though still of tremendous value, arguably provide a less natural way to specify DCK. We specify the syntax for our language as well as a PDDL-based semantics following Fox and Long (2003).

   An immediate advantage of our semantics is that it can be used to implement native Golog control support in any forward-chaining planner.

2. With a well-defined procedural DCK language in hand, we examine how to use state-of-the-art planning techniques together with DCK. Of course, most state-of-the-art planners are unable to exploit DCK. As such, we present an algorithm that translates a PDDL2.1-specified ADL planning instance and associated procedural DCK into an equivalent, program-free PDDL2.1 instance whose plans provably adhere to the DCK. Any PDDL2.1-compliant planner can take such a planning instance as input to their planner, generating a plan that adheres to the DCK.

3. Since they were not designed for this purpose, existing state-of-the-art planners may not exploit techniques that optimally leverage the DCK embedded in the planning instance. As such, we investigate how state-of-the-art planning techniques, rather than planners, can be used in conjunction with our compiled DCK planning instances. In particular, we propose domain-independent search heuristics for planning with our newly-generated planning instances. We examine three different approaches to generating heuristics, and evaluate them on three domains of the 5th International Planning Competition. Our results show that procedural DCK improves the performance of state-of-the-art planners, and that our heuristics are sometimes key to achieving good performance.

5.1.2 Outline

The rest of the chapter is organized as follows. Section 5.2 presents background on PDDL necessary for the rest of the chapter. Section 5.3 presents our procedural control language. Section 5.4 presents a procedure to compile Golog control into PDDL. Then, in section 5.5 we show how our compiled theory can be integrated with state-of-the-art planners. Section 5.6 presents an experimental analysis showing the benefits of our approach. Finally, Section 5.7 summarizes the chapter and discusses related work.

5.2 Background

5.2.1 A Subset of PDDL 2.1

In PDDL, a *planning instance* is a pair $I = (D, P)$, where D is a domain definition and P is a problem. To simplify notation, we assume that D and P are described in an ADL subset of PDDL. The difference between this ADL subset and PDDL 2.1 is that no concurrent or durative actions are allowed.

Following convention, domains are tuples of finite sets $(PF, Ops, Objs_D, T, \tau_D)$, where PF defines domain predicates and functions, Ops defines operators, $Objs_D$ contains domain objects, T is a set of types, and $\tau_D \subseteq Objs_D \times T$ is a type relation associating objects to types. An operator (or action schema) is also a tuple $\langle O(\vec{x}), \vec{t}, Prec(\vec{x}), Eff(\vec{x})\rangle$, where $O(\vec{x})$ is the unique operator name and $\vec{x} = (x_1, \ldots, x_n)$ is a vector of variables. Furthermore, $\vec{t} = (t_1, \ldots, t_n)$ is a vector of types. Each variable x_i ranges over objects associated with type t_i. Moreover, $Prec(\vec{x})$ is a boolean formula with quantifiers (BFQ) that specifies the operator's preconditions. BFQs are defined inductively as follows. Atomic BFQs are either of the form $t_1 = t_2$ or $R(t_1, \ldots, t_n)$, where t_i ($i \in \{1, \ldots, n\}$) is a term (i.e. either a variable, a function literal, or an object), and R is a predicate symbol. If φ is a BFQ, then so is $Qx\text{-}t\,\varphi$, for a variable x, a type symbol t, and $Q \in \{\exists, \forall\}$. BFQs are also formed by applying standard boolean operators over other BFQs. Finally $Eff(\vec{x})$ is a list of conditional effects, each of which can be in one of the following forms:

$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow R(\vec{x}, \vec{y}), \tag{5.1}$$

$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow \neg R(\vec{x}, \vec{y}), \tag{5.2}$$

$$\forall y_1\text{-}t_1 \cdots \forall y_n\text{-}t_n.\, \varphi(\vec{x}, \vec{y}) \Rightarrow f(\vec{x}, \vec{y}) = obj, \tag{5.3}$$

where φ is a BFQ whose only free variables are among $\vec{x}$ and $\vec{y}$, R is a predicate, f is a function, and obj is an object After performing a ground operator – or *action* – $O(\vec{c})$ in a certain state s, for all tuples of objects that may instantiate $\vec{y}$ such that $\varphi(\vec{c}, \vec{y})$ holds in s, effect (5.1) (resp. (5.2)) expresses that $R(\vec{c}, \vec{y})$ becomes true (resp. false), and effect (5.3) expresses that $f(\vec{c}, \vec{y})$ takes the value obj. As usual, states are represented as finite sets of atoms (ground formulae of the form $R(\vec{c})$ or of the form $f(\vec{c}) = obj$).

Planning problems are tuples $(Init, Goal, Objs_P, \tau_P)$, where $Init$ is the initial state, $Goal$ is a sentence with quantifiers for the goal, and $Objs_P$ and τ_P are defined analogously as for domains.

Semantics: Fox and Long (2003) have given a formal semantics for PDDL 2.1. In particular, they define when a sentence is *true* in a state and what *state trace* is the result of performing a set of *timed actions*. A state trace intuitively corresponds to an execution trace, and the sets of timed actions are ultimately used to refer to plans. In the ADL subset of PDDL2.1, since there are no concurrent or durative actions, time does not play any role. Hence, state traces reduce to sequences of states and sets of timed actions reduce to sequences of actions.

Building on Fox and Long's semantics, we assume that $\models$ is defined such that $s \models \varphi$ holds when sentence φ is true in state s. Moreover, for a planning instance I, we assume there exists a relation *Succ* such that $Succ(s,a,s')$ iff s' results from performing an executable action a in s. Finally, a sequence of actions $a_1 \cdots a_n$ is a plan for I if there exists a sequence of states $s_0 \cdots s_n$ such that $s_0 = Init$, $Succ(s_i, a_{i+1}, s_{i+1})$ for $i \in \{0, \ldots, n-1\}$, and $s_n \models Goal$.

5.3 A Language for Procedural Control

In contrast to state-centric languages, that often use LTL-like logical formulae to specify properties of the states traversed during plan execution, procedural DCK specification languages are predominantly action-centric, defining a plan template or skeleton that dictates *actions* to be used at various stages of the plan.

Procedural control is specified via *programs* rather than logical expressions. The specification language for these programs incorporates desirable elements from imperative programming languages such as iteration and conditional constructs. However, to make the language more suitable to planning applications, it also incorporates nondeterministic constructs. These elements are key to writing flexible control since they allow programs to contain missing or open program segments, which are filled in by a planner at the time of plan generation. Finally, our language also incorporates property testing, achieved through so-called *test actions*. These actions are not real actions, in the sense that they do not change the state of the world, rather they can be used to specify properties of the states traversed while executing the plan. By using test actions, our programs can also specify properties of executions similarly to state-centric specification languages.

The rest of this section describes the syntax and semantics of the procedural DCK specification language we propose to use. We conclude this section by formally defining what it means to plan under the control of such programs.

5.3.1 Syntax

The language we propose is based on Golog (Levesque *et al.*, 1997), a robot programming language developed by the cognitive robotics community. In contrast to Golog, our language supports specification of types for program variables, but does not support procedures.

Programs are constructed using the implicit language for actions and boolean formulae defined by a particular planning instance I. Additionally, a program may refer to variables drawn from a set of program variables V. This set V will contain variables that are used for nondeterministic choices of arguments. In what follows, we assume $\mathcal{O}$ denotes the set of operator names from *Ops*, fully instantiated with objects defined in I or elements of V.

The set of programs over a planning instance I and a set of program variables V can be defined by induction. In what follows, assume ϕ is a boolean formula with quantifiers on the language of I, possibly including terms in the set of program variables V. Atomic programs are as follows.

1. *nil*: Represents the empty program.
2. o: Is a single operator instance, where $o \in \mathcal{O}$.
3. **any**: A keyword denoting "any action".
4. ϕ?: A *test action*.

If σ_1, σ_2 and σ are programs, so are the following:

1. $(\sigma_1;\sigma_2)$: A sequence of programs.
2. **if** ϕ **then** σ_1 **else** σ_2: A conditional sentence.
3. **while** ϕ **do** σ: A while-loop.
4. σ^*: A nondeterministic iteration.
5. $(\sigma_1|\sigma_2)$: Nondeterministic choice between two programs.
6. $\pi(x\text{-}t)\,\sigma$: Nondeterministic choice of variable $x \in V$ of type $t \in T$.

Before we formally define the semantics of the language, we show some examples that give a sense of the language's expressiveness and semantics.

- **while** $\neg clear(B)$ **do** $\pi(b\text{-}block)\, putOnTable(b)$: while B is not clear choose any b of type block and put it on the table.
- **any***; $loaded(A, Truck)$?: Perform any sequence of actions until A is loaded in *Truck*. Plans under this control are such that $loaded(A, Truck)$ holds in the final state.
- $(\,load(C,P); fly(P,LA)\,|\,load(C,T); drive(T,LA)\,)$: Either load C on the plane P or on the truck T, and perform the right action to move the vehicle to LA.

5.3.2 Semantics

The problem of planning for an instance I under the control of program σ corresponds to finding a plan for I that is also an execution of σ from the initial state. In the rest of this section we define what those legal executions are. Intuitively, we define a formal device to check whether a sequence of actions $\vec{a}$ corresponds to the execution of a program σ. The device we use is a nondeterministic finite state automaton with ε-transitions (ε-NFA).

For the sake of readability, we remind the reader that ε-NFAs are like standard nondeterministic automata except that they can transition without reading any input symbol, through the so-called ε-transitions. ε-transitions are usually defined over a state of the automaton and a special symbol ε, denoting the empty symbol.

An ε-NFA $A_{\sigma,I}$ is defined for each program σ and each planning instance I. Its alphabet is the set of operator names, instantiated by objects of I. Its states are *program configurations* which have the form $[\sigma, s]$, where σ is a program and s is a planning state. Intuitively, as it reads a word of actions, it keeps track, within its state $[\sigma, s]$, of the part of the program that remains to be executed, σ, as well as the current planning state after performing the actions it has read already, s.

Formally, $A_{\sigma,I} = (Q, \mathcal{A}, Tr, q_o, F)$, where Q is the set of program configurations, the alphabet $\mathcal{A}$ is a set of domain actions, the transition function is $Tr : Q \times (\mathcal{A} \cup \{\varepsilon\}) \rightarrow 2^Q$, $q_0 = [\sigma, Init]$, and F is the set of final states.

Our definition of *Tr* closely follows the definition of *Trans* and *Final* from Golog's transition semantics (De Giacomo, Lespérance, and Levesque, 2000).

The transition function *Tr* is defined as follows for atomic programs.

$$Tr([a,s],a) = \{[nil,s']\} \quad \text{iff } Succ(s,a,s'), \text{ s.t. } a \in \mathcal{A}, \tag{5.4}$$

$$Tr([\mathbf{any},s],a) = \{[nil,s']\} \text{ iff } Succ(s,a,s'), \text{ s.t. } a \in \mathcal{A}, \tag{5.5}$$

$$Tr([\phi?,s],\varepsilon) = \{[nil,s]\} \quad \text{iff } s \models \phi. \tag{5.6}$$

Equations 5.4 and 5.5 dictate that actions in programs change the state according to the *Succ* relation described in the previous section. Finally, Eq. 5.6 defines transitions for ϕ? when ϕ is a sentence (i.e., a formula with no program variables). It expresses that a transition can only be carried out if the plan state so far satisfies ϕ.

Now we define *Tr* for non-atomic programs. In the definitions below, assume that $a \in \mathcal{A} \cup \{\varepsilon\}$, and that σ_1 and σ_2 are subprograms of σ, where occurring elements in V may have been instantiated by any

object in the planning instance I.

$$Tr([(\sigma_1;\sigma_2),s],a) = \{[(\sigma_1';\sigma_2),s'] \,|\, [\sigma_1',s'] \in Tr([\sigma_1,s],a)\} \text{ if } \sigma_1 \neq nil, \tag{5.7}$$

$$Tr([(nil;\sigma_2),s],\varepsilon) = \{[\sigma_2,s]\}, \tag{5.8}$$

$$Tr([\textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2,s],\varepsilon) = \begin{cases} [\sigma_1,s] & \text{if } s \models \phi, \\ [\sigma_2,s] & \text{if } s \not\models \phi, \end{cases} \tag{5.9}$$

$$Tr([(\sigma_1|\sigma_2),s],\varepsilon) = \{[\sigma_1,s],[\sigma_2,s]\} \tag{5.10}$$

$$Tr([\textbf{while}\,\phi\,\textbf{do}\,\sigma_1,s],\varepsilon) = \begin{cases} \{[nil,s]\} & \text{if } s \not\models \phi, \\ \{[\sigma_1;\textbf{while}\,\phi\,\textbf{do}\,\sigma_1,s]\} & \text{if } s \models \phi, \end{cases} \tag{5.11}$$

$$Tr([\sigma_1^*,s],\varepsilon) = \{[(\sigma_1;\sigma_1^*),s],[nil,s]\}, \tag{5.12}$$

$$Tr([\pi(x\text{-}t)\,\sigma_1,s],\varepsilon) = \{[\sigma_1|_{x/o},s] \,|\, (o,t) \in \tau_D \cup \tau_P\}. \tag{5.13}$$

where $\sigma_1|_{x/o}$ denotes the program resulting from replacing any occurrence of x in σ_1 by o. We now give some intuitions for the definitions. First, a transition on a sequence corresponds to transitioning on its first component first (Eq. 5.7), unless the first component is already the empty program, in which case we transition on the second component (Eq. 5.8). A transition on a conditional corresponds to a transition in the *then* or *else* part depending on the truth value of the condition (Eq. 5.9). A transition of the nondeterministic choice leads to the consideration of either of the programs (Eq. 5.10). A transition of a while-loop corresponds to the *nil* program if the condition is false, and corresponds to the body followed by the while-loop if the condition is true (Eq. 5.11). On the other hand, a transition of σ_1^* represents two alternatives: executing σ_1 at least once, or stopping the execution of σ_1^*, with the remaining program *nil* (Eq. 5.12). Finally, a transition of the nondeterministic choice corresponds to a transition of its body when the variable has been replaced by any object of the right type (Eq. 5.13).

To end the definition of $A_{\sigma,I}$, Q corresponds precisely to the program configurations $[\sigma',s]$ where σ' is either *nil* or a subprogram of σ such that program variables may have been replaced by objects in I, and s is any possible planning state. Moreover, Tr is assumed empty for elements of its domain not explicitly mentioned above. Finally, the set of accepting states is $F = \{[nil,s] \,|\, s \text{ is any state over } I\}$, i.e., those where no program remains in execution. We can now formally define an execution of a program.

Definition 5.1 (Execution of a program) *A sequence of actions $a_1 \cdots a_n$ is an execution of σ in I if $a_1 \cdots a_n$ is accepted by $A_{\sigma,I}$.*

As usual, we use the symbol $\vdash$ to represent a single computation of the automaton. We say that $q \vdash q'$ iff there exists an a such that $q' \in Tr(q,a)$. The symbol $\vdash^*$ represents the reflexive and transitive closure of $\vdash$. Finally, $q_0 \vdash^k q_k$ iff there are exist states $q_1,\ldots,q_{k-1}$ such that $q_0 \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_{k-1} \vdash q_k$.

Before defining what we mean by planning in the presence of control, we prove several results that attempt to justify the correctness of the defined semantics in terms of the standard intuition. The first result proves that the definition of the sequence is intuitively correct, i.e., the execution of $\sigma_1;\sigma_2$ corresponds to the execution of σ_1 followed by σ_2.

Proposition 5.1 *Let σ_1 and σ_2 be programs. If*

$$[\sigma_1;\sigma_2,s] \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_{k-1} \vdash q_k = [nil,s'],$$

then for some $i \in [1,k]$, $q_i = [\sigma_2,s']$ and $[\sigma_1,s] \vdash^* [nil,s']$.

Proof: See Section C.1 (p. 173). ■

Our second result proves that the semantics for the execution of an **if-then-else** is intuitively correct

Proposition 5.2 *Let ϕ be a BFQ and let σ_1 and σ_2 be programs. Then the following holds*

$$[\mathbf{if}\,\phi\,\mathbf{then}\,\sigma_1\,\mathbf{else}\,\sigma_2,s] \vdash^* [nil,s']$$

iff

$$s \models \phi \text{ and } [\sigma_1,s] \vdash^* [nil,s'], \text{ or } s \not\models \phi \text{ and } [\sigma_2,s] \vdash^* [nil,s'].$$

Proof: Straightforward by definition of *Tr*. ■

The execution of a nondeterministic choice of programs have the intended meaning too, as shown by the following result.

Proposition 5.3 *Let σ_1 and σ_2 be programs. Then the following holds*

$$[(\sigma_1|\sigma_2),s] \vdash^* [nil,s']$$

iff

$$[\sigma_1,s] \vdash^* [nil,s'] \text{ or } [\sigma_2,s] \vdash^* [nil,s'].$$

Proof: Straightforward by definition of *Tr*. ■

Now we prove that the execution of the while loop correspond to a repeated execution of the body of the loop.

Proposition 5.4 *Let ϕ be a BFQ and σ be a program. If*

$$[\mathbf{while}\,\phi\,\mathbf{do}\,\sigma,s] \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_k \vdash [nil,s'],$$

then:

1. *for all* $i \in [1,k]$, q_i *is either of the form* $q_i = [\sigma_r; \mathbf{while}\,\phi\,\mathbf{do}\,\sigma, r_i]$, *or of the form* $q_i = [\mathbf{while}\,\phi\,\mathbf{do}\,\sigma, r_i]$.

2. *for all* $i \in [1,k]$, *if* q_i *is of the form* $q_i = [\mathbf{while}\,\phi\,\mathbf{do}\,\sigma, r_i]$ *then* $i < k$ *iff* $r_i \models \phi$. *State* q_k *is of the form* $q_k = [\mathbf{while}\,\phi\,\mathbf{do}\,\sigma, r_k]$

3. *Finally, let n be the number of states* q_i $(i \in [1,k])$ *of the form* $q_i = [\mathbf{while}\,\phi\,\mathbf{do}\,\sigma, r_i]$. *Then,* $[\sigma^n, s] \vdash^* [nil, s']$, *where* σ^n *represents the sequence that repeats* σ *n times.*

Proof: See Section C.2 (p. 174). ■

In the Golog language (Levesque *et al.*, 1997), the **if-then-else** construct is defined by macro expansion, in terms of test actions and non-deterministic choices. Below we prove that our semantics for the **if-then-else** and for the Golog macro expansion of such a construct are equivalent.

Proposition 5.5 *Let* ϕ *be a BFQ and let* σ_1 *and* σ_2 *be programs. Then the following holds*

$$[\mathbf{if}\,\phi\,\mathbf{then}\,\sigma_1\,\mathbf{else}\,\sigma_2, s] \vdash [\sigma, s]$$

iff

$$[(\phi?;\sigma_1)|(\neg\phi?;\sigma_2), s] \vdash^3 [\sigma, s].$$

Proof: Straightforward from the definition of *Tr*. ■

Now that we have justified the correctness of the semantics of the control language, we return to planning. We are now ready to define the notion of planning under procedural control.

Definition 5.2 (Planning under procedural control) *A sequence of actions* $a_1 a_2 \cdots a_n$ *is a* plan for instance I under the control of program σ *if* $a_1 a_2 \cdots a_n$ *is a plan in* I *and is an execution of* σ *in* I.

5.4 Compiling Control into the Action Theory

This section describes a translation function that, given a program σ in the DCK language defined above together with a PDDL2.1 domain specification D, outputs a new PDDL2.1 domain specification D_σ and problem specification P_σ. The two resulting specifications can then be combined with any problem P defined over D, creating a new planning instance that embeds the control given by σ, i.e. that is such that only action sequences that are executions of σ are possible. This enables any PDDL2.1-compliant planner to exploit search control specified by any program.

To account for the state of execution of program σ and to describe legal transitions in that program, we introduce a few bookkeeping predicates and a few additional actions. Figure 5.1 graphically illustrates the translation of an example program shown as a finite state automaton. Intuitively, the operators

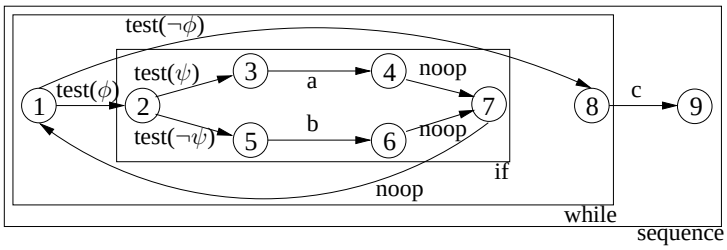


Figure 5.1: Automaton for **while** ϕ **do** (**if** ψ **then** a **else** b); c.

we generate in the compilation define the transitions of this automaton. Their preconditions and effects condition on and change the automaton's state.

The translation is defined inductively by a function $C(\sigma, n, E)$ which takes as input a program σ, an integer n, and a list of program variables with types $E = [e_1\text{-}t_1, \ldots, e_k\text{-}t_k]$, and outputs a tuple (L, L', n') with L a list of domain-independent operator definitions, L' a list of domain-dependent operator definitions, and n' another integer. Intuitively, E contains the program variables whose scope includes (sub-)program σ. Moreover, L' contains restrictions on the applicability of operators defined in D, and L contains additional control operators needed to enforce the search control defined in σ. Integers n and n' abstractly denote the program state before and after execution of σ.

We use two auxiliary functions. $Cnoop(n_1, n_2)$ produces an operator definition that allows a transition from state n_1 to n_2. Similarly $Ctest(\phi, n_1, n_2, E)$ defines the same transition, but conditioned on ϕ. They are defined as:[2]

$$
\begin{aligned}
&Cnoop(n_1, n_2) = \langle noop\_n_1\_n_2(), [\,], state = s_{n_1}, [state = s_{n_2}] \rangle \\
&Ctest(\phi, n_1, n_2, E) = \langle test\_n_1\_n_2(\vec{x}), \vec{t}, Prec(\vec{x}), Eff(\vec{x}) \rangle \text{ with} \\
&\quad (\vec{e\text{-}t}, \vec{x}) = mentions(\phi, E),\ \vec{e\text{-}t} = e_1\text{-}t_1, \ldots, e_m\text{-}t_m, \\
&\quad Prec(\vec{x}) = \big(state = s_{n_1} \wedge\ \phi[e_i/x_i]_{i=1}^{m} \wedge \textstyle\bigwedge_{i=1}^{m} bound(e_i) \rightarrow map(e_i, x_i)\big), \\
&\quad Eff(\vec{x}) = [state = s_{n_2}] \cdot [bound(e_i), map(e_i, x_i)]_{i=1}^{m}.
\end{aligned}
$$

Function $mentions(\phi, E)$ returns a vector $\vec{e\text{-}t}$ of program variables and types that occur in ϕ, and a vector $\vec{x}$ of new variables of the same length. Bookkeeping predicates serve the following purposes: *state* denotes the state of the automaton; $bound(e)$ expresses that the program variable e has been bound to an object of the domain; $map(e, o)$ states that this object is o. Thus, the implication $bound(e_i) \rightarrow map(e_i, x_i)$ forces parameter x_i to take the value to which e_i is bound, but has no effect if e_i is not bound.

[2] We use $A \cdot B$ to denote the concatenation of lists A and B.

Consider the inner box of Figure 5.1, depicting the compilation of the if statement. It is defined as:

$$
\begin{aligned}
C(\textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2, n, E) &= (L_1 \cdot L_2 \cdot X, L_1' \cdot L_2', n_3) \\
&\text{with } (L_1, L_1', n_1) = C(\sigma_1, n+1, E), \\
&(L_2, L_2', n_2) = C(\sigma_2, n_1+1, E),\ n_3 = n_2+1, \\
&X = [\, \mathit{Ctest}(\phi, n, n+1, E),\ \mathit{Ctest}(\neg\phi, n, n_1+1, E), \\
&\qquad \mathit{Cnoop}(n_1, n_3),\ \mathit{Cnoop}(n_2, n_3)\,]
\end{aligned}
$$

and in the example we have $\phi = \psi, n = 2, n_1 = 4, n_2 = 6, n_3 = 7, \sigma_1 = a$, and $\sigma_2 = b$.

The inductive definitions for other programs σ are:

$$C(nil,n,E) = ([\,],[\,],n)$$

$$C(O(\vec{r}),n,E) = ([\,],[\langle O(\vec{x}),\vec{t},Prec'(\vec{x}),Eff'(\vec{x})\rangle],n+1) \text{ with}$$

$$\langle O(\vec{x}),\vec{t},Prec(\vec{x}),Eff(\vec{x})\rangle \in Ops,\ \ \vec{r} = r_1,\ldots,r_m,$$

$$Prec'(\vec{x}) = (state = s_n \wedge \bigwedge_{i \text{ s.t. } r_i \in E} bound(r_i) \rightarrow map(r_i,x_i) \wedge \bigwedge_{i \text{ s.t. } r_i \notin E} x_i = r_i),$$

$$Eff'(\vec{x}) = [\ state = s_n \Rightarrow state = s_{n+1}] \cdot [state = s_n \Rightarrow bound(r_i) \wedge map(r_i,x_i)]_{i \text{ s.t. } r_i \in E}$$

$$C(\phi?,n,E) = (\ [Ctest(\phi,n,n+1,E)],\ [\,],\ n+1)$$

$$C((\sigma_1;\sigma_2),n,E) = (L_1 \cdot L_2,\ L_1' \cdot L_2',\ n_2) \text{ with}$$

$$(L_1,L_1',n_1) = C(\sigma_1,n,E), (L_2,L_2',n_2) = C(\sigma_2,n_1,E)$$

$$C((\sigma_1|\sigma_2),n,E) = (L_1 \cdot L_2 \cdot X, L_1' \cdot L_2', n_2+1) \text{ with}$$

$$(L_1,L_1',n_1) = C(\sigma_1,n+1,E),$$

$$(L_2,L_2',n_2) = C(\sigma_2,n_1+1,E),$$

$$X = [\ Cnoop(n,n+1),\ Cnoop(n,n_1+1), Cnoop(n_1,n_2+1),\ Cnoop(n_2,n_2+1)\ ]$$

$$C(\textbf{while}\ \phi\ \textbf{do}\ \sigma,n,E) = (L \cdot X, L', n_1+1) \text{ with}$$

$$(L,L',n_1) = C(\sigma,n+1,E),\ X = [Ctest(\phi,n,n+1,E), Ctest(\neg\phi,n,n_1+1,E), Cnoop(n_1,n)]$$

$$C(\sigma^*,n,E) = (L \cdot [Cnoop(n,n_2), Cnoop(n_1,n)], L', n_2)$$

$$\text{with } (L,L',n_1) = C(\sigma,n,E), n_2 = n_1+1$$

$$C(\pi(x\text{-}t,\sigma),n,E) = (L \cdot X, L', n_1+1) \text{ with}$$

$$(L,L',n_1) = C(\sigma,n,E \cdot [x\text{-}t]),$$

$$X = [\langle free\_n_1(x),t,\ state = s_{n_1}, [state = s_{n_1+1}, \neg bound(x), \forall y.\neg map(x,y)]\rangle\ ]$$

The atomic program **any** is handled by macro expansion to above defined constructs.

As mentioned above, given program σ, the return value (L,L',n_{final}) of $C(\sigma,0,[\,])$ is such that L contains new operators for encoding transitions in the automaton, whereas L' contains restrictions on the applicability of the original operators of the domain. Now we are ready to integrate these new operators and restrictions with the original domain specification D to produce the new domain specification D_σ.

D_σ contains a constrained version of the operators $O(\vec{x})$ of the original domain D also mentioned in

L'. Let $[\langle O(\vec{x}),\vec{t},Prec_i(\vec{x}),Eff_i(\vec{x})\rangle]_{i=1}^{n}$ be the sublist of L' that contains additional conditions for operator $O(\vec{x})$. The operator replacing $O(\vec{x})$ in D_σ is defined as:

$$\langle O'(\vec{x}),\ \vec{t},\ Prec(\vec{x}) \wedge \bigvee_{i=1}^{n} Prec_i(\vec{x}),\ Eff(\vec{x}) \cup \bigcup_{i=1}^{n} Eff_i(\vec{x})\rangle$$

Additionally, D_σ contains all operator definitions in L. Objects in D_σ are the same as those in D, plus a few new ones to represent the program variables and the automaton's states s_i ($0 \leq i \leq n_{\text{final}}$). Finally D_σ inherits all predicates in D plus $bound(x)$, $map(x,y)$, and function *state*.

The translation, up to this point, is problem-independent; the problem specification P_σ is defined as follows. Given any predefined problem P over D, P_σ is like P except that its initial state contains condition $state = s_0$, and its goal contains $state = s_{n_{\text{final}}}$. Those conditions ensure that the program must be executed to completion.

As is shown below, planning in the generated instance $I_\sigma = (D_\sigma, P_\sigma)$ is equivalent to planning for the original instance $I = (D,P)$ under the control of program σ, except that plans on I_σ contain actions that were not part of the original domain definition (*test*, *noop*, and *free*).

Theorem 5.1 (Correctness) *Let $Filter(\alpha,D)$ denote the sequence that remains when removing from α any action not defined in D. If α is a plan for instance $I_\sigma = (D_\sigma,P_\sigma)$ then $Filter(\alpha,D)$ is a plan for $I = (D,P)$ under the control of σ. Conversely, if α is a plan for I under the control of σ, there exists a plan α' for I_σ, such that $\alpha = Filter(\alpha',D)$.*

Proof: See Section C.3 (page 174) . ■

Now we turn our attention to analyzing the size of the output planning instance relative to the original instance and control program. Assume we define the size of a program as the number of programming constructs and actions it contains. Then we obtain the following result.

Theorem 5.2 (Succinctness) *Let σ is a program of size m, and let k be the maximal nesting depth of π(x-t) statements in σ, then $|I_\sigma|$ (the overall size of I_σ) is $O((k+p)m)$, where p is the size of the largest operator in I.*

Proof: See Section C.4 (page 183). ■

The encoding of programs in PDDL2.1 is, hence, in worst case $O(k)$ times bigger than the program itself. It is also easy to show that the translation is done in time linear in the size of the program, since, by definition, every occurrence of a program construct is only dealt with once.

5.5 Exploiting DCK in State-of-the-Art Heuristic Planners

Our objective in translating procedural DCK to PDDL2.1 was to enable *any* PDDL2.1-compliant state-of-the-art planner to seamlessly exploit our DCK. In this section, we investigate ways to best leverage our translated domains using domain-independent heuristic search planners.

There are several compelling reasons for wanting to apply domain-independent heuristic search to these problems. Procedural DCK can take many forms. Often, it will provide explicit actions for some parts of a sequential plan, but not for others. In such cases, it will contain unconstrained fragments (i.e., fragments with nondeterministic choices of actions) where the designer expects the planner to figure out the best choice of actions to realize a sub-task. In the absence of domain-specific guidance for these unconstrained fragments, it is natural to consider using a domain-independent heuristic to guide the search.

In many domains it is very hard to write deterministic procedural DCK, i.e. DCK that restricts the search space in such a way that solutions can be obtained very efficiently, even using blind search. An example of such a domain is one where plans involve solving an optimization sub-problem. In such cases, procedural DCK will contain open parts (fragments of nondeterministc choice within the DCK), where the designer expects the planner to figure out the best way of completing a sub-task. However, in the absence of domain-specific guidance for these open parts, it is natural to consider using a domain-independent heuristic to guide the search.

In other cases, it is the choice of action arguments, rather than the choice of actions that must be optimized. In particular, fragments of DCK may collectively impose global constraints on action argument choices that need to be enforced by the planner. As such, the planner needs to be *aware* of the procedural control in order to avoid backtracking. By way of illustration, consider a travel planning domain comprising two tasks “buy air ticket” followed by “book hotel”. Each DCK fragment restricts the actions that can be used, but leaves the choice of arguments to the planner. Further suppose that budget is limited. We would like our planner to realize that actions used to complete the first part should save enough money to complete the second task. The ability to do such lookahead can be achieved via domain-independent heuristic search.

In the rest of the section we propose three ways in which one can leverage our translated domains using a domain-independent heuristic planner. These three techniques differ predominantly in the operands they consider in computing heuristics.

5.5.1 Direct Use of Translation (*Simple*)

As the name suggests, a simple way to provide heuristic guidance while enforcing program awareness is to use our translated domain directly with a domain-independent heuristic planner. In short, take the original domain instance I and control σ, and use the resulting instance I_σ with any heuristic planner.

Unfortunately, when exploiting a relaxed graph to compute heuristics, two issues arise. First, since both the *map* and *bound* predicates are relaxed, whatever value is already assigned to a variable, will remain assigned to that variable. This can cause a problem with iterative control. For example, assume program $\sigma_L \stackrel{\text{def}}{=} \textbf{while}\,\phi\,\textbf{do}\,\pi(c\text{-}crate)\,unload(c,T)$, is intended for a domain where crates can be only unloaded sequentially from a truck. While expanding the relaxed plan, as soon as variable *c* is bound to some value, action *unload* can only take that value as argument. This leads the heuristic to regard most instances as unsolvable, returning misleading estimates.

The second issue is one of efficiency. Since fluent *state* is also relaxed, the benefits of the reduced branching factor induced by the programs is lost. This could slow down the computation of the heuristic significantly.

5.5.2 Modified Program Structure (*H-ops*)

The *H-ops* approach addresses the two issues potentially affecting the computation of the *Simple* heuristic. It is designed to be used with planners that employ relaxed planning graphs for heuristic computation. The input to the planner in this case is a pair $(I_\sigma, HOps)$, where $I_\sigma = (D_\sigma, P_\sigma)$ is the translated instance, and *HOps* is an additional set of planning operators. The planner uses the operators in D_σ to generate successor states while searching. However, when computing the heuristic for a state *s* it uses the operators in *HOps*.

Additionally, function *state* and predicates *bound* and *map* are *not* relaxed. This means that when computing the relaxed graph we actually delete their instances from the relaxed states. As usual, *deletes* are processed before *adds*. The expansion of the graph is stopped if the goal or a fixed point is reached. Finally, a relaxed plan is extracted in the usual way, and its length is reported as the heuristic value. In the computation of the length, auxiliary actions such as tests and noops are ignored.

The un-relaxing of *state*, *bound* and *map* addresses the problem of reflecting the reduced branching factor provided by the control program while computing the heuristics. However, it introduces other problems. Returning to the σ_L program defined above, since *state* is now un-relaxed, the relaxed graph expansion cannot escape from the loop, because under the relaxed planning semantics, as soon as ϕ is true, it remains true forever. A similar issue occurs with the nondeterministic iteration. Furthermore, we want to avoid state duplication, i.e. having *state* equal to two different values at the same time in the same relaxed state. This could happen for example while reaching an **if** construct whose condition is both true and false at the same time (this can happen because *p* and *not-p* can both be true in a relaxed state).

This issue is addressed by the *HOps* operators. To avoid staying in the loop forever, the loop will be exited when actions in it are no longer adding effects. Figure 5.2 provides a graphical representation. An important detail to note is that the loop is not entered when ϕ is not found true in the relaxed state. (The expression *not* ϕ should be understood as negation as failure.) Moreover, the pseudo-fluent fp is

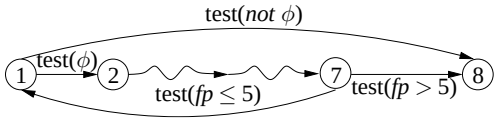


Figure 5.2: *H-ops* translation of **while** loops. While computing the heuristics, pseudo-fluent *fp* is increased each time no new effect is added into the relaxed state, and it is set to 0 otherwise. The loop can be exited if the last five (7-2) actions performed didn't add any new effect.

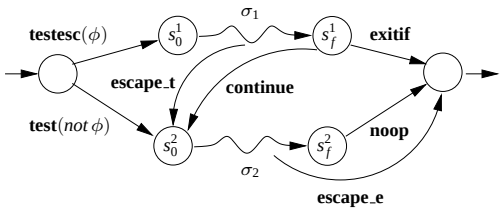


Figure 5.3: *H-ops* translation for **if-then-else**. Action **testesc**(ϕ) is possible if condition ϕ is true. If condition $\neg\phi$ is also true in the relaxed state, the **testesc**(ϕ) dds a fact *escape_active* that will enable the execution of **continue** and **escape_t** and **escape_e**. Actions **escape_t** and **escape_e** are possible only when no other actions are possible. This is checked using the pseudo-fluent fp described in Figure 5.2. Action **exitif** is only possible if *escape_active* is true. Both the **noop** and the **escape_e** actions delete the fact *escape_active*. Nested **if** constructs are handled using a parameterized version of the *escape_active* predicate.

an internal variable of the planner that acts as a real fluent for the *HOps*. A similar approach is adopted for nodeterministic iterations, whose description we omit here.

Since loops are guaranteed to be exited, the computation of *H-ops* is guaranteed to finish because at some relaxed state the final state of the automaton will be reached. At this point, if the goal is not true, no operators will be possible and a fixed point will be produced immediately.

For **if**'s, if the condition is both true and false at the same time, the **then** part is processed first, followed by the **else** part. The objective of this is avoidance of state duplication. However, this new interpretation of the **if** introduces a new problem. This problem occurs when, while performing the actions of one of the parts, no action is possible anymore. Intuitively, this could happen because the heuristics has chosen the wrong subprogram to execute actions from. Indeed, if there exists an execution of the program from state *s* that executes the "then" part of the **if**, it can happen that, during the computation of the heuristic for *s*, the "else" part forces some actions to occur that are not possible. Under normal circumstances, the non existence of any possible action produces a fixed point. Because the goal is not reached on such a fixed point, the heuristic regards the goal as unreachable, which could be a wrong estimation.

To solve this problem, *H-ops* considers new "escape" actions, that are executable only when no more actions are possible. Escapes can be performed only inside "then" or "else" bodies. After executing an escape, the simulation of the program's execution jumps to the else part if the escape occurs in the "then" part, or to the end of the **if**, if the escape occurs in the then part. Figure 5.5.2 shows a graphical representation of the *H-ops* generated for the **if**.

5.5.3 A Program-Unaware Approach (*Basic*)

Our program-unaware approach (*Basic*) completely ignores the program when computing heuristics. Here, the input to the planner is a pair (I_σ, Ops), where I_σ is the translated instance, and *Ops* are the *original* domain operators. The *Ops* operators are used exclusively to compute the heuristic. Hence, *Basic*'s output is not at all influenced by the control program.

Although *Basic* is program unaware, it can sometimes provide good estimates, as we see in the following section. This is especially true when the DCK characterizes a solution that would be naturally found by the planner if no control were used. It is also relatively fast to compute.

5.6 Implementation and Experiments

Our implementation[3] takes a PDDL planning instance and a DCK program and generates a new PDDL planning instance. It will also generate appropriate output for the *Basic* and *H-ops* heuristics, which require a different set of operators. Thus, the resulting PDDL instance may contain definitions for operators that are used only for heuristic computation using the `:h-action` keyword, whose syntax is analogous to the PDDL keyword `:action`.

Our planner is a modified version of TLPLAN, which does a best-first search using an FF-style heuristic. It is capable of reading the PDDL with extended operators.

We performed our experiments on the *trucks*, *storage* and *rovers* domains (30 instances each). We wrote DCK for these domains. For details on the Golog code used for these examples, see Section E. We ran our three heuristic approaches (*Basic*, *H-ops*, and *Simple*) and cycle-free, depth-first search on the translated instance (*blind*). Additionally, we ran the original instance of the program (DCK-free) using the domain-independent heuristics provided by the planner (*original*). Table 5.1 shows various statistics on the performance of the approaches. Furthermore, Fig. 5.4 shows times for the different heuristic approaches.

Not surprisingly, our data confirms that DCK helps to improve the performance of the planner, solving more instances across all domains. In some domains (i.e. storage and rovers) blind depth-first cycle-free search is sufficient for solving most of the instances. However, quality of solutions (plan

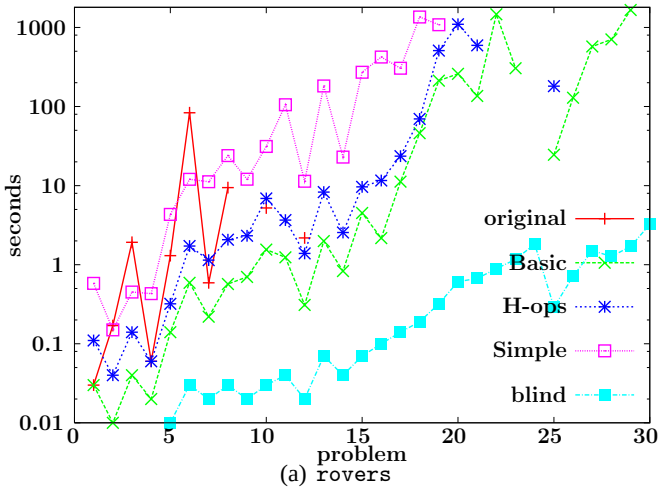
[3] Available at `www.cs.toronto.edu/kr/systems`

		original	*Simple*	*Basic*	*H-ops*	*blind*
Trucks	#n	1	0.31	0.41	0.26	19.85
	#s	9	9	15	14	3
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	1.1	1.03	1.02	1.04	1.04
	ℓ_{max}	1.2	1.2	1.07	1.2	1.07
Rovers	#n	1	0.74	1.06	1.06	1.62
	#s	10	19	28	22	30
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	2.13	1.03	1.05	1.21	1.53
	ℓ_{max}	4.59	1.2	1.3	1.7	2.14
Storage	#n	1	1.2	1.13	0.76	1.45
	#s	18	18	20	21	20
	ℓ_{min}	1	1	1	1	1
	ℓ_{avg}	4.4	1.05	1.01	1.07	1.62
	ℓ_{max}	21.11	1.29	1.16	1.48	2.11

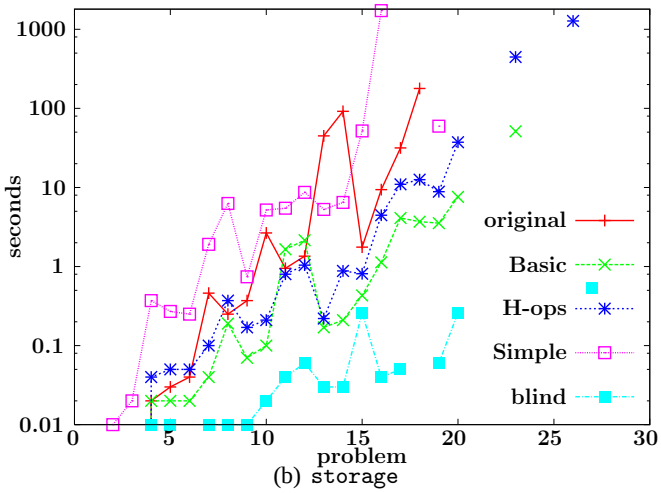
Table 5.1: Comparison between different approaches to planning (with DCK). #n is the average factor of expanded nodes to the number of nodes expanded by *original* (i.e., #n=0.26 means the approach expanded 0.26 times the number of nodes expanded by original). #s is the number of problems solved by each approach. ℓ_{avg} denotes the average ratio of the plan length to the shortest plan found by any of the approaches (i.e., ℓ_{avg}=1.50 means that on average, on each instance, plans where 50% longer than the shortest plan found for that instance). ℓ_{min} and ℓ_{max} are defined analogously.

length) is poor compared to the heuristic approaches. In trucks, DCK is only effective in conjunction with heuristics; blind search can solve very few instances.

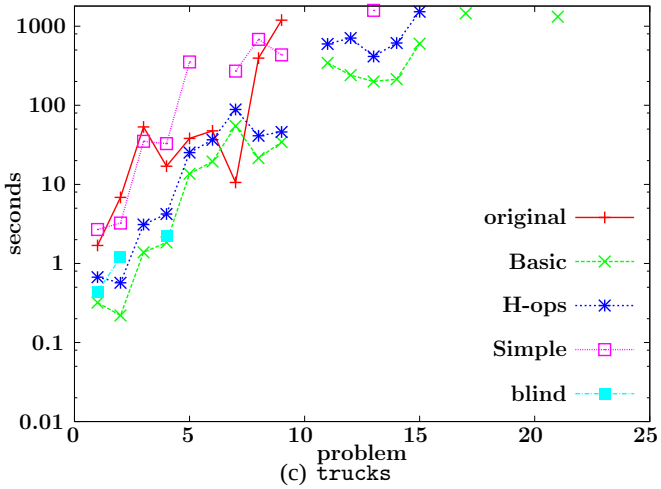
We observe that *H-ops* is the most informative (expands fewer nodes). This fact does not pay off in time in the experiments shown in the table. Nevertheless, it is easy to construct instances where the *H-ops* performs better than *Basic*. This happens when the DCK control restricts the space of valid plans (i.e., prunes out valid plans). We have experimented with various instances of the storage domain, where we restrict the plan to use only one hoist. In some of these cases *H-ops* outperforms *Basic* by orders of magnitude.



(a) `rovers`



(b) `storage`



(c) `trucks`

Figure 5.4: Running times of the three heuristics and the original instance; logarithmic scale; run on an Intel Xeon, 3.6GHz, 2GB RAM

5.7 Summary and Related Work

DCK can be used to constrain the set of valid plans and has proven an effective tool in reducing the time required to generate a plan. Moreover, DCK can be used to specify "plan skeletons" which may be effective for addressing other problems, such as WSC or component software composition.

Nevertheless, many of the planners that exploit it use arguably less natural state-centric DCK specification languages, and their planners use blind search. In this chapter we examined the problem of exploiting procedural DCK with state-of-the-art planners. Our goal was to specify rich DCK naturally in the form of a program template and to exploit state-of-the-art planning techniques to actively plan towards the achievement of this DCK. To this end we made three contributions: provision of a procedural DCK language syntax and semantics; a polynomial-time algorithm to compile DCK and a planning instance into a PDDL2.1 planning instance that could be input to any PDDL2.1-compliant planner; and finally a set of techniques for exploiting domain-independent heuristic search with our translated DCK planning instances. Each contribution is of value in and of itself. The language can be used without the compilation, and the compiled PDDL2.1 instance can be input to any PDDL2.1-compliant state-of-the-art planner, not just the domain-independent heuristic search planner that we propose. Our experiments show that procedural DCK improves the performance of state-of-the-art planners, and that our heuristics are sometimes key to achieving good performance.

Much of the previous work on DCK in planning has exploited state-centric specification languages. In particular, TLPLAN (Bacchus and Kabanza, 1998) and TALPLAN (Kvarnström and Doherty, 2000) employ declarative, state-centric, temporal languages based on LTL to specify DCK. Such languages define necessary properties of states over fragments of a valid plan. We argue that they could be less natural than our procedural specification language.

Though not described as DCK specification languages there are a number of languages from the agent programming and/or model-based programming communities that are related to procedural control. Among these are EAGLE, a goal language designed to also express intentionality (dal Lago *et al.*, 2002). Moreover, Golog is a procedural language proposed as an alternative to planning by the cognitive robotics community. It essentially constrains the possible space of actions that could be performed by the programmed agent allowing non-determinism. Our DCK language can be viewed as a version of Golog. Further, languages such as the Reactive Model-Based Programming Language (RMPL) (Kim, Williams, and Abramson, 2001) – a procedural language that combines ideas from constraint-based modeling with reactive programming constructs – also share expressive power and goals with procedural DCK. Finally, HTN specification languages such as those used in SHOP (Nau *et al.*, 1999) domain-dependent hierarchical task decompositions together with partial order constraints, not easily describable in our language. However, Fritz, Baier, and McIlraith (2008) have recently provided a compilation of ConGolog (De Giacomo *et al.*, 2000), a successor of Golog, to PDDL. ConGolog can

represent various HTN constructs.

A focus of our work was to exploit state-of-the-art planners and planning techniques with our procedural DCK. In contrast, well-known DCK-enabled planners such as TLPLAN and TALPLAN use DCK to prune the search space at each step of the plan and then employ blind depth-first cycle-free search to try to reach the goal. Unfortunately, pruning is only possible for maintenance-style DCK and there is no way to plan towards achieving other types of DCK as there is with the heuristic search techniques proposed here.

Similarly, Golog interpreters, while exploiting procedural DCK, have traditionally employed blind search to instantiate nondeterministic fragments of a Golog program. Most recently, Claßen *et al.* (2007) have proposed to integrate an incremental Golog interpreter with a state-of-the-art planner. Their motivation is similar to ours, but there is a subtle difference: they are interested in combining *agent programming* and efficient planning. The integration works by allowing a Golog program to make explicit calls to a state-of-the-art planner to achieve particular conditions identified by the user. The actual planning, however, is not controlled in any way. Also, since the Golog interpreter executes the returned plan immediately without further lookahead, backtracking does not extend over the boundary between Golog and the planner. As such, each fragment of nondeterminism within a program is treated independently, so that actions selected locally are not informed by the constraints of later fragments as they are with the approach that we propose. Their work, which focuses on the semantics of ADL in the situation calculus, is hence orthogonal to ours.

Finally, there is related work that compiles DCK into standard planning domains. Baier and McIlraith (2006b), Cresswell and Coddington (2004), Edelkamp (2006a), and Rintanen (2000), propose to compile different versions of LTL-based DCK into PDDL/ADL planning domains. The main drawback of these approaches is that translating full LTL into ADL/PDDL is worst-case exponential in the size of the control formula whereas our compilation produces an addition to the original PDDL instance that is linear in the size of the DCK program. Son, Baral, Nam, and McIlraith (2006) further show how HTN, LTL, and Golog-like DCK can be encoded into planning instances that can be solved using answer set solvers. Nevertheless, they do not provide translations that can be integrated with PDDL-compliant state-of-the-art planners, nor do they propose any heuristic approaches to planning with them.

Chapter 6

Planning with Programs that Sense

6.1 Introduction

In the previous chapter, we developed an algorithm that enables any PDDL-compliant planner to plan with Golog procedural control. This is important because many applications require planning in such conditions. However, as we mentioned in the introduction of this document, in many cases the *building blocks* for plans are not simply primitive actions but *complex actions* or *programs*. Moreover, programs may be able to *sense* the environment.

Our interest in this chapter is to develop an algorithm that will enable existing operator-based planners to plan with *programs*, rather than or in addition to primitive actions, as the building blocks for plans. By doing so, we enable recent advances in these planners to be leveraged for planning with programs. Our approach is distinct from previous work (McIlraith and Fadel, 2002) in that it can handle programs that sense the environment.

Our approach is to develop a technique for compiling programs into new primitive actions that can be exploited by standard operator-based planning techniques. To achieve this, we automatically extract (knowledge) preconditions and (knowledge) effects from programs. We study this problem in the language of the situation calculus, appealing to Golog to represent our programs. The output of our compilation process is expressed as a situation calculus theory. This output can be translated into a PDDL specification rather straightforwardly (Pednault, 1989; Claßen *et al.*, 2007).

A primary motivation for this work is to provide a theoretical framework for the use of conditional (knowledge producing) macro-actions. Planning with some form of macro-actions (e.g Fikes, Hart, and Nilsson, 1972; Sacerdoti, 1974; Korf, 1987; McIlraith and Fadel, 2002; Erol *et al.*, 1994) can dramatically improve the efficiency of plan generation. As such, our work enables practitioners that want to improve the performance of planning applications by adding or learning complex macro-actions to plan

with these actions without requiring to implement any extensions to their planner.

Our secondary motivation for investigating this topic is to address the problem of automated component software composition and specifically WSC (e.g. McIlraith and Son, 2002). Web services are self-contained, Web-accessible computer programs, such as the airline ticket service at www.aircanada.com, or the weather service at www.weather.com. These services are indeed programs that sense—e.g. by determining the balance of an account or flight costs by querying a database—and act in the world—e.g. by arranging for the delivery of goods, by debiting accounts, etc. As such, the task of WSC can be conceived as the task of planning with programs, or as a specialized version of a program synthesis task.

6.1.1 Contributions and Outline

The main contributions or our work follow.

1. Levesque (1996) argued that when planning with sensing, the outcome of the planning process should be a plan which the executing agent knows at the outset will lead to a final situation in which the goal is satisfied. Even in cases where no uncertainty in the outcome of actions, and no exogenous actions are assumed this remains challenging because of incomplete information about the initial state. To plan effectively with programs, we must consider whether we have the knowledge to actually execute the program prior to using it in a plan. To that end, in Section 6.3 we propose an offline execution semantics for Golog programs with sensing that enables us to determine that we know how to execute a program. We prove the equivalence of our semantics to the original Golog semantics, under certain conditions.

2. The main contribution of this work is the compilation method that transforms our action theory with programs into a new theory where programs are replaced by primitive actions (Section 6.4.1). This enables us to use traditional operator-based planning techniques to plan with programs that sense in a restricted but compelling set of cases.

3. Because the operators that result from the compilation may sense various properties at the same time, and, additionally have conditional knowledge effects, it is not obvious whether or not our compilation can be immediately used by standard operator-based planners with sensing. Thus, we provide an analysis of the applicability of the results presented in the chapter to those planners (Section 6.5). We also extend the PKS planning system (Petrick and Bacchus, 2002) to handle sensing for complex formulae

In Section 6.6 we discuss the practical relevance of this work by illustrating the potential computational advantages of planning with programs that sense. We also discuss the relevance of this work to WSC.

6.2 Preliminaries

The situation calculus and Golog provide the theoretical foundations for our work. In the two subsections that follow we briefly review the situation calculus (McCarthy and Hayes, 1969; Reiter, 2001), including a treatment of sensing actions and knowledge. We also review the transition semantics for Golog, a high-level agent programming language that we employ to represent the programs we are composing.

6.2.1 The Situation Calculus

The Situation Calculus, as described by Reiter (2001), is a sorted second-order language for specifying and reasoning about dynamical systems. In the Situation Calculus, the world changes as the result of *actions*. A *situation* is a term denoting the history of actions performed from an initial distinguished situation, S_0. The function $do(a,s)$ denotes the situation that results from performing action a in situation s[1]. Relational *fluents* are situation-dependent predicates that capture the changing state of the world.[2] Finally, the distinguished predicate $Poss(a,s)$ is used to express that it is possible to execute action a in situation s.

To represent knowledge in the Situation Calculus, we essentially follow Scherl and Levesque's formalism (2003). Thus, we consider an additional predicate symbol K, which has two arguments of the sort situation, such that $K(s',s)$ indicates that s' is *accessible* from s. K intuitively represents that s' is a "possible world" given that we are in s, and therefore adapts Moore's possible-world model of knowledge (1985) to the Situation Calculus. Finally, the relation $\sqsubset$ is such that if $s \sqsubset s'$ it is possible to reach situation s' from s by performing a non-empty sequence of actions.

6.2.2 Basic Action Theories

The dynamics of a particular domain is described by *basic action theories* (BATs). Before defining BATs precisely, we need to introduce the notion of *uniform formulae*. Intuitively, a uniform formula in the set of situation terms Υ is one that does not contain *Poss* or $\sqsubset$, it does not quantify over variables of the sort situation, it does not mention equality on situations, and whenever it mentions a term of sort situation in the situation argument position of a fluent, then that term is in Υ.

Definition 6.1 (Uniform Formula) *Let Υ be a set of terms of the sort situation.*

- *Any situation-independent term is uniform in Υ.*[3]

[1] $do([a_1,\ldots,a_n],s)$ abbreviates $do(a_n,do(\ldots,do(a_1,s)\ldots))$.

[2] In this chapter we do not deal with *functional fluents* and thus omit their description.

[3] Note that since we do not allow functional fluents, the only situation-independent terms of the language may be either variables not of the sort situation or situation-independent function terms

- *Formulae that are uniform in Υ are inductively defined as follows:*

  1. *Any formula that does not mention a term of sort situation is uniform in Υ.*
  2. *When F is an $(n+1)$-ary relational fluent, $t_1,\ldots,t_n$ are terms uniform in $\{\sigma\}$ whose sorts are appropriate[4] for F, and $\sigma \in \Upsilon$, then $F(t_1,\ldots,t_m,\sigma)$ is uniform in Υ.*
  3. *If U_1 and U_2 are formulae uniform in Υ, so are $\neg U_1$, $U_1 \wedge U_2$, and $(\exists v)U_1$, provided v is a variable not of sort situation.*

Our definition generalizes Definition 4.4.1 by Reiter (2001), in that it considers multiple situation terms in which a formula can be uniform. We introduce this generalized version since later in this chapter we will guarantee certain conditions on our compiled theories, that can be expressed compactly in terms of this definition.

In the remainder of the chapter, whenever a formula W is uniform in a singleton set $\{s\}$ we simply say that W is *uniform in s*. Likewise we say that W is *uniform in s and s'* whenever W is uniform in $\{s,s'\}$.

Now we are ready to describe how to model a dynamic domain using the Situation Calculus. Reiter's basic action theory with a treatment for knowledge has the form

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{K}_{init},$$

where,

- Σ is a set of foundational axioms.
- $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs), of the form:[5,6]

$$F(\vec{x}, do(a,s)) \equiv \Phi_F(a,\vec{x},s), \tag{6.1}$$

  where $\Phi_F(a,\vec{x},s)$ is a formula uniform in s and whose free variables are among a, $\vec{x}$, and s. The set of SSAs can be compiled from a set of *effect axioms*, $\mathcal{D}_{eff}$ (Reiter, 2001). An effect axiom describes the effect of an action on the truth value of certain fluents, e.g.,

$$a = startCar \supset engineStarted(do(a,s)).$$

  The general form of effect axioms is:

$$\gamma(\vec{x},a,s) \supset [\neg]F(\vec{x},do(a,s)), \tag{6.2}$$

[4] We borrow the term "appropriate" from Reiter's definition. In short, a term that is the argument to a predicate in the i-th position as the appropriate sort iff it has the expected sort such a position. Among other things, this means that an argument to a fluent atom cannot be sort situation unless it is the last argument.

[5] For notational convenience we use the notation $\phi(\vec{x},s)$ to represent a formula $\phi(x_1,\ldots,x_n,s)$ of the Situation Calculus, for some n.

[6] All free variables in formulae are regarded as universally quantified.

where $[\neg]$ indicates that $\neg$ may or may not appear. $\gamma(\vec{x},a,s)$ is a formula uniform in s and whose free variables are among $\vec{x}$, a, and s.

$\mathcal{D}_{ss}$ contains an axiom for the K fluent that we review below.

- $\mathcal{D}_{ap}$ contains action precondition axioms. For each action function symbol A of the language, there is a precondition axiom of the form:

$$Poss(A(\vec{x}),s) \equiv \Pi_A(\vec{x},s), \tag{6.3}$$

where $\Pi_A(\vec{x},s)$ is a formula uniform in s and whose free variables are among $\vec{x}$ and s. $\Pi_A(\vec{x},s)$ expresses all the conditions under which a can be performed in s.

- $\mathcal{D}_{una}$ contains unique names axioms for actions.

- $\mathcal{D}_{S_0}$ describes the initial state of the world.

- $\mathcal{K}_{init}$ defines the properties of the K fluent in the initial situation. The form of the successor state axiom for K guarantees that these properties are preserved in all successors of s. One of the conditions that we assume for K in the rest of the chapter is *reflexivity*. This condition is enforced by adding the following axiom to $\mathcal{K}_{init}$.

$$(\forall s).Init(s) \supset K(s,s), \tag{6.4}$$

where $Init(s) \stackrel{\text{def}}{=} \neg(\exists a,s')\, s = do(a,s')$. Reflexivity of K implies that everything that is known in situation s is also true in s.

6.2.3 Representing Knowledge

Our representation of knowledge closely follows the formalism introduced by Scherl and Levesque (2003). As they do, we use the distinguished fluent K to capture the knowledge of an agent in the Situation Calculus. However, as opposed to their treatment, we assume that the successor state axiom for K has a slightly different form that allows a sensing action to sense multiple formulae. Our SSA for K looks closer in syntax to the one that was proposed earlier by the same authors (Scherl and Levesque, 1993), which was also adopted by Reiter (2001). Although we use regression extensively, we do not appeal to an extension of Reiter's regression operators proposed by Scherl and Levesque that allows regressing the knowledge of the agent.

We now proceed to formally define two notions of knowledge for an agent, and we then describe the structure of the successor state axiom for K.

What Does an Agent Know?

We will extensively use two notions of knowledge. First, we want to establish that an agent *knows* a formula ϕ in a certain situation s. Essentially, this means that ϕ should hold in any situation that is accessible from s. Our second notion of knowledge corresponds to *knowing whether or not* a formula ϕ is true. This second notion is weaker, but is useful since it allows to reason hypothetically about the knowledge of the agent. For example, we expect from our formalization that if an agent has a sensor for ϕ, then we can conclude that after performing an action that reads this sensor, the agent will know whether or not ϕ, even if the action has not been actually performed.

The following are the formal definitions that summarize the intuitions given above.

$$\textbf{Knows}(\phi, s) \stackrel{\text{def}}{=} (\forall s').K(s', s) \supset \phi[s'],$$
$$\textbf{KWhether}(\phi, s) \stackrel{\text{def}}{=} \textbf{Knows}(\phi, s) \vee \textbf{Knows}(\neg\phi, s),$$

where ϕ is a *situation-suppressed* formula (i.e. a Situation Calculus formula whose situation terms are suppressed), and $\phi[s]$ denotes the formula that restores situation arguments in ϕ by s.

A Successor State Axiom for K

Scherl and Levesque (2003) define a standard SSA for the K predicate. The axiom is:[7]

$$K(s'', do(a,s)) \equiv (\exists s').s'' = do(a,s') \wedge K(s',s) \wedge Poss(a,s') \wedge SF(a,s) \equiv SF(a,s'), \tag{6.5}$$

Where $SF(a,s)$ is a distinguished predicate that defines the formula that is sensed by a sensing action a.

Intuitively, if an action a is performed in s and s' was K-accessible from s then $do(a,s')$ is K-accessible from $do(a,s)$ only if $SF(a,s)$ and $SF(a,s')$ have the same truth value. For sensing actions, $SF(a,s)$ is equivalent to a formula $\phi(s)$, which is *sensed* by a. For a non-sensing action a, $SF(a,s)$ is equivalent to $True$. This means that if one performs a non-sensing action a in s, if s' was K-accessible from s then so is $do(a,s')$ from $do(a,s)$.

As mentioned above, if an action $sense_\phi$ senses whether or not fluent ϕ is true, then we would add $SF(sense_\phi, s) \equiv \phi(s)$ to the domain theory. The SF notation also allows expressing context-dependent sensing. For example, by using the axioms:

$$InRoom_1(s) \supset (SF(senseLight, s) \equiv Light_1(s)),$$
$$InRoom_2(s) \supset (SF(senseLight, s) \equiv Light_2(s)),$$

we establish that action *senseLight* inspects the truth value of different fluents depending on the location of the robot. Even though SF provides a good level of flexibility, it cannot be used straightforwardly to

[7] Scherl and Levesque's paper actually uses SR instead of SF. Since SF seems to be standard (e.g. Levesque, 1996) when dealing with knowledge about Boolean formulae, we stick to it here.

represent actions that sense the truth value of multiple formulae at the same time. The reason is that in those cases, we need to specify multiple equivalencies for a single action.

In the rest of this chapter we adopt a slightly different successor state axiom for K, that allows sensing multiple properties and also allows context-dependent sensing. If the language of a theory contains n action function symbols $A_1, \ldots, A_n$ our $\mathcal{D}_{ss}$ includes:

$$K(s'', do(a,s)) \equiv (\exists s').\, s'' = do(a,s') \wedge K(s',s) \wedge Poss(a,s') \wedge \bigwedge_{i=1}^{n} \{a = A_i(\vec{x}_i) \supset \mathsf{SensedCond}(A_i(\vec{x}_i), s, s')\}, \tag{6.6}$$

where $\mathsf{SensedCond}(A_i(\vec{x}_i), s, s'')$ stands for a formula that expresses the sensing condition of action $A_i(\vec{x})$. $\mathsf{SensedCond}(A_i(\vec{x}_i), s, s'')$ is defined by macro expansion and, if A_i is a sensing action, it has the following form:

$$\mathsf{SensedCond}(A_i(\vec{x}_i), s, s') \stackrel{\text{def}}{=} \bigwedge_{j=1}^{n_i} \alpha_j(\vec{x}_i, s) \supset (\psi_j(\vec{x}_i, s) \equiv \psi_j(\vec{x}_i, s')), \tag{6.7}$$

where α_j is a condition under which A_i senses property ψ_j. Both $\alpha_j(\vec{x}_i, s)$ and $\psi_j(\vec{x}_i, s)$ are formulae uniform in s and are such that their free variables are among those in $\vec{x}$ and s. On the other hand, if A_i is *not* a sensing action, then it has the following form:

$$\mathsf{SensedCond}(A_i(\vec{x}), s, s') \stackrel{\text{def}}{=} True \tag{6.8}$$

Note that the resulting axiom for K is very similar in form to that of Reiter (2001). There are two main differences however. First, it contains the term $Poss(a, s')$ in the right hand side. This term also appears in Scherl and Levesque's axiom (6.5), and allows the agent to know the precondition of an action after performing it. Second, it is explicit about the fact that an action can *conditionally* sense possibly *multiple* formulae. Although Reiter's axiom (2001, Expression 11.7) does not seem to disallow multiple sensing effects, it is not in a form that allows conditional sensing effects. Finally, we insist that $\alpha_j(\vec{x}_i, s)$ and $\psi_j(\vec{x}_i, s)$ be uniform formulae; this condition was not imposed explicitly neither by Reiter (2001) nor by Scherl and Levesque (2003) but seems necessary for the SSA for K to have a form that resembles that of the SSAs for other fluents (cf. Expression 6.1).

Example 6.1 Let $sense_\phi$ and *senseLight* be as defined above. Moreover assume action *lookMonitor*

senses the value of both $Light_1$ and $Light_2$. Then, we define SensedCond in the following way.

$$\begin{aligned}
\mathsf{SensedCond}(sense_{\phi}, s, s') &\stackrel{\text{def}}{=} True \supset (\phi(s) \equiv \phi(s')) \\
\mathsf{SensedCond}(senseLight, s, s') &\stackrel{\text{def}}{=} [InRoom_1(s) \supset (Light_1(s) \equiv Light_1(s'))] \wedge \\
&\quad [InRoom_2(s) \supset (Light_2(s) \equiv Light_2(s'))], \\
\mathsf{SensedCond}(lookMonitor, s, s') &\stackrel{\text{def}}{=} [True \supset (Light_1(s) \equiv Light_1(s'))] \wedge \\
&\quad [True \supset (Light_2(s) \equiv Light_2(s'))],
\end{aligned}$$

6.2.4 Regression

Finally, our compilation procedure will make extensive use of a generalization of Reiter's *regression operator* (Reiter, 2001). The regression of $\alpha = \varphi(do([a_1, \ldots, a_n], S_0))$, denoted by $\mathcal{R}[\alpha]$, is a formula equivalent to α but such that the only situation terms occurring in it are S_0. Roughly, to regress a formula, one iteratively replaces each occurrence of fluent atomic formulae $F(x, do(a, s))$ by the right-hand side of Expression 6.1 until all atomic subformulae mention only situation S_0.

In this chapter, we need to use a more general version of regression with two main objectives. First, to produce the physical effects for our new primitive actions, we will need to be able to regress formulae in order to produce a formula that only mentions a situation term s, when s is not necessarily S_0. Second, we will need to regress formulae that contain situations that are in the future of multiple different situations (not just one, as in Reiter's definition) – the reason for this is that we will be obtaining knowledge effects of programs by regressing formulae that refer to equivalencies of the sort of those in Expression 6.7. To that end, we propose a generalization of Reiter's operator, $\mathcal{R}[W, \Upsilon]$, where W is a formula of the Situation Calculus and Υ is a set of situation terms. Intuitively, this operator regresses formulae that mention situation terms that may depend only on the terms in Υ. Additionally, the regression "stops" when situation terms mentioned in the formula are all in Υ.

Following Reiter (2001), we start off by defining *regressable* formulae in a set Υ of situation terms. Our definition extends Reiter's in the sense that our regressable formulae are those that refer to situations in the future of situations in Υ rather than only in the future of S_0. Other aspects of Reiter's definition are not changed.

Definition 6.2 (Regressable formula in a set of situations terms) *A formula W of the Situation Calculus is* regressable in a set of situation terms Υ *iff*

1. *Each term of the sort situation mentioned by W has the syntactic form $do([\alpha_1, \ldots, \alpha_n], s)$ for some $s \in \Upsilon$, and some $n \geq 0$, where $\alpha_1, \ldots, \alpha_n$ are terms of the sort action.*

2. *For each atom of the form Poss(α,σ) mentioned by W, α has the form $A(t_1,\ldots,t_n)$ for some action function symbol A.*

3. *W does not quantify over situations.*

4. *W does not mention the predicate symbol $\sqsubset$, nor does it mention any equality atom $\sigma = \sigma'$ for terms σ,σ' of the sort situation.*

5. *Each atom is either situation-independent, of the form Poss(α,σ), or of the form $F(t_1,\ldots,t_n,\sigma)$ for some fluent symbol F that is not K, some action term α, and some situation term σ.*

Note that this definition coincides with Reiter's Definition 4.5.1 (2001) when $\Upsilon = \{S_0\}$. Reiter's definition—since it was not designed to deal with the K fluent—does not insist on atoms being of the form in requirement number 5. This however, is necessary since our language contains K, which has an SSA whose right-hand side is not regressable because it quantifies over situations.

Our definition of the regression operator differs from Reiter's essentially only for the case of atomic formulae. There is an important property that atoms of a regressable formula satisfy:

Proposition 6.1 *Let W be a formula regressable in Υ. Then for any atom U in W, there exists a situation $s_U \in \Upsilon$, such that U is regressable in s_U.*

Proof: Since W is regressable in Υ, no atom of W can contain two different situation terms. Indeed, both the arguments to fluents atoms in W that are not the situation argument, and the arguments to action terms in a *Poss* atom of W may only be situation-independent terms since we do not deal with functional fluents. If an atom U of W contains 0 situation terms, the result follows immediately. If U contains 1 situation term, then this must be a term in the future of some situation in $s_U \in \Upsilon$. It follows that this atom is regressable in s_U. ■

To define $\mathcal{R}[W,\Upsilon]$ we slightly modify Reiter's definition (2001). Our definition differs from Reiter's in two aspects. The first is that we do not regress up to S_0 but only until all situation terms are in Υ. The second is that we do not regress terms that are in the future of a single situation but we allow to regress formulae that refer to situations in the future of multiple situations.

Definition 6.3 (Regression of a formula over a set of situation terms) *Let Υ be a set of situation terms. Furthermore, let W be a formula regressable in Υ that mentions no functional fluents. Then the regression of W over Υ, $\mathcal{R}[W,\Upsilon]$, is defined as follows.*

1. *Assume W is an atom then we have the following cases.*

   (a) *W is situation-independent, then*

   $$\mathcal{R}[W,\Upsilon] = W.$$

(b) W is of the form $F(t_1,\ldots,t_n,s)$ for some $s \in \Upsilon$, then

$$\mathcal{R}[W,\Upsilon] = W.$$

(c) If W is of the form $Poss(A(\vec{t}),\sigma)$,

$$\mathcal{R}[W,\Upsilon] = \mathcal{R}[\Pi_A(\vec{t},\sigma),\Upsilon].$$

(d) If W is of the form $F(\vec{x},do(\alpha,\sigma))$, then

$$\mathcal{R}[W,\Upsilon] = \mathcal{R}[\Phi_F(\vec{x},\alpha,\sigma),\Upsilon].$$

2. If W is a non-atomic formula, then the operator is defined as follows.

$$\mathcal{R}[\neg W,\Upsilon] = \neg\mathcal{R}[W,\Upsilon],$$
$$\mathcal{R}[W_1 \wedge W_2,\Upsilon] = \mathcal{R}[W_1,\Upsilon] \wedge \mathcal{R}[W_2,\Upsilon],$$
$$\mathcal{R}[(\exists v)\,W,\Upsilon] = (\exists v)\,\mathcal{R}[W,\Upsilon],$$

Proposition 6.2 *Let W be both regressable in Υ and regressable in Υ'. Assume further that Υ is a proper subset of Υ'. Then,*

$$\mathcal{R}[W,\Upsilon'] = \mathcal{R}[W,\Upsilon].$$

Proof: Follows straightforwardly by observing that W does not depend on any situation term in the future of any variable in $\Upsilon' \setminus \Upsilon$; otherwise, it would not be regressable in Υ. ■

As with Reiter's operator, we prove that by applying our operator we preserve the models of the formula given as argument, and that, furthermore, the resulting formula satisfies a uniformity condition.

Theorem 6.1 *Let W be a formula regressable in Υ. Let $\mathcal{D}$ be a basic action theory. Then $\mathcal{R}[W,\Upsilon]$ is uniform in Υ, and furthermore,*

$$\mathcal{D} \models \mathcal{R}[W,\Upsilon] \equiv W$$

Proof: First note that the regression operator is well-defined, in the sense that it always produces a situation calculus formula. The proof follows from Reiter's Theorem 4.5.4 (2001), and Propositions 6.1 and 6.2. First, observe that both our regression operator and Reiter's coincide for non-atomic formulae (modulo the extra parameter). Hence, their behaviour is only different at the atom level. On the other hand, by Proposition 6.1, we have that for any atom U in W, U is regressable in only one situation $s_U \in \Upsilon$. By Proposition 6.2 we have that $R[U,\Upsilon] = R[U,\{s_U\}]$. Moreover, $R[U,\{s_U\}]$ coincides with Reiter's operator, the only difference being that the root of the regression is not S_0 but s_U. The result now follows almost directly from (1) Reiter's Theorem 4.5.4 by noticing that the only difference for this case is the different root for the regression, and by (2) the fact that a logical combination of formulae uniform in a subset of Υ is uniform in Υ. ■

6.2.5 Golog's Syntax and Semantics

Golog is a high-level agent programming language whose semantics is based on the situation calculus (Reiter, 2001). A Golog program is a complex action[8]. For this chapter we focus on the Golog deterministic tree programs. Golog deterministic tree programs can be formed by applying the following constructs.

nil	– the empty program
a	– a primitive action
$\phi?$	– test action
$\delta_1;\delta_2$	– sequences (δ_1 is followed by δ_2)
if ϕ **then** δ_1 **else** δ_2 **endif**	– conditional

For the definition above, we assume that a is a primitive action of the form $A(\vec{t})$, where A is an action function symbol. Assume also that δ_1 and δ_2 are Golog deterministic tree programs. Finally, assume that $\phi[s]$ is a formula of the language that is regressable in s.

The Golog deterministic tree programs that we deal with in this chapter impose four restrictions on general Golog programs. The first and most obvious one is determinism. This restriction is necessary in order to reduce a program into a primitive action. The second two are the form of the primitive action a and the form of $\phi[s]$. This enables us to use regression over Golog programs later in this chapter, ultimately allowing us to extract the effects of Golog programs. Restricting conditions to be regressable still allows the user to define a wide range of formulae, in particular any boolean combination of formulae whose atoms are situation-suppressed fluents, including quantification. There are some formulae that are not allowed, but they arguably express less compelling conditions, that hardly appear in real applications. For example, we do not allow the condition ϕ to refer to *Poss* used on an action term that is not ground. We would not allow, for example, $(\forall a)\,Poss(a)$ as a condition in an *if-then-else* statement.

The final restriction is that of disallowing unbounded iteration. This restriction may seem too strong. Nevertheless, in practical applications most loops in terminating programs can be replaced by a bounded loop (i.e. a loop that is guaranteed to end after a certain number of iterations). Therefore, following McIlraith and Fadel (2002), we extend the Golog language with a *bounded loop* construct, defined as follows.

$$\textbf{while}_n\ \phi\ \textbf{do}\ \delta\ \textbf{endwhile} = \begin{cases} nil & \text{if } n = 0 \\ \textbf{if}\ \phi\ \textbf{then}\ \{\delta;\textbf{while}_{n-1}\ \phi\ \textbf{do}\ \delta\ \textbf{endwhile}\}\ \textbf{else}\ nil\ \textbf{endif} & \text{if } n > 0 \end{cases}$$

The semantics of Golog (Levesque *et al.*, 1997) is defined by the macro $Do(\delta, s, s')$ which expands into a formula that is true if and only if the execution of program δ in situation s leads to situation s'. *Do*

[8] We use the symbol δ to denote complex actions. ϕ is a situation-suppressed formula.

is defined as follows.

$$Do(nil, s, s') \stackrel{\text{def}}{=} s = s' \tag{6.9}$$

$$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a[s], s) \wedge s' = do(a[s], s) \tag{6.10}$$

$$Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s \tag{6.11}$$

$$Do(\delta_1; \delta_2, s, s') \stackrel{\text{def}}{=} (\exists s'').Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s') \tag{6.12}$$

$$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif}, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge Do(\delta_1, s, s') \vee \neg\phi[s] \wedge Do(\delta_2, s, s') \tag{6.13}$$

6.2.6 Do^-: A *Poss*-less Version of *Do*

In the rest of the chapter, we will be using regression on *Do* to obtain the preconditions and physical and knowledge effects of programs. Since the definition of *Do* includes *Poss*, when we regress it directly, we obtain an expression that includes the preconditions of the program. This is a feature that, as we see later, will enable us to obtain a precondition to the program by regressing *Do*. However, when using regression to obtain the effects of the program by regressing *Do* directly, we obtain that each effect of the program is conditioned on the program's precondition. Such an expression is redundant and thus undesirable. To address this issue we use a *Poss*-less version of *Do*, Do^-. Its definition follows.

$$Do^-(nil, s, s') \stackrel{\text{def}}{=} s = s' \tag{6.14}$$

$$Do^-(a, s, s') \stackrel{\text{def}}{=} s' = do(a[s], s) \tag{6.15}$$

$$Do^-(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s \tag{6.16}$$

$$Do^-(\delta_1; \delta_2, s, s') \stackrel{\text{def}}{=} (\exists s'').Do^-(\delta_1, s, s'') \wedge Do^-(\delta_2, s'', s') \tag{6.17}$$

$$Do^-(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif}, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge Do^-(\delta_1, s, s') \vee \neg\phi[s] \wedge Do^-(\delta_2, s, s') \tag{6.18}$$

Note that the only aspect in which Do^- differs from *Do* is in the formula for the primitive action a.

A rather obvious property of Do^- is that set of situations that are executions of a program δ under Do^- is a superset of those under *Do*. Formally,

Proposition 6.3 *Let δ be a Golog program and let $\mathcal{D}$ be a theory of action. Then,*

$$\mathcal{D} \models (\forall s, s').Do(\delta, s, s') \supset Do^-(\delta, s, s')$$

Proof: Fairly straightforward by induction in the structure of δ. ■

As we mentioned earlier, both *Do* and Do^- expand into a formula of the language. Both of these formulae can be put into a certain form that will later allow us to use regression. This is justified by the following result.

Lemma 6.1 *Let $\delta(\vec{y})$ be a Golog deterministic tree program whose free variables are among those in $\vec{y}$. Then there exists a set of sequences of primitive action terms $\{\vec{a}_1,\ldots,\vec{a}_n\}$, where each action term in each sequence is of the form $A(\vec{t})$, for some action function symbol A of the language, and there exists a set of formulae $\{\psi_1(\vec{y},s),\ldots,\psi_n(\vec{y},s)\}$, where each $\psi_i(\vec{y},s)$ is regressable in s, such that:*

$$Do(\delta(\vec{y}),s,s') \equiv \bigvee_{i=1}^{n} \psi_i(\vec{y},s) \wedge s' = do(\vec{a}_i,s)$$

Analogously, there exists a set of sequences of primitive action terms $\{\vec{b}_1,\ldots,\vec{b}_n\}$, where each action term in each sequence is of the form $A(\vec{t})$, for some action A of the language, and there exists a set of formulae $\{\mu_1(\vec{y},s),\ldots,\mu_n(\vec{y},s)\}$, where each $\mu_i(\vec{y},s)$ is regressable in s, such that:

$$Do^{-}(\delta(\vec{y}),s,s') \equiv \bigvee_{i=1}^{n} \mu_i(\vec{y},s) \wedge s' = do(\vec{b}_i,s)$$

Proof: The ψ_i (resp. μ_i) formulae can be constructed by unrolling the definition of *Do* (resp. Do^{-}). Note that in almost all cases for δ except for the sequence, the right-hand side of *Do* (resp. Do^{-}) provides a formula exactly in the form required above. For the sequence, we eliminate the existential quantifier by performing substitution of s''. ■

The intuition behind this result is central to the compilation method proposed in this chapter. Intuitively this result establishes that there are *n* possible situations that correspond to the execution of the program. The situation represented by $do(\vec{a}_i,s)$ is conditioned on formula ψ_i. Each of these *n* possible situations correspond to the execution of one of the branches of the tree program, and each condition ψ_i is the condition under which the branch is executable. Note also that since the program is deterministic, for every model $\mathcal{M}$ of $\mathcal{D}$ there is only one executable branch. Formally, this means that if $\psi_i(\vec{y},s)$ and $\psi_j(\vec{y},s)$ are such that both $\mathcal{M} \models \psi_i(\vec{y},s)$ and $\mathcal{M} \models \psi_j(\vec{y},s)$ then $\vec{b}_i = \vec{b}_j$.

6.3 Semantics for Executable Golog Programs

Again, as Levesque (1996) argued, when planning with sensing, the outcome of the planning process should be a plan which the executing agent knows at the outset will lead to a final situation in which the goal is satisfied. When planning with programs, as we are proposing here, we need to be able to determine when it is possible to execute a program with sensing actions and what situations could be the result of the program. Unfortunately, Golog's original semantics does not consider sensing actions and furthermore does not consider whether the agent has the ability to execute a given program.

Example 6.2 Let $\mathcal{D}$ be an action theory, then $\mathcal{D} \models \phi[S_0]$ and $\mathcal{D} \not\models \neg\phi[S_0]$, and let

$$\Delta \stackrel{\text{def}}{=} \textbf{if } \phi \textbf{ then } a \textbf{ else } b \textbf{ endif}.$$

Assume furthermore that a and b are always possible. Then, it holds that $\mathcal{D} \models (\exists s) Do(\Delta, S_0, s)$, i.e. δ is executable in S_0 (in fact, $\mathcal{D} \models Do(\Delta, S_0, s) \equiv s = do(a, S_0) \vee s = do(b, S_0)$). This fact is counter-intuitive since in S_0 the agent does not have enough information to determine whether ϕ holds, so Δ is not really executable.

As a *first objective* towards planning with programs that sense, we define what property a Golog program must satisfy to ensure it will be executable. Our *second objective* is to define a semantics that will enable us to determine the family of situations resulting from executing a program with sensing actions. This semantics provides the foundation for results in subsequent sections.

To achieve our first objective, we need to ensure that at each step of program execution, an agent has all the knowledge necessary to execute that step. In particular, we need to ensure that the program is *epistemically feasible*. Once we define the conditions under which a program is epistemically feasible, we can either use them as constraints on the planner, or we can ensure that our planner only builds plans using programs that are known to be epistemically feasible at the outset.

The problem of knowing how to execute a plan was addressed by Davis (1994). For Golog programs, the first approach—due to Lespérance, Levesque, Lin, and Scherl (2000)—, defines a predicate *CanExec* to establish when a program can be executed by an agent. A program can be executed by an agent if the agent possesses a strategy function σ that allows it to choose the right execution path. Using such a predicate, Lespérance *et al.* define the two notions of *knowing how* to execute a program. The second one—called "smart" know how—expresses that the agent knows how to execute a program iff there exists a strategy σ under which it can succeed executing the program. Lespérance *et al.* (2000) do not deal however with the problem of how can the agent determine such a strategy σ.

Sardina, de Giacomo, Lespérance, and Levesque (2004) define epistemically feasible programs using the online semantics of De Giacomo and Levesque (1999). Epistemic feasibility ensures that at each point in the execution the agent knows that there is a unique way of making a transition in the program. This notion does not require the concept of strategy and is much closer to what we want to achieve here.

Nevertheless because Sardina *et al.* (2004) define feasibility in an online rather than an offline setting, we prefer using a simpler, but slightly weaker definition proposed by McIlraith and Son (2002), which defines a self-sufficient property, *ssf*, such that $ssf(\delta, s)$ is true iff an agent knows how to execute program δ in situation s. We appeal to this property to characterize when a Golog program is executable. Its definition is given below.[9]

[9] We differ from the original definition in two aspects. First, in our case, we define *ssf* as a macro rather than a predicate. Second, we differ on the definition of Expression 6.20, since we do not contemplate the so-called desirable actions.

$$ssf(nil,s) \stackrel{\text{def}}{=} True \tag{6.19}$$

$$ssf(a,s) \stackrel{\text{def}}{=} \mathbf{KWhether}(Poss(a),s), \tag{6.20}$$

$$ssf(\phi?,s) \stackrel{\text{def}}{=} \mathbf{KWhether}(\phi,s) \tag{6.21}$$

$$ssf(\delta_1;\delta_2,s) \stackrel{\text{def}}{=} ssf(\delta_1,s) \wedge (\forall s').Do(\delta_1,s,s') \supset ssf(\delta_2,s') \tag{6.22}$$

$$\begin{aligned} ssf(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif},s) \stackrel{\text{def}}{=} \mathbf{KWhether}(\phi,s) \wedge \\ (\phi[s] \supset ssf(\delta_1,s)) \wedge (\neg\phi[s] \supset ssf(\delta_2,s)) \end{aligned} \tag{6.23}$$

Self-sufficiency is weaker than epistemic feasibility because it sometimes may require knowledge of certain properties when there is actually no need to require such knowledge. Differences are apparent, however, only in very contrived cases. For example, the Golog program **if** ϕ **then** a **else** a **endif** is not self sufficient in a situation s in which $\neg\mathbf{KWhether}(\phi,s)$ but it *is* epistemically feasible, because no matter what the agent knows there is a unique way of completing the program by performing a. We do not view these differences however as an obstacle to our main purpose.

We now focus on our second objective, i.e. to define a semantics for Golog programs with sensing actions. To our knowledge, no such semantics exists. Nevertheless, there is related work. De Giacomo and Levesque (1999) define the semantics of programs with sensing in an *online* manner, i.e. it is determined during the execution of the program. An execution is formally defined as a mathematical object, and the semantics of the program depends on such an object. The semantics is thus defined in the metalanguage, and therefore it is not possible to refer to the situations that would result from the execution of a program within the language.

To define a semantics for executable programs with sensing, we modify the existing Golog semantics so that it refers to the knowledge of the agent, defining a Do_K macro. This new macro is such that $Do_K(\delta,s,s')$ expands into a formula that is true if and only if the agent has the sufficient knowledge to perform program δ in s, and gets to situation s' after doing so.

$$Do_K(nil,s,s') \stackrel{\text{def}}{=} s = s' \tag{6.24}$$

$$Do_K(a,s,s') \stackrel{\text{def}}{=} \mathbf{Knows}(Poss(a),s) \wedge s' = do(a[s],s) \tag{6.25}$$

$$Do_K(\phi?,s,s') \stackrel{\text{def}}{=} \mathbf{Knows}(\phi,s) \wedge s' = s \tag{6.26}$$

$$Do_K(\delta_1;\delta_2,s,s') \stackrel{\text{def}}{=} (\exists s'').Do_K(\delta_1,s,s'') \wedge Do_K(\delta_2,s'',s') \tag{6.27}$$

$$\begin{aligned} Do_K(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif},s,s') \stackrel{\text{def}}{=} \mathbf{Knows}(\phi,s) \wedge Do_K(\delta_1,s,s') \vee \\ \mathbf{Knows}(\neg\phi,s) \wedge Do_K(\delta_2,s,s') \end{aligned} \tag{6.28}$$

In contrast to Do, Do_K of an *if-then-else* explicitly requires the agent to know the value of the condition. Returning to Example 6.2, if now $\mathcal{D} \not\models \mathbf{KWhether}(\phi,S_0)$, then $\mathcal{D} \models \neg(\exists s)\,Do_K(\Delta,S_0,s)$. However, if $sense_\phi$ senses ϕ, then $\mathcal{D} \models (\exists s)\,Do_K(sense_\phi;\Delta,S_0,s)$.

A natural question to ask is when this semantics is equivalent to the original semantics. We can prove that both are equivalent for self-sufficient programs (in the sense of McIlraith and Son (2002)).

Lemma 6.2 *Let $\mathcal{D}$ be a theory of action such that the K fluent is reflexive, and let δ be a Golog deterministic tree program. Then,*

$$\mathcal{D} \models (\forall s).ssf(\delta, s) \supset \{(\forall s').Do(\delta, s, s') \equiv Do_K(\delta, s, s')\}$$

Proof: See Section D.1 (p. 184). ■

The preceding lemma is fundamental to the rest of our work. In the following sections we show how theory compilation relies strongly on the use of regression of the Do_K predicate. Given our equivalence we can now regress Do instead of Do_K which produces significantly simpler formulae.

An important point is that the equivalence of the semantics is achieved for self-sufficient programs. Proving that a program is self-sufficient may be as hard as doing the regression of Do_K. Fortunately, there are syntactic accounts of self-sufficiency (McIlraith and Son, 2002; Sardina *et al.*, 2004), such as programs in which each *if-then-else* and *while* loop that conditions on ϕ is preceded by a $sense_\phi$, or more generally that knowledge about ϕ is established prior to these constructs and persists until their usage.

6.4 Planning with Programs that Sense

We now return to the main objective of this chapter – how to plan with programs that sense by enabling operator-based planners to treat programs as black-box primitive actions. A plan in the presence of sensing is a program that may contain conditionals and loops (Levesque, 1996). As such, we define a plan as a Golog program.

Definition 6.4 (A plan) *Given a theory of action $\mathcal{D}$, and a goal G we say that Golog program δ is a plan for situation-suppressed formula G in situation s relative to theory $\mathcal{D}$ iff*

$$\mathcal{D} \models (\exists s')Do_K(\delta, s, s') \wedge (\forall s').Do_K(\delta, s, s') \supset G[s'].$$

Intuitively, the Golog program δ is a plan if it terminates and achieves the goal.

In classical planning, a planner constructs plan δ by choosing actions from a set A of primitive actions. Here we assume the planner has an additional set C of programs from which to construct plans.

Example 6.3 Consider an agent that uses the following complex action to paint objects:

$$\delta(o) \stackrel{\text{def}}{=} sprayPaint(o); look(o);$$
$$\textbf{if } \neg wellPainted(o) \textbf{ then } brushPaint(o) \textbf{ else } nil \textbf{ endif}$$

The action $sprayPaint(o)$ paints an object o with a spray gun, and action $brushPaint(o)$ paints it with a brush. We assume that action $sprayPaint(o)$ well-paints o if the spray is not malfunctioning, whereas action $brushPaint(o)$ always well-paints o (this agent prefers spray-painting for cosmetic reasons). Action $look(o)$ is a sense action that senses whether or not o is well painted.

Below we show some axioms in $\mathcal{D}_{ap}$ and $\mathcal{D}_{eff}$ that are relevant for our example.

$$Poss(sprayPaint(o), s) \equiv have(o),$$
$$Poss(look(o), s) \equiv have(o),$$
$$a = sprayPaint(o) \wedge \neg malfunct(s) \supset wellPainted(o, do(a, s)),$$
$$a = brushPaint(o) \supset wellPainted(o, do(a, s))$$
$$a = scratch(o) \supset \neg wellPainted(o, do(a, s))$$

The SSAs for the fluents *wellPainted* is as follows.

$$wellPainted(x, do(a, s)) \equiv$$
$$a = brushPaint(x) \vee (a = sprayPaint(x) \wedge \neg malfunct(s)) \vee$$
$$wellPainted(x, s) \wedge a \neq scratch(x),$$

Finally, action $look(x)$ informs the agent of whether or not x is well painted. We achieve this by defining the following sensed condition:

$$\mathsf{SensedCond}(look(o), s, s') \stackrel{\text{def}}{=} wellPainted(x, s) \equiv wellPainted(x, s').$$

Furthermore, for all remaining primitive actions SensedCond is defined as $True$.

The SSA for *wellPainted* says that x is well painted if it has just been brush painted, or it has just been spray painted and the spray is not malfunctioning or if it was well painted in the preceding situation and x has not been scratched. On the other hand, the SSA for K talks about a unique sensing action, $look(x)$, which senses whether x is well painted.

Suppose we want to use action δ to construct a plan using an operator-based planner. Instead of a program, we would need to consider δ's effects and preconditions (i.e. we would need to represent δ as a primitive action). Among the effects we must describe both physical effects (e.g., after we perform $\delta(B)$, B is *wellPainted*) and knowledge effects. A rather non-trivial knowledge effect is that if we know that o is not *wellPainted*, after we perform $\delta(B)$, we know whether or not *malfunct*.

The rest of this section presents a method that, under certain conditions, transforms a theory of action $\mathcal{D}$ and a set of programs with sensing C into a new theory, $\mathsf{Comp}[\mathcal{D}, C]$, that describes the same domain as $\mathcal{D}$ but that is such that programs in $\mathsf{Comp}[\mathcal{D}, C]$ each appear modeled by a new primitive action.

6.4.1 Theory Compilation

A program with sensing may produce both effects in the world and in the knowledge of the agent. Therefore, we want to replace a program δ by one primitive action $prim_\delta$, with the same preconditions and the same physical and knowledge effects as δ. We now describe how we can generate a new theory of action that contains this new action. Then we prove that when $prim_\delta$ is executed, it captures all world and knowledge-level effects of the original program δ.

Our translation process is restricted to tree programs. This is because programs containing loops can be problematic since they may have arbitrarily long executions. However, in many practical cases loops can be replaced by bounded loops using the **while**$_n$ construct introduced in Section 6.2.

We start with a theory $\mathcal{D} = \Sigma \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{K}_{init}$, describing a set $\mathcal{A}$ of primitive actions and a set C of tree programs, and we generate a new theory $\mathsf{Comp}[\mathcal{D}, C]$ that contains new, updated SSA, precondition and unique name axioms.

We make the following assumptions. First we assume that the set of successor state axioms, $\mathcal{D}_{ss}$, has been compiled from set $\mathcal{D}_{eff}$ of effect axioms. Furthermore, we assume we are given a set C of deterministic tree programs $\delta(\vec{y})$ that contain free variables among those in $\vec{y}$. Vector $\vec{y}$ can be seen as the parameters for the complex action. Finally, each program $\delta(\vec{y})$ is such that $\mathcal{D} \models (\forall s, \vec{y}).\, ssf(\delta(\vec{y}), s)$.

Moreover, we want $prim_\delta$ to preserve the physical effects of δ. To that end, for each fluent, we add effect axioms for $prim_\delta$ such that whenever δ makes F true/false, $prim_\delta$ will also make it true/false. Finally, because we want to preserve knowledge effects of δ, $prim_\delta$ will emulate δ with respect to the K fluent. To write these new axioms we use the regression operator $\mathcal{R}[\cdot,\cdot]$ of Section 6.2 because we will need that precondition and effect axioms only talk about situation s. We generate the new theory $\mathsf{Comp}[\mathcal{D}, C]$ in the following way. First, we set $\mathcal{D}'_{eff} := \mathcal{D}_{eff}$, $\mathcal{D}'_{ap} := \mathcal{D}_{ap}$, and $\mathcal{D}'_{una} := \mathcal{D}_{una}$. The definition of the new preconditions and physical and knowledge effects for the new primitive actions follow.

New Preconditions

Intuitively, since $prim_\delta(\vec{y})$ replaces $\delta(\vec{y})$, we want $prim_\delta(\vec{y})$ to be executable in s precisely when $\delta(\vec{y})$ is executable in s. Note that program $\delta(\vec{y})$ is executable in s iff there exists a situation s' that corresponds to the execution of the program in s. In other words, $prim_\delta(\vec{y})$ is executable in s iff

$$(\exists s')\, Do_K(\delta(\vec{y}), s, s') \tag{6.29}$$

is entailed by $\mathcal{D}$. Nevertheless, we cannot add the formula in Expression 6.29 directly as a precondition since this formula is not uniform in s. To obtain a formula uniform in s we would want to appeal to regression but unfortunately Expression 6.29 is not regressable!

We appeal now to results proven earlier in this chapter to transform Expression 6.29 into an equivalent formula that *is* regressable in s. First, by Lemma 6.2 and the fact that $\delta(\vec{y})$ is self-sufficient, we

know Expression 6.29 is equivalent to

$$(\exists s')\, Do(\delta(\vec{y}), s, s'). \tag{6.30}$$

Finally we can transform Expression 6.30 into an equivalent formula that is regressable in s, as shown by the following remark.

Remark 6.1 *Let $\psi_1, \psi_2, \ldots, \psi_n$ be the formulae of Lemma 6.1. Expression 6.30 is equivalent to:*

$$\bigvee_{i=1}^{n} \psi_i(\vec{y}, s) \tag{6.31}$$

Proof: By Lemma 6.1, Expression 6.30 is equivalent to $(\exists s') \bigvee_{i=1}^{n} \psi_i(\vec{y}, s) \wedge s' = do(\vec{a}_i, s)$. Then, we apply quantifier elimination of the existential quantifier, which also implies eliminating all terms of the form $s' = do(\vec{a}_i, s)$. ■

Intuitively, the condition of Expression 6.31 summarizes the existence of a branch of the program that is executable. Since $\psi_i(\vec{y}, s)$ is regressable in s, we formulate the precondition for $prim_\delta$ as follows:

$$Poss(prim_\delta(\vec{y}), s) \equiv \mathcal{R}[\bigvee_{i=1}^{n} \psi_i(\vec{y}, s), s], \tag{6.32}$$

where ψ_i are the formulae of Lemma 6.1. Thus, $prim_\delta(\vec{y})$ can be executed iff program δ could be executed in s.

Finally note that this precondition axiom is in the form of Expression 6.3, i.e., it is a proper precondition axiom because the right-hand side is a formula uniform in s. This fact follows from Theorem 6.1.

New Physical Effects

We now turn our attention to the effects of $prim_\delta(\vec{y})$. Intuitively, we want to say here that whenever s' is an execution of the program in s and fluent $F(\vec{x})$ is true in such an s', then we want $F(\vec{x})$ to be an effect of $prim_\delta(\vec{y})$. More precisely, $F(\vec{x})$ is true after performing $prim_\delta(\vec{y})$ in s iff

$$(\exists s').\, Do_K(\delta(\vec{y}), s, s') \wedge F(\vec{x}, s') \tag{6.33}$$

Note that since the program is deterministic, in each model of the theory, there is only one situation that can be referred by s' in Expression 6.33.[10] As in the previous step, it is not possible to add this condition directly as an effect axiom since the Expression 6.33 is not uniform in s. Expression 6.33 is not directly regressable in s either. As a consequence, we again appeal to Lemma 6.2, to obtain the equivalent condition:

$$(\exists s').\, Do(\delta(\vec{y}), s, s') \wedge F(\vec{x}, s') \tag{6.34}$$

[10] This means actually that Expression 6.33 is equivalent to $(\exists s')\, Do_K(\delta(\vec{y}), s, s') \wedge (\forall s').\, Do_K(\delta(\vec{y}), s, s') \supset F(\vec{x}, s')$. We prefer to work with Expression 6.33 however, because it is simpler.

Here, we could apply Lemma 6.1 in order to obtain a regressable condition, but by doing so, we would obtain a formula that encodes the preconditions of $\delta(\vec{y})$. The reason for this is that *Do* makes reference to *Poss* whenever a primitive action is executed. Hence, the condition for $F(\vec{x})$ to be true after performing $prim_\delta$ in s will be the following one:

$$(\exists s').Do^-(\delta(\vec{y}),s,s') \wedge F(\vec{x},s') \tag{6.35}$$

Recall that the only difference between Do^- and *Do* is that the former does not check the preconditions of the actions that are being performed. This implies that the set of situations that are executions under Do^- is a superset of the set of situations that are executions under *Do* (this means that Expression 6.34 implies Expression 6.35). Since our programs are deterministic, however, only a unique situation can result from the execution of the program in a particular model of the theory $\mathcal{D}$ under Do^-. Thus, given a model of the theory, Do^- and *Do* could only differ in the executions of δ iff δ is *not* executable in s. Otherwise, both formulae coincide in terms of the situations that are regarded as executions. As a result, by using Do^- instead of *Do* we may define the effects of $prim_\delta$ in cases in which δ is not executable in s. This is not a problem, since we only will consider performing δ when its preconditions are satisfied. Indeed, our definition of plan (Def. 6.4) ensures that this is the case.

As with the preconditions, the final step is to transform Expression 6.35 into a regressable formula. We can do this using Lemma 6.1.

Remark 6.2 *Let $\mu_1,\ldots,\mu_n$ and $\vec{b}_1,\ldots,\vec{b}_n$ be respectively the formulae and action sequences of Lemma 6.1. Then, Expression 6.35 is equivalent to the following expression regressable in s.*

$$\bigvee_{i=1}^{n} \mu_i(\vec{y},s) \wedge F(\vec{x},do(\vec{b}_i,s)) \tag{6.36}$$

Proof: Follows directly by substituting s' in $F(\vec{x},s')$ in the formula of Lemma 6.1 and then eliminating the existential quantifier. Since $\mu_i(\vec{y},s)$ is regressable in s, the resulting expression is clearly regressable in s. ■

The new effect axioms for F are generated as follows. For each relational fluent $F(\vec{x},s)$ in the language of $\mathcal{D}$ that is not the K fluent, and each complex action $\delta(\vec{y}) \in C$ we add the following positive and negative effect axioms to $\mathcal{D}'_{eff}$:

$$a = prim_\delta(\vec{y}) \wedge \mathcal{R}[\bigvee_{i=1}^{n} \mu_i(\vec{y},s) \wedge F(\vec{x},do(\vec{b}_i,s)),s] \supset F(\vec{x},do(a,s)), \tag{6.37}$$

$$a = prim_\delta(\vec{y}) \wedge \mathcal{R}[\bigvee_{i=1}^{n} \mu_i(\vec{y},s) \wedge \neg F(\vec{x},do(\vec{b}_i,s)),s] \supset \neg F(\vec{x},do(a,s)), \tag{6.38}$$

where $\mu_1,\ldots\mu_n$ and $\vec{b}_1,\ldots,\vec{b}_n$ are respectively the formulae and action sequences of Lemma 6.2.

New Sensed Condition

We now turn our attention to the generation of the new sensed condition for action $prim_\delta$. In order to capture all the knowledge effects of δ, our new action $prim_\delta$ must emulate δ's dynamics with respect to the K fluent. More precisely, suppose we perform δ in situation s. Assume further that s' is K accessible from s, and that performing δ in s leads to situation $do([a_1,\ldots,a_n],s)$. If $do([a_1,\ldots,a_n],s')$ is also K-accessible from $do([a_1,\ldots,a_n],s)$, then we want to replicate this by making $do(prim_\delta,s')$ K-accessible from $do(prim_\delta,s)$.

It is rather simple to find a condition independent of K that expresses the necessary and sufficient conditions under which a situation $do([a_1,\ldots,a_n],s')$ is K-accessible from $do([a_1,\ldots,a_n],s)$ given that s' is K-accessible from s. This is given by the following result.

Proposition 6.4 *Let $\mathcal{D}$ be a theory of action, let δ be a Golog deterministic tree program, and let $a_1,\ldots,a_n$ be action terms. Then the following holds:*

$$\mathcal{D} \models (\forall s',s).K(s',s) \wedge Do_K(\delta,s,do([a_1,\ldots,a_n],s)) \supset \{K(do([a_1,\ldots,a_n],s'),do([a_1,\ldots,a_n],s)) \equiv \bigwedge_{i=1}^{n} \mathsf{SensedCond}(a_i,do([a_1,\ldots,a_{i-1}],s),do([a_1,\ldots,a_{i-1}],s'))\} \quad (6.39)$$

Proof: The proof is by induction on n. First we prove that for all $1 \leq i \leq n$,

$$\mathcal{D} \models (\forall s',s).K(s',s) \wedge Do_K(\delta,s,do([a_1,\ldots,a_n],s)) \supset Poss(a_i,do([a_1,\ldots,a_{i-1}],s').$$

Then, the proof is rather straightforward using the successor state axiom for K and simplifying away the terms containing *Poss*. ■

Note that this result suggests that the sensed condition for a new action $prim_\delta$ corresponds precisely to

$$\bigwedge_{i=1}^{n} \mathsf{SensedCond}(a_i,do([a_1,\ldots,a_{i-1}],s),do([a_1,\ldots,a_{i-1}],s')) \quad (6.40)$$

when the situation that results from the execution of δ is $do([a_1,\ldots,a_n],s)$.

Our last step for the definition of the sensed condition of an action involves relating Expression 6.40 with the situations that actually refer to the executions of the program δ (for now we've been using $do([a_1,\ldots,a_{i-1}],s)$ to show the structure of an execution only). As we have done for the physical effects, we characterize the situations that may result from the execution of δ as all situations s' that satisfy the formula:

$$Do^-(\delta,s,s') \quad (6.41)$$

As in the case of physical effects, by using Do^- instead of Do we may define the knowledge effects of $prim_\delta$ in cases in which δ is not executable in s; this is not a problem here either because we only allow the application of possible actions when planning.

Recall that by Lemma 6.1 we obtain explicit situation terms $do(\vec{b}_1, s), \ldots, do(\vec{b}_n, s)$ that refer to the execution of δ. These situations are precisely the ones we use define the sensed condition of $prim_\delta$.

For each $\delta(\vec{y}) \in C$, our new theory will contain the following definition for the sensed condition of $prim_\delta(\vec{y})$.

$$\mathsf{SensedCond}(prim_\delta(\vec{y}), s, s') \stackrel{\text{def}}{=}$$
$$\mathcal{R}\big[\bigvee_{i=1}^{n} \mu_i(\vec{y}, s) \wedge \bigwedge_{j=1}^{|\vec{b}_i|} \mathsf{SensedCond}(b_{ij}, do(\vec{b}_i|_1^{j-1}, s), do(\vec{b}_i|_1^{j-1}, s')), \{s, s'\}\big], \quad (6.42)$$

where $\mu_1, \ldots \mu_n$ and $\vec{b}_1, \ldots, \vec{b}_n$ are respectively the formulae and action sequences of Lemma 6.2. Furthermore b_{ij} denotes the j-th element of sequence $\vec{b}_i$ and $\vec{b}_i|_\ell^k$ denotes the subsequence of $\vec{b}_i$ that contains all elements between the ℓ-th and the k-th element. Finally, $|\vec{b}|$ denotes the number of action terms in $\vec{b}$.

Intuitively, the right-hand side of Expression 6.42 establishes that $do(prim_\delta, s')$ is K-accessible from $do(prim_\delta, s)$ only if s' is accessible from s and there is one executable branch, characterized by $\vec{b}_i$ conditioned on $\mu_i(\vec{y}, s)$, that satisfies the condition in Expression 6.40 that we discussed above. Note that because the program is deterministic, in a particular model of the theory only one branch actually corresponds to an execution. Thus the external disjunction captures the fact that there are only n ways in which the program may be executed; as we have been insisting in this chapter, this *does not* mean that two different branches may be executed at the same time.

Finally, note that because $\mathsf{SensedCond}(a, s, s')$ expands into a formula that is uniform in s and s', $\mathcal{R}$ in Expression 6.42 is applied over a formula that is regressable in s and s'. Thus, by Theorem 6.1 it follows that the sensing condition for our new action is a formula uniform in s and s', and hence has the form that we require in Expression 6.7.

New Unique Names Axioms

For each $\delta(\vec{y}), \delta'(\vec{x}) \in C$ such that $\delta(\vec{y}) \neq \delta'(\vec{x})$ add $prim_\delta(\vec{x}) \neq prim_{\delta'}(\vec{y})$ to $\mathcal{D}'_{una}$. For each action A of the language of the original theory $\mathcal{D}$ and each $\delta(\vec{y}) \in C$, add $A(\vec{x}) \neq prim_\delta(\vec{y})$ to $\mathcal{D}'_{una}$.

The New Basic Action Theory

Compile a new set of SSAs $\mathcal{D}'_{ss}$ from $\mathcal{D}'_{eff}$, and replace the successor state axiom for K with the one that refers to the actions in $\mathcal{D}'_{ss}$, which in particular will now refer to the sensed conditions of the actions $prim_\delta(\vec{y})$ for all $\delta(\vec{y}) \in C$. The new theory is defined as follows.

$$\mathsf{Comp}[\mathcal{D}, C] = \Sigma \cup \mathcal{D}'_{ss} \cup \mathcal{D}'_{ap} \cup \mathcal{D}'_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{K}_{init}.$$

We now turn to the analysis of some properties of the resulting theory $\mathsf{Comp}[D, C]$.

Theorem 6.2 *If $\mathcal{D}$ is consistent and C contains only deterministic tree programs then* $\mathsf{Comp}[\mathcal{D},C]$ *is consistent.*

Proof: The consistency property (Reiter, 2001, pg. 31) follows from the form of the effect axioms and fact that the programs we are considering are deterministic. ■

Now we establish a complete correspondence at the physical level between our original programs and the compiled primitive actions after performing $prim_\delta$.

Theorem 6.3 *Let $\mathcal{D}$ be a theory of action such that $\mathcal{K}_{init}$ contains the reflexivity axiom. Let C be a set of deterministic Golog tree programs. Finally, let ϕ be a situation-suppressed formula such that $\phi[s]$ is regressable in s. Then,*

$$\mathsf{Comp}[\mathcal{D},C] \models (\forall s,s').Do_K(\delta,s,s') \supset (\phi[s'] \equiv \phi[do(prim_\delta,s)])$$

Proof: See Section D.2 (p. 184). ■

It is worth noting that the preceding theorem is also valid when δ does not contain sensing actions.

Also, there is a complete correspondence at a knowledge level between our original complex actions and the compiled primitive actions after performing $prim_\delta$.

Theorem 6.4 *Let $\mathcal{D}$ be a theory of action such that $\mathcal{K}_{init}$ contains the reflexivity axiom. Let C be a set of deterministic Golog tree programs, and ϕ be a situation-suppressed formula such that $\phi[s]$ is regressable in s. Then,*

$$\mathsf{Comp}[\mathcal{D},C] \models (\forall s,s_1).Do_K(\delta,s,s_1) \supset \{\mathbf{Knows}(\phi,s_1) \equiv \mathbf{Knows}(\phi,do(prim_\delta,s))\}. \tag{6.43}$$

Proof: See the Section D.3 (p. 185). ■

Now that we have established the correspondence between $\mathcal{D}$ and $\mathsf{Comp}[\mathcal{D},C]$ we return to planning. In order to achieve a goal G in a situation s, we now obtain a plan using theory $\mathsf{Comp}[\mathcal{D},C]$. In order to be useful, this plan should have a counterpart in $\mathcal{D}$, since the executor cannot execute any of the "new" actions in $\mathsf{Comp}[\mathcal{D},C]$. The following result establishes a way to obtain such a counterpart.

Theorem 6.5 *Let $\mathcal{D}$ be a theory of action, C be a set of deterministic Golog tree programs, and G be a formula of the Situation Calculus. Then, if Δ is a plan for G in theory* $\mathsf{Comp}[D,C]$ *and situation s, then there exists a plan Δ' for G in theory $\mathcal{D}$ and situation s. Moreover, Δ' can be constructed from Δ.*

Proof sketch: We construct Δ' by replacing every occurrence of $prim_\delta$ in Δ by δ. Then we prove that Δ' also achieves the goal, from Theorems 6.3 and 6.4. ■

Example 6.3 (cont.) The result of applying theory compilation to the basic action theory of our example follows.

By unrolling of Do and Do^- for $\delta(o)$, we obtain the following formulae and action sequences that correspond to Lemma 6.1:

$$\begin{aligned}\psi_1(o,s) = {} & Poss(sprayPaint(o),s)\wedge \\ & Poss(look(o),do(sprayPaint(o),s))\wedge \\ & \neg wellPainted(o,do([sprayPaint(o),look(o)],s))\wedge \\ & Poss(brushPaint(o),do([sprayPaint(o),look(o)],s)),\end{aligned} \tag{6.44}$$

$$\begin{aligned}\psi_2(o,s) = {} & Poss(sprayPaint(o),s)\wedge \\ & Poss(look(o),do(sprayPaint(o),s))\wedge \\ & wellPainted(o,do([sprayPaint(o),look(o)],s)),\end{aligned} \tag{6.45}$$

with

$$\vec{a}_1 = [sprayPaint(o),look(o),brushPaint(o)], \tag{6.46}$$

$$\vec{a}_2 = [sprayPaint(o),look(o)]. \tag{6.47}$$

On the other hand, for Do^- we obtain the following:

$$\mu_1(o,s) = \neg wellPainted(o,do([sprayPaint(o),look(o)],s)) \tag{6.48}$$

$$\mu_2(o,s) = wellPainted(o,do([sprayPaint(o),look(o)],s)) \tag{6.49}$$

with

$$\vec{b}_1 = [sprayPaint(o),look(o),brushPaint(o)], \tag{6.50}$$

$$\vec{b}_2 = [sprayPaint(o),look(o)]. \tag{6.51}$$

New Precondition By simplifying the expression of Expression 6.32, we obtain the following precondition axiom for $prim_\delta$:

$$Poss(prim_\delta(o),s) \equiv have(o,s). \tag{6.52}$$

New Effect and Successor State Axioms For the fluent *wellPainted*, the negative effect axiom of the form of Expression 6.38 simplifies to:

$$\begin{aligned}a = prim_\delta(o) \wedge \mathcal{R}[(\neg wellPainted(o,S_2) \wedge \neg wellPainted(o,S_1)) \vee \\ (wellPainted(o,S_2) \wedge \neg wellPainted(o,S_2)),s] \supset \\ \neg wellPainted(o,do(a,s)),\end{aligned}$$

where $S_1 = do([sprayPaint(o), look(o), brushPaint(o)], s)$ and $S_2 = do([sprayPaint(o), look(o)], s)$. It is easy to verify that:

$$\mathcal{R}[wellPainted(o, S_1), s] = True,$$
$$\mathcal{R}[wellPainted(o, S_2), s] = \neg malfunct(s) \vee wellPainted(o, s).$$

Substituting, the negative action reduces to the futile axiom:

$$a = prim_\delta(o) \wedge False \supset \neg wellPainted(o, do(a, s)). \tag{6.53}$$

Similarly, the positive effect axiom obtained for *wellPainted*:

$$a = prim_\delta(o) \wedge \mathcal{R}[(\neg wellPainted(o, S_2) \wedge wellPainted(o, S_1)) \vee (wellPainted(o, S_2) \wedge wellPainted(o, S_2))] \supset \neg wellPainted(o, do(a, s)),$$

where S_1 and S_2, are defined as before, reduces to

$$a = prim_\delta(o) \supset wellPainted(o, do(a, s)). \tag{6.54}$$

Considering the new effect axiom for $prim_\delta(o)$, the new SSA for *wellPainted* is therefore:

$$wellPainted(o, do(a, s)) \equiv a = brushPaint(o) \vee a = prim_\delta(o) \vee a = sprayPaint(o) \wedge \neg malfunct(s) \vee wellPainted(o, s) \wedge a \neq scratch(o),$$

which means that o is well painted after $prim_\delta(o)$ is performed.

New Sensed Condition Now we focus our attention on the K axiom. The Expression 6.42 simplifies to:

$$\mathsf{SensedCond}(prim_\delta(o), s, s') \stackrel{\text{def}}{=} \mathcal{R}\left[\mu_1(o, s) \wedge (wellPainted(o, S_2) \equiv wellPainted(o, S_2'))) \vee \mu_2(o, s) \wedge (wellPainted(o, S_2) \equiv wellPainted(o, S_2'))), \{s, s'\}\right]$$

where S_2 is as defined above and $S_2' = do([sprayPaint(o), look(o)], s')$. After performing regression, the expression simplifies into:

$$\mathsf{SensedCond}(prim_\delta(o), s, s') \stackrel{\text{def}}{=} (malfunct(s) \wedge \neg wellPainted(o, s)) \equiv (malfunct(s') \wedge \neg wellPainted(o, s'))$$

Thus, our new primitive action senses the truth value of the (situation-suppressed) formula $\mathit{malfunct} \wedge \neg \mathit{wellPainted}(o)$ in the current situation. Observe that sensing the truth value of this formula is equivalent to sensing the truth value of its negation, which can be written as $\neg \mathit{wellPainted}(o) \supset \neg \mathit{malfunct}$.

Clearly, the process has captured the world-altering effect of $\delta(o)$, namely that $\mathit{wellPainted}(o)$. Moreover, it is easy to confirm a conditional knowledge effect:

$$\textbf{Knows}(\neg \mathit{wellPainted}(o), s) \supset \textbf{KWhether}(\mathit{malfunct}, do(\mathit{prim}_{\delta}(o), s)).$$

Note that our theory compilation can only be used for complex actions that can be proved self-sufficient for all situations. As noted previously, an alternative was to use the conditions that need to hold true for a program to be self-sufficient as a precondition for the newly generated primitive actions. Indeed, formula $\mathit{ssf}(\delta, s)$ encodes all that is required to hold in s to be able to know how to execute δ, and therefore we could have added something like $\mathit{Poss}(\mathit{prim}_{\delta}(\vec{y}), s) \equiv \mathcal{R}[(\exists s')\, \mathit{Do}(\delta, s, s') \wedge \mathit{ssf}(\delta, s), s]$ instead of Expression 6.32. This modification keeps the validity of Theorems 6.3 and 6.4 only if no actions of the original theory have preconditions that mention knowledge, and provided we extend regression for **Knows** following Scherl and Levesque (2003). The resulting precondition however, may contain complex formulae referring to the knowledge of the agent, which we view as problematic for practical applications. On the other hand, Reiter's version of the Situation Calculus does not allow actions with knowledge preconditions. The good news is that most Web services are self-sufficient by design.

Finally, the compilation method we have described here is only defined for programs that contain primitive actions, i.e. it does not allow programs to invoke other programs. However, the method can be extended for a broad class of programs that include such calls. If there are no unbounded recursions or the programs can be stratified with respect to recursive calls, it is always possible to iteratively apply the compilation method presented until all programs have been reduced to a primitive action.

6.5 From Theory to Practice

We have shown that under certain circumstances, planning with programs can be in theory reduced to planning with primitive actions. In this section we identify properties necessary for operator-based planners to exploit these results, with particular attention to some of the more popular existing planners. There are several planning systems that have been proposed in the literature that are able to consider the knowledge of an agent and (in some cases) sensing actions. These include Sensory Graphplan (SGP) (Weld, Anderson, and Smith, 1998), the MDP-based planner GPT (Bonet and Geffner, 2000), the model-checking-based planner MBP[11] (Bertoli, Cimatti, Roveri, and Traverso, 2001), the logic-programming-

[11] MBP does not consider sensing actions explicitly, however they can be 'simulated' by representing within the state the last action executed.

based planner $\pi(\mathcal{P})$ (Son, Tu, and Baral, 2004), the knowledge-level planner PKS (Petrick and Bacchus, 2002), and Contingent FF (CFF) (Hoffmann and Brafman, 2005).

All of these planners are able to represent conditional effects of physical actions, therefore, the representation of the physical effects of $prim_\delta$ is straightforward. Unfortunately, the representation of the knowledge effects of $prim_\delta$ is not trivial in some cases. Indeed, without loss of generality, suppose that C contains only one program $\delta(\vec{y})$. After theory compilation, the SSA for the K fluent in $\mathsf{Comp}[\mathcal{D},C]$ has the general form:

$$K(s',do(a,s)) \equiv (\exists s'').s' = do(a,s'') \wedge K(s'',s) \wedge Poss(a,s'') \wedge \varphi(s) \wedge \bigwedge_j \left\{ (\forall \vec{y}).a = prim_\delta(\vec{y}) \wedge \alpha_j(\vec{y},s) \supset \bigwedge_i \beta_{ij}(\vec{y},s) \equiv \beta_{ij}(\vec{y},s'') \right\}, \quad (6.55)$$

where $\varphi(s)$ describes the knowledge effect for the original actions in $\mathcal{D}$, and therefore does not mention the action term $prim_\delta$. Intuitively, as before, β_{ij} are the (regressed) properties that are sensed and α_j are the (regressed) conditions of if-then-else constructs that had to be true for the program to sense β_{ij}.

From the syntax of K, we determine the following requirements for achieving planning with programs that sense in practical planners.

1. The planner must be able to represent *conditional* sensing actions. These are the α_j formulae appearing in (6.55).

2. The planner must be able to represent that $prim_\delta$ senses the truth value of, in general, arbitrary formulae. This is because β_{ij} in (6.55) could be any first-order formula.

Most of the planners do not satisfy these requirements directly. However, in most cases one can modify the planning domain, and still plan with our compiled actions. Below we show how this can be done.

6.5.1 Belief-State-Based Planners

All the planners we investigated, except PKS, are in this category. They represent explicitly or implicitly all the states in which the agent could be during the execution of the plan (sometimes called *belief states*). They are propositional and cannot represent functions[12].

Among the planners investigated, SGP is the only one that cannot be adapted to achieve requirement 1. The reason is that sensing actions in SGP cannot have preconditions or conditional effects. Others ($\pi(\mathcal{P})$, MBP) can be adapted to simulate conditional sensing actions by splitting $prim_\delta$ into several actions with different preconditions.

Regarding requirement 2, SGP and MBP can handle arbitrary (propositional) observation formulae. However, all the remaining planners are only able to sense propositions (GPT, $\pi(\mathcal{P})$, PKS, and CFF). In

[12] GPT can indeed represent functions, but with limited, integer range.

GPT, or in any other propositional planner able to handle actions that have both physical and knowledge effects, this limitation can be overcome by adding two extra fluents for each $prim_\delta$ action. For each formula β_{ij}, add the the fluents F_{ij} and G_{ij} to the compiled theory. Fluent $F_{ij}(\vec{y},s)$ is such that its truth value is equivalent to that of formula $\beta_{ij}(\vec{y},s)$. The SSA for F_{ij} can be obtained by the following expression (Lin and Reiter, 1994):

$$F_{ij}(\vec{y},do(a,s)) \equiv \mathcal{R}[\beta_{ij}(\vec{y},do(a,s)),s]. \tag{6.56}$$

Furthermore, we add $F_{ij}(\vec{y},S_0)$ iff $\beta_{ij}(\vec{y},S_0)$, for all possible constant vectors $\vec{y}$ of the appropriate size.

To define the knowledge effects of $prim_\delta$, we require a little additional effort. For many planners, the semantics for actions that modify the world and sense at the same time is that the sensing (observation) is done *after* the world-level effects have been applied to the world. In contrast, our Situation Calculus formulation of $prim_\delta$ implicitly assumes that the sensing is done before the effects are applied in the world. We need therefore, to modify our theory in order to simulate a sensing effect on the immediate past. To that end, we define the fluent $G_{ij}(\vec{y},do(a,s))$ is such that its truth value is equivalent to that of $\beta_{ij}(\vec{y},s)$ (i.e., it "remembers" the truth value that β_{ij} had in the previous situation). The SSA for G_{ij} is simply $G_{ij}(\vec{y},do(a,s)) \equiv F_{ij}(\vec{y},s)$.

To model $prim_\delta$ in these planners we can obtain their *world-level* effects by looking into the SSA of every fluent (Pednault, 1989). On the other hand, the *knowledge-level* effect is simply that $prim_\delta(\vec{y})$ senses the truth value of fluent $G_{ij}(\vec{y},do(a,s))$, for all i, conditioned on whether $\alpha_j(\vec{y},s)$ is true. The correctness of this approach is justified by the following result.

Proposition 6.5 *Let* $\mathsf{Comp}[\mathcal{D},C]$ *be a theory of action that contains axiom* (6.55)*, and fluents* F_{ij} *and* G_{ij}*. Then* $\mathsf{Comp}[\mathcal{D},C]$ *entails that* (6.55) *is equivalent to*

$$K(s',do(a,s)) \equiv (\exists s'').s' = do(a,s'') \wedge K(s'',s) \wedge \varphi(s,s'') \wedge Poss(a,s'') \wedge \\ \bigwedge_j \{a = prim_\delta(\vec{y}) \wedge \alpha_j(\vec{y},s) \supset \bigwedge_i G_{ij}(\vec{y},do(a,s)) \equiv G_{ij}(\vec{y},do(a,s''))\}.$$

Proof: Follows from the correctness of regression. ■

The immediate consequence of this result is that

$$\mathsf{Comp}[\mathcal{D},C] \models \alpha_j(\vec{y},s) \supset \bigwedge_i \mathbf{KWhether}(G_{ij}(\vec{y}),do(prim_\delta,s)),$$

which intuitively expresses that $prim_\delta(\vec{y})$ is observing the truth value of $G_{ij}(\vec{y})$.

As we mentioned, the previous construction works with planners like GPT, where actions can have both *world effects* and *observations*. However, this still doesn't solve the problem completely for the planners like $\pi(\mathcal{P})$ and CFF, since (currently) they do not support actions with both world-level and knowledge-level effects. Nonetheless, this can be addressed by splitting $prim_\delta$ into two actions, say

$Phys_\delta$ and Obs_δ. Action $Phys_\delta$ would have all the world-level effects of $prim_\delta$ and action Obs_δ would be a sensing action that observes F_{ij}. In this case, we also need to add special preconditions for action $Phys_\delta$, since we would need it to be performed always and only *immediately after* Obs_δ. Such an axiomatization is described by Baier and McIlraith (2005).

6.5.2 Extending PKS

Currently, PKS (Petrick and Bacchus, 2002) is one of the very few planners in the literature that does not represent belief states explicitly.[13] Moreover, it can represent domains using first-order logic and functions. Nevertheless, it does not allow the representation of knowledge about arbitrary formulae. In particular it cannot represent disjunctive knowledge.

PKS, does not directly support requirement 2 either. Moreover, its reasoning algorithm is not able to obtain reasonable results when adding the fluents F_{ij} and G_{ij}, due to its incompleteness.

PKS deals with knowledge of an agent using four databases. Among them, database K_w stores formulae whose truth values are known by the agent. In practice, this means that if an action senses property p, then p is added to K_w after performing it. While constructing a conditional plan, the K_w database is used to determine the properties on which it is possible to condition different branches of the plan. PKS's inference algorithm, **IA**, when invoked with ε can return value **T** (resp. **F**) if ε is known to be true (resp. false) by the agent. On the other hand, it returns **W** (resp. **U**) if the truth value of ε is known (resp. unknown) by the agent.

Nevertheless, since K_w can only store first-order conjunctions of literals, this means that in some cases, information regarding sensing actions of the type generated by our translation procedure would be lost. E.g., if $\neg f$ and g are known to the planner and an action that senses $f \vee g \wedge h$ is performed, PKS is not able to infer that it knows the truth value of h. For cases like this, this limitation can be overcome by the extension we propose below.

We propose to allow K_w to contain first-order CNF formulae. In fact, assume that K_w can contain a formula $\Gamma_1(\vec{x}) \wedge \Gamma_2(\vec{x}) \wedge \ldots \wedge \Gamma_k(\vec{x})$, where Γ_i is a first order clause, and free variables $\vec{x}$ are implicitly universally quantified. We now modify PKS's inference algorithm **IA** by replacing rule 7 of the algorithm of Bacchus and Petrick (1998) by the following rule. We assume the procedure is called with argument ε:

7. If there exists $\phi(\vec{x}) = \Gamma_1(\vec{x}) \wedge \ldots \wedge \Gamma_k(\vec{x}) \in K_w$ and a ground instance of ϕ, $\phi(\vec{x}/\vec{a})$ is such that (1) $\vec{a}$ are constants appearing in K_f, (2) There exists an $\alpha_m \in \Gamma_i$ such that $\alpha_m(\vec{x}/\vec{a}) = \varepsilon$, (3) For every Γ_j ($j \neq i$) there exists a $\beta \in \Gamma_j$ such that $\mathbf{IA}(\beta(\vec{x}/\vec{a})) = \mathbf{T}$, and (4) For every $\alpha_\ell \in \Gamma_i$ ($\ell \neq m$), $\mathbf{IA}(\alpha_\ell(\vec{x}/\vec{a})) = \mathbf{F}$. Then, **return(W)**.

[13] Probably the planner by Pistore, Marconi, Bertoli, and Traverso (2005) is the only other exception.

N	CFF	PKS	CFF +seek	PKS +seek
1	0.01	5.19	0.0	0.01
2	0.1	nomem	0.01	0.01
3	5.01	nomem	0.01	0.08
4	nomem	nomem	0.02	0.77
5	nomem	nomem	0.03	5.89

Table 6.1: Instances of the briefcase domain with sensing solved by PKS and CFF. "nomem" means the planner ran out of memory.

Theorem 6.6 *The modified inference algorithm of PKS is sound.*

Proof sketch: The proof is based on the following facts (1) The modification only affects when the algorithm returns a **W** (2) The new rule's conclusions are based on the following valid formulae **KWhether**$(\alpha \wedge \beta, s) \wedge$ **Knows**$(\alpha, s) \supset$ **KWhether**(β, s), **KWhether**$(\alpha \vee \beta, s) \wedge$ **Knows**$(\neg\beta, s) \supset$ **KWhether**(α, s), and **Knows**$(\alpha, s) \vee$ **Knows**$(\beta, s) \supset$ **Knows**$(\alpha \vee \beta, s)$. ■

To actually use action $prim_\delta$ to plan with PKS, we need to divide it into two primitive actions, a world-altering action, say $Phys_\delta$, and a sensing action, say Obs_δ. Action Obs_δ has the effect of adding β_{ij}—in CNF—to the K_w database. On the other hand, $Phys_\delta$ contains all the world effects of $prim_\delta$. Again, through preconditions, we need to ensure that action $Phys_\delta$ is performed only and always immediately after Obs_δ. This transformation is essentially the same that was proposed for belief-state-based planners that cannot handle actions with both physical and knowledge effects, and can be proved correct (Baier and McIlraith, 2005).

This extension to PKS' inference algorithm is not yet implemented but is part of our future work. In the experiments that follow, we did not need to use this extension since the sensed formulae were simple enough.

6.6 Practical Relevance

There were at least two underlying motivations to the work presented in this chapter that speak to its practical relevance.

6.6.1 Web Service Composition

Web services are self-contained programs that are published on the Web. The airline ticket service at www.aircanada.com, or the weather service at www.weather.com are examples of Web services. Web services are compellingly modeled as programs comprising actions that effect change in the world (e.g., booking you a flight, etc.) as well as actions that sense (e.g., telling you flight schedules, the weather

in a particular city, etc.). Interestingly, since Web services are self-contained, they are generally self-sufficient in the formal sense of this term, as described in this chapter. As such, they fall into the class of programs that can be modeled as planning operators. This is just what is needed for WSC.

WSC is the task of composing existing Web services to realize some user objective. Planning your trip to the KR2006 conference over the Web is a great example of a WSC task. WSC is often conceived as a planning or restricted program synthesis task (McIlraith and Son, 2002). Viewed as a planning task, one must plan with programs that sense to achieve WSC. While there has been significant work on WSC, few have addressed the issue of distinguishing between world-altering and sensing actions, fewer still have addressed the problem of how to represent and plan effectively with programs rather than primitive (one step) services. This work presents an important contribution towards addressing the WSC task.

6.6.2 Experiments

Beyond WSC, the second more general motivation for this work was to understand how to plan with macro-actions or programs, using operator-based planners. The advantages of using operator-based planners are many, including availability of planners and the ability to use fast heuristic search techniques. In general, the search space of plans of length k is exponential in k. When using macro-actions usually we can find shorter plans (composed by such macro-actions), therefore, the planner will effectively explore an exponentially smaller search space. When planning with sensing actions, plans are normally contingent, i.e. they have branches to handle different situations. The search space, therefore, is much bigger and any reduction in the length of the plan may exponentially reduce the time needed for planning.

To illustrate the computational advantages of planning with programs that sense, we performed experiments with a version of the *briefcase* domain (Pednault, 1988), enhanced with sensing actions. In this domain, there are K rooms. The agent carries a briefcase for transporting objects. In any room r, the agent can perform an action $look(o)$ to determine whether o is in r. In the initial state, the agent is in room LR. There are N objects in rooms other than LR. The agent does not know the exact location of any of the objects. The goal is to be in LR with all the objects.

We performed experiments with PKS and CFF for $K = 4$ and $N = 1, \ldots, 5$. Each planner was required to find a plan with and without the use of macro-action $seek(o)$ (Figure 6.1). $seek(o)$ was compiled into a primitive action by our technique. We compared the running time of the planners using a 2 GHz linux machine with 512MB of main memory. PKS was run in iterative deepening mode. Table 6.1 shows running times for both planners with and without the *seek* action. These experiments illustrate the applicability of our approach in a domain that is challenging for state-of-the-art planners when only simple primitive actions are considered.

$$
\begin{aligned}
seek(o) = {} & go(R_1); look(o); \textbf{if } at(o,R_1) \textbf{ then } grasp(o); go(LR) \\
& \textbf{else } go(R_2); look(o); \textbf{if } at(o,R_2) \textbf{ then } grasp(o); go(LR) \\
& \textbf{else } go(R_3); look(o); \textbf{if } at(o,R_3) \textbf{ then } grasp(o); go(LR) \\
& \textbf{else } go(R_4); look(o); \textbf{if } at(o,R_4) \textbf{ then } grasp(o); go(LR) \\
& \textbf{endIf endIf endIf endIf}
\end{aligned}
$$

Figure 6.1: Program *seek* is a tree program that makes the agent move through all the rooms looking for *o* and then bringing it to *LR*

6.7 Summary and discussion

In this chapter we addressed the problem of enabling operator-based planners to plan with *programs*. A particular challenge of this work was to ensure that the proposed method worked for programs that include sensing, though all the contributions are applicable to programs without sensing. We studied the problem in the situation calculus, using Golog to represent our programs. We did this to facilitate formal analysis of properties of our work. Nevertheless, because of the well-understood relationship between ADL and the situation calculus (Pednault, 1989), the results apply very broadly to the class of planning operators represented in the popular plan domain description language PDDL (McDermott, 1998).

Our contributions include a compilation algorithm for transforming programs into operators that are guaranteed to preserve program behaviour for the class of self-sufficient deterministic Golog tree programs. Intuitively, these are programs whose execution is guaranteed to be finite and whose outcome is determinate upon execution. We then showed how to plan with these new operators using existing operator-based planners that sense. In the case of PKS, we proposed a modification to the code to enable its use in the general case. For those interested in Golog, a side effect of this work was to define an offline transition semantics for executable Golog programs.

There were two underlying motivations to this work that speak to its practical relevance. The first was to address the problem of WSC. The class of programs that we can encapsulate as operators corresponds to most, if not all, Web services. As such, this work provides an important contribution to addressing WSC as a planning task. Our second motivation was the use of programs to represent macro-actions and how to use them effectively in operator-based planners. Again, our compilation algorithm provides a means of representing macro-actions as planning operators. Our experimental results, though in no way rigorous, illustrate the effectiveness of our technique.

Chapter 7

Conclusions, Related Work, and Future Work

7.1 Conclusions

In this thesis, we have investigated how recent advances in classical planning can be leveraged to effectively solve some non-classical planning tasks. The planning tasks we have focused on may include rich temporally extended goals and preferences. As well, they may need plans whose building blocks are programs rather than primitive actions, or they may be required to conform to some pre-specified procedural skeleton. Planning problems containing those characteristics appear in a wide variety of compelling applications, including component software composition, web service composition, and agent programming.

To solve these non-classical planning tasks we employ a common approach: *reformulation*. Our reformulation algorithms will take a non-classical planning task problem, and generate a new task. This new task is more amenable to be solved by current state-of-the-art techniques. In some cases, the reformulation results in a *classical* planning task. In other cases, we generate another non-classical planning task that necessitates adaptation of existing planning techniques.

Our approach has a number of advantages. The foremost is that for most of the problems we deal with we produce a standard output, which can be directly input to a wide variety of planners. Another advantage is that the approach is composable. Thus, for example, if one wants to plan with temporally extended preferences using Golog DCK, we could first reformulate the problem using the techniques in Chapter 5—obtaining an instance with no procedural control—and then apply the techniques in Chapter 4 to obtain a "regular" planning instance with only simple preferences. On the other hand, our reformulation approach can give insights to researchers adapting classical techniques for the

non-classical problems we deal with. An example of this can be seen in Chapter 5, where we design the *H-ops* heuristic that specifically addresses particular problems of the FF heuristic in the reformulated instances.

For each of the non-classical planning tasks that we examined, we showed that we can obtain improvement, often very significant, over existing approaches in terms of the efficiency of plan generation. We conclude, therefore, that the reformulation techniques we presented are powerful tools, that usually enable the application of state-of-the-art classical techniques for non-classical planning tasks, allowing us to solve them more effectively.

The rest of this chapter contains a recapitulation of the problems we have addressed and our contributions. Finally, we sketch potential future work.

7.1.1 Problems and Contributions

We have examined four significant types of non-classical planning tasks in this thesis. Our contribution and analysis is both theoretical and experimental. In what follows we summarize some of the most significant contributions.

Planning with Temporally Extended Goals In these problems, goals express conditions that hold throughout the execution of the plan and are therefore more expressive than properties that only refer to the final state. We proposed a method for planning with temporally extended goals using heuristic search, one of the current most effective approaches in classical planning. To this end, we reformulate planning task with TEGs into an equivalent *classical* planning task. With this translation in hand, we exploit heuristic search to determine a plan. Our translation is based on the construction of a parameterized nondeterministic finite automaton that provably accepts the models of the TEG. These automata have the advantage that can be represented in a compact way in a planning domain. In our experiments, we showed that our approach consistently outperforms existing techniques for planning with TEGs that were only based in formula progression combined with blind search.

Planning with Temporally Extended Preferences Here the task is to find a plan that optimizes a quality function that is dependent on those preferences. Our technique involves reformulating a planning problem with TEPs into an equivalent planning problem containing only simple preferences. Since the resulting task is not classical, we provide a collection of new heuristics and a specialized search algorithm that can guide the planner towards preferred plans. We prove that under some fairly general conditions our method is able to find a most preferred plan—i.e., an optimal plan. We have implemented our approach in a planning system we called HPlan-P, and used it to compete in the 5th International Planning Competition, where it achieved *distinguished performance* in the *Qualitative Preferences* track.

Planning and Reasoning with Procedural DCK We have shown that Golog is amenable to representing DCK in planning by defining a PDDL semantics for Golog programs.

Additionally, we show that any planner that can input planning tasks in PDDL is able to plan with our Golog DCK. We do this by giving an algorithm that reformulates any PDDL planning task and a control program, into an equivalent, program-free PDDL task whose plans are only those that "behave" according to the control program.

Finally, we show that the resulting planning task is amenable to use with domain-independent heuristic planners. In particular, we propose three approaches. Our experiments on familiar planning benchmarks show that the combination of DCK and heuristics produce better performance than using DCK with blind search and than using heuristics alone.

Planning and Reasoning with Programs that Sense In this problem, the building blocks for plans are programs instead of or in addition to primitive actions. We propose and prove the correctness of a compilation method that transforms our action theory with programs into a new theory where programs are replaced by primitive actions. This enables us to use state-of-the-art, operator-based planning techniques to plan with programs that sense for a restricted but compelling class of programs. Finally, we discuss the applicability of these results to existing operator-based planners that support sensing and illustrate the computational advantage of planning with programs that sense via an experiment. This work has broad applicability to planning with programs or macro-actions with or without sensing. In our experiments, we have shown that planning with the compiled instances can result in orders of magnitude of improved performance.

7.2 Other Related Work

Each of the technical chapters of the thesis describes work that is closely related to the topic that is exposed therein. However, there are several pieces of work that have also applied reformulation to planning. We group them in two sets and we describe some of them below.

In the first set, we consider work that reformulates classical planning instances into classical instances in a different representation language. Gazen and Knoblock (1997) provide an algorithm to transform ADL planning instances into STRIPS planning instances. This allows to use techniques developed for STRIPS for the more expressive, but still classical, ADL formalism. Their translation is worst-case exponential. Edelkamp and Helmert (1999) proposed an algorithm to convert STRIPS classical tasks into the SAS+ representation (Bäckström and Nebel, 1995). The algorithm was extended and improved by Helmert (2009). Viewing planning problems in the SAS+ may be very useful as this representation is compact and makes the structure of the problem more apparent. Indeed, the structure underlying the SAS+ representation has been exploited by successful enhancements to heuristic-search-based

planners. An example is the causal graph heuristic (Helmert, 2006a), improved landmark extraction (Richter *et al.*, 2008), and analysis of planning complexity (Giménez and Jonsson, 2007).

In the second set we include work that is more related to our work, and that has addressed the problem of reformulating non-classical planning instances into classical or deterministic planning. Palacios and Geffner (2006) reformulate conformant planning problems into classical planning problems. As with our reformulations, this allows them to exploit classical planning technology and thus greatly improve over previous approaches. The techniques we present in this thesis seem to be compatible with their translation. We conjecture that many conformant instances with TEGs could be reduced to classical planning by composing our techniques and Palacios and Geffner's (2006). Also Yoon, Fern, and Givan (2007) and Yoon *et al.* (2008) reformulate probabilistic planning problems into deterministic problems. In doing so, like us, they greatly benefit from deterministic planning technology. Again, it is fairly conceivable to think that our techniques could be used along with theirs in cases where there are TEGs or TEPs.

The focus of the aforementioned related work is to reformulate the entire problem into one described in significantly different language. In most of our work however (Chapters 3, 4, and 5) we reformulate the planning objective, leaving most of the structure of the problem untouched. In particular, we do not alter the transition model as we input a deterministic instance and output a deterministic one. This means that our techniques can also be extended, with little effort, to other transition models (e.g., non-deterministic settings). In those cases our input and output instances would be non-deterministic.

7.3 Future Work

The work presented in this thesis suggest many avenues for future work. In what follows we list a subset of these directions.

Improved Heuristics for TEGs We have shown that off-the-shelf heuristic approaches can be very effective for planning with TEGs. However, as we illustrated in Section 4.6, there are cases in which relaxed plan heuristics are pretty uninformative. In particular, while planning with TEGs that are safety goals (e.g., of the form $\Box\phi$), relaxed plan heuristics are not informative at all. This happens because the predicate that represents the acceptance of $\Box\phi$ is true at any legal state, and thus true in always in any successor of such as state in the delete relaxation. A potential avenue of research is to investigate how recent techniques that more closely approximate the planning problem (e.g. Baier, 2007; Benton *et al.*, 2007; Coles, Fox, Long, and Smith, 2008) may also be successful in producing better heuristics for TEGs.

Another avenue is the investigation of how other heuristics, such as the causal graph heuristic (Helmert, 2006a), can be exploited to provide better guidance for TEGs.

Landmarks as TEGs An effective technique to enhance the performance of classical planners is the use of *landmarks* (Hoffmann, Porteous, and Sebastia, 2004; Richter *et al.*, 2008). Landmarks essentially specify a sequence of sub-goals that need to be achieved before reaching the goals defined in the planning task. As such, they can be specified as TEGs. However, current techniques for planning with landmarks, do not recognize these as being TEGs. For example, the LAMA planner (Richter *et al.*, 2008) uses a pseudo-heuristic that is computed from its landmarks. This pseudo-heuristic seems quite *ad hoc*, and seems to have some problems as it may not recognize certain dead ends. We hypothesize that viewing landmarks as TEGs and exploiting techniques such as ours may provide a more fundamental view to planning in the presence of landmarks.

New Heuristics for Planning with Preferences The branch-and-bound algorithm that we have defined for planning with preferences finds a sequence of plans of increasing quality. After a plan is found, the only piece of information we use in the next planning round is the metric of the last plan found. We use this metric to prune by bounding. However, it is there is more information that we could use after finding a plan. In particular, after finding a plan, we know that there are certain preferences that can, *for sure*, be achieved from certain states. This suggests that there should be a way of producing new heuristics (or modify existing ones) in order to account not only for heuristic information but for *certain* information.

TEGs and TEPs under Other Types of DCK We mentioned in the previous section that TEGs and TEPs could be integrated with Golog DCK by composing our reformulation algorithms. However, other types of DCK, like for example HTNs (Erol *et al.*, 1994), could also benefit from the techniques we have proposed. Sohrabi *et al.* (2009) have very recently made a contribution to this problem by extending the HTN formalism to support PDDL3 preferences, and proposing heuristics based on our reformulated instances.

More General Golog DCK Control in State-of-the-Art Planners In Chapter 5 we considered a subset of Golog for the specification of procedural control. Our subset does not consider procedures—which are standard in Golog—, and does not consider concurrency—which is standard in the ConGolog language (De Giacomo *et al.*, 2000). Fritz *et al.* (2008) have shown that, under certain conditions, it is possible to translate ConGolog DCK intro PDDL. It remains an open question, however, whether or not this translation can be exploited well by state-of-the-art planners directly. From our experience with Golog, we conjecture that this might not be the case, and thus new modifications to our *H-ops* approach might provide better guidance in the presence of concurrency and procedures.

Glossary of Acronyms and Abbreviations

Notation	Description	
ADL	Action Description Language	12
BA	Büchi automata	47
BFQ	Boolean Formula with Quantifiers	93
DCK	Domain Control Knowledge	9, **90**
EPNF	Extended Prenex Normal Form	26
f-FOLTL	Finite First-Order Linear Temporal Logic	21, 22
FSM	Finite state machine	49
Golog	alGOl in LOGic. A high-level action-centric language for programming agents (Levesque *et al.*, 1997)	5, **121**
HTN	Hierarchical task network	89, 92, **110**
IPC	International Planning Competition. See `http://ipc.icaps-conference.org/`.	2
LTL	Linear Temporal Logic	20, 22, 23
PDDL	Planning Domain Definition Language (McDermott, 1998)	9, **13**, 93
PDDL3	Version of PDDL that supports preferences and hard constraints	57
PNFA	Parameterized NFA	29, **35**
PSLNFA	Parameterized State-Labeled NFA	29, **29**

Notation	Description	
SSA	Successor state axiom	115
TEG	Temporally extended goal.	5, 7, **20**
TEP	Temporally extended preference.	5, **52**
WSC	Web service composition.	2, 113

Bibliography

Bacchus, F. and Ady, M. (1999). Precondition Control. URL `http://www.cs.toronto.edu/~fbacchus/Papers/BApre.pdf`. Unpublished manuscript.

Bacchus, F. and Kabanza, F. (1998). Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence* , *22*(1-2), 5–27.

Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* , *116*(1-2), 123–191.

Bacchus, F. and Petrick, R. (1998). Modeling an agent's incomplete knowledge during planning and execution. In *Proceedings of the 6th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 432–443. Morgan Kaufmann Publishers, San Francisco, CA.

Bäckström, C. and Nebel, B. (1995). Complexity Results for SAS+ Planning. *Computational Intelligence* , *11*(4), 625–655.

Baier, J. and McIlraith, S. (2005). Planning with Programs that Sense. In *5th Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)*, pp. 7–14. Edinburgh, Scotland.

Baier, J. A. (2007). Improving Relaxed-Plan-Based Heuristics. In *First ICAPS Workshop on Heuristics for Domain-independent Planning: Progress, Ideas, Limitations, Challenges*. Providence, RI.

Baier, J. A., Bacchus, F., and McIlraith, S. A. (2009). A Heuristic Search Approach to Planning with Temporally Extended Preferences. *Artificial Intelligence* , *173*(5-6), 593–618.

Baier, J. A., Fritz, C., and McIlraith, S. A. (2007). Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 26–33. Providence, Rhode Island, USA.

Baier, J. A. and McIlraith, S. A. (2006a). On Planning with Programs that Sense. In *Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 492–502. Lake District, UK.

Baier, J. A. and McIlraith, S. A. (2006b). Planning with First-Order Temporally Extended Goals Using Heuristic Search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pp. 788–795. Boston, MA.

Baier, J. A. and McIlraith, S. A. (2006c). Planning with Temporally Extended Goals Using Heuristic Search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 342–345.

Baier, J. A. and McIlraith, S. A. (2007). On Domain-Independent Heuristics for Planning with Qualitative Preferences. In *7th Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)*.

Baier, J. A. and McIlraith, S. A. (2008). Planning with Preferences. *Artificial Intelligence Magazine*, *29*(4), 25–36.

Baral, C., Kreinovich, V., and Trejo, R. (2000). Computational complexity of planning and approximate planning in the presence of incompleteness. *Artificial Intelligence* , *122*(1-2), 241–267.

Benton, J., Kambhampati, S., and Do, M. B. (2006). YochanPS: PDDL3 Simple Preferences and Partial Satisfaction Planning. In *5th International Planning Competition Booklet (IPC-2006)*, pp. 54–57. Lake District, England.

Benton, J., van den Briel, M., and Kambhampati, S. (2007). A Hybrid Linear Programming and Relaxed Plan Heuristic for Partial Satisfaction Problems. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 34–41. Providence, RI.

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001). Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 473–478. Seattle, WA, USA.

Bienvenu, M., Fritz, C., and McIlraith, S. (2006). Planning with Qualitative Temporal Preferences. In *Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 134–144. Lake District, England.

Blum, A. and Furst, M. L. (1997). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* , *90*(1-2), 281–300.

Bonet, B. and Geffner, H. (2000). Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Systems (AIPS)*, pp. 52–61.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence* , *129*(1-2), 5–33.

Bonet, B. and Geffner, H. (2006). Heuristics for Planning with Penalties and Rewards using Compiled Knowledge. In *Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 452–462.

Brafman, R. and Chernyavsky, Y. (2005). Planning with Goal Preferences and Constraints. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 182–191. Monterey, CA.

Bylander, T. (1994). The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* , *69*(1-2), 165–204.

Cerrito, S., Mayer, M. C., and Praud, S. (1999). First Order Linear Temporal Logic over Finite Time Structures. In *Proceedings of 6th International Conference on Logic Programming and Automated Reasoning (LPAR), LNCS*, volume 1705, pp. 62–76. Tbilisi, Georgia.

Claßen, J., Eyerich, P., Lakemeyer, G., and Nebel, B. (2007). Towards an Integration of Golog and Planning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1846–1851.

Coles, A., Fox, M., Long, D., and Smith, A. (2008). A Hybrid Relaxed Planning Graph-LP Heuristic for Numeric Planning Domains. In *Proceedings of the 18th International Conference on Automated Planning and Sched. (ICAPS)*, pp. 52–59.

Coles, A. I. and Smith, A. J. (2007). Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research*, *28*, 119–156.

Cresswell, S. and Coddington, A. M. (2004). Compilation of LTL Goal Formulas into PDDL. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, (edited by R. L. de Mántaras and L. Saitta), pp. 985–986. IOS Press, Valencia, Spain.

dal Lago, U., Pistore, M., and Traverso, P. (2002). Planning with a Language for Extended Goals. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pp. 447–454. Edmonton, Alberta, Canada.

Daniele, M., Giunchiglia, F., and Vardi, M. Y. (1999). Improved Automata Generation for Linear Temporal Logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV), LNCS*, volume 1633, pp. 249–260. Springer, Trento, Italy.

Davis, E. (1994). Knowledge Preconditions for Plans. *Journal of Logic and Computation*, *4*(5), 721–766.

De Giacomo, G., Lespérance, Y., and Levesque, H. (2000). ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence* , *121*(1-2), 109–169.

De Giacomo, G. and Levesque, H. (1999). An Incremental Interpreter for High-Level Programs with Sensing. In *Logical foundation for cognitive agents: contributions in honor of Ray Reiter*, (edited by H. Levesque and F. Pirri), pp. 86–102. Springer Verlag, Berlin.

Delgrande, J. P., Schaub, T., and Tompits, H. (2007). A General Framework for Expressing Preferences in Causal Reasoning and Planning. *Journal of Logic and Computation*, *17*, 871–907.

Dimopolus, Y., Gerevini, A., Haslum, P., and Saetti, A. (2006). The Benchmark Domains of the Detrministic Part of IPC-5. In *5th International Planning Competition Booklet (IPC-2006)*. `http://zeus.ing.unibs.it/ipc-5/`.

Do, M. B., Benton, J., van den Briel, M., and Kambhampati, S. (2007). Planning with Goal Utility Dependencies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1872–1878. Hyderabad, India.

Edelkamp, S. (2006a). On the Compilation of Plan Constraints and Preferences. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*. To appear.

Edelkamp, S. (2006b). Optimal Symbolic PDDL3 Planning with MIPS-BDD. In *5th International Planning Competition Booklet (IPC-2006)*, pp. 31–33. Lake District, England.

Edelkamp, S. and Helmert, M. (1999). Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length. In *Proceedings of the 5th European Conference on Planning (ECP)*, pp. 135–147.

Edelkamp, S. and Hoffmann, J. (2004). PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Computer Science Department, University of Freiburg.

Edelkamp, S., Jabbar, S., and Naizih, M. (2006). Large-Scale Optimal PDDL3 Planning with MIPS-XXL. In *5th International Planning Competition Booklet (IPC-2006)*, pp. 28–30. Lake District, England.

Erol, K., Hendler, J., and Nau, D. (1994). HTN Planning: Complexity and Expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, volume 2, pp. 1123–1128.

Etessami, K. and Holzmann, G. J. (2000). Optimizing Büchi Automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, *LNCS*, volume 1877, pp. 153–167. Springer, University Park, PA.

Feldmann, R., Brewka, G., and Wenzel, S. (2006). Planning with Prioritized Goals. In *Proceedings of the 10th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 503–514. Lake District, England.

Ferrein, A., Fritz, C., and Lakemeyer, G. (2005). Using Golog for Deliberation and Team Coordination in Robotic Soccer. *Künstliche Intelligenz, 19*(1), 24–31.

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

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence* , *3*, 251–288.

Fox, M. and Long, D. (2003). PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, *20*, 61–124.

Fritz, C. (2003). Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA), LNCS*, volume 2759, pp. 35–48. Springer, Santa Barbara, CA.

Fritz, C., Baier, J. A., and McIlraith, S. A. (2008). ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proceedings of the 11th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 600–610. Sydney, Australia.

Gastin, P. and Oddoux, D. (2001). Fast LTL to Büchi Automata Translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, (edited by G. Berry, H. Comon, and A. Finkel), *LNCS*, volume 2102, pp. 53–65. Springer, Paris, France.

Gazen, B. C. and Knoblock, C. A. (1997). Combining the Expressivity of UCPOP with the Efficiency of Graphplan. In *ECP97*, pp. 221–233. Toulouse, France.

Gerevini, A., Dimopoulos, Y., Haslum, P., and Saetti, A. (2006). 5th International Planning Competition. `http://zeus.ing.unibs.it/ipc-5/`.

Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, *173*(5-6), 619–668.

Gerevini, A., Saetti, A., and Serina, I. (2003). Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *Journal of Artificial Intelligence Research*, *20*, 239–290.

Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. (1995). Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pp. 3–18. Warsaw, Poland.

Giménez, O. and Jonsson, A. (2007). On the Hardness of Planning Problems with Simple Causal Graphs. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 152–159.

Giunchiglia, E. and Maratea, M. (2007). Planning as Satisfiability with Preferences. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 987–992. Vancouver, British Columbia.

Gupta, N. and Nau, D. S. (1992). On the Complexity of Blocks-World Planning. *Artificial Intelligence* , *56*(2-3), 223–254.

Haslum, P. (2007). Openstacks SP-NCE domain. URL `http://users.rsise.anu.edu.au/~patrik/ipc5.html`.

Helmert, M. (2003). Complexity results for standard benchmark domains in planning. *Artificial Intelligence* , *143*(2), 219–262.

Helmert, M. (2006a). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M. (2006b). New Complexity Results for Classical Planning Benchmarks. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 52–62.

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* , *173*(5-6), 503–535.

Hendler, J. (1999). Is there an intelligent agent in your future? *Nature Web Matters*, *March*. URL `http://www.nature.com/nature/webmatters/agents/agents.html`.

Hoffmann, J. (2003). The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, *20*, 291–341.

Hoffmann, J. and Brafman, R. (2005). Contingent Planning via Heuristic Forward Search with Implicit Belief States. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 71–80. Morgan Kaufmann, Monterey, CA, USA.

Hoffmann, J. and Edelkamp, S. (2005). The Deterministic Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research*, *24*, 519–579.

Hoffmann, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Hoffmann, J., Porteous, J., and Sebastia, L. (2004). Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, *22*, 215–278.

Hsu, C.-W., Wah, B., Huang, R., and Chen, Y. (2007). Constraint Partitioning for Solving Planning Problems with Trajectory Constraints and Goal Preferences. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1924–1929. Hyderabad, India.

Hui, B., Liaskos, S., and Mylopoulos, J. (2003). Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE)*, pp. 117–126.

Jonsson, P. and Bäckström, C. (1998). Tractable Plan Existence Does Not Imply Tractable Plan Generation. *Annals of Mathematics and Artificial Intelligence* , *22*(3-4), 281–296.

Kabanza, F. and Thiébaux, S. (2005). Search Control in Planning for Temporally Extended Goals. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 130–139.

Kim, P., Williams, B. C., and Abramson, M. (2001). Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 487–493.

Korf, R. E. (1987). Planning as Search: A Quantitative Approach. *Artificial Intelligence* , *33*(1), 65–88.

Kvarnström, J. and Doherty, P. (2000). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics Artificial Intelligence* , *30*(1-4), 119–169.

Lespérance, Y., Levesque, H., Lin, F., and Scherl, R. (2000). Ability and Knowing How in the Situation Calculus. *Studia Logica*, *66*(1), 165–186.

Levesque, H. (1996). What is Planning in the Presence of Sensing? In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI)*, pp. 1139–1146. Portland, Oregon.

Levesque, H. and Lakemeyer, G. (2007). *Cognitive robotics*. Handbook of Knowledge Representation. Elsevier.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. (1997). GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, *31*(1-3), 59–83.

Levesque, H. J. (2005). Planning with Loops. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 509–515. Edinburgh, Scotland.

Lin, F. and Reiter, R. (1994). State Constraints Revisited. *Journal of Logic and Computation*, *4*(5), 655–678.

McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, (edited by B. Meltzer and D. Michie), pp. 463–502. Edinburgh University Press.

McDermott, D. V. (1996). A Heuristic Estimator for Means-Ends Analysis in Planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning and Systems (AIPS)*, pp. 142–149.

McDermott, D. V. (1998). PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McIlraith, S. and Son, T. C. (2002). Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR)*, pp. 482–493. Toulouse, France.

McIlraith, S. A. and Fadel, R. (2002). Planning with complex actions. In *9th International Workshop on Non-Monotonic Reasoning (NMR)*, pp. 356–364. Toulouse, France.

McIlraith, S. A., Son, T. C., and Zeng, H. (2001). Semantic Web Services. *IEEE Intelligent Systems*, *16*(2), 46–53.

Moore, R. C. (1985). A formal Theory of Knowledge and Action. In *Formal Theories of the Commonsense World*, (edited by J. B. Hobbs and R. C. Moore), chapter 9, pp. 319–358. Ablex Publishing Corp., Norwood, New Jersey.

Nau, D. S., Cao, Y., Lotem, A., and Muñoz-Avila, H. (1999). SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 968–975.

Nebel, B. (2000). On the Compilability and Expressive Power of Propositional Planning Formalisms. *Journal of Artificial Intelligence Research*, *12*, 271–315.

Palacios, H. and Geffner, H. (2006). Compiling Uncertainty Away: Solving Conformant Planning Problems using a Classical Planner (Sometimes). In *AAAI*.

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

Pednault, E. P. D. (1989). ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of the 1st International Conference of Knowledge Representation and Reasoning (KR)*, pp. 324–332. Toronto, Canada.

Petrick, R. P. A. and Bacchus, F. (2002). A Knowledge-Based Approach to Planning with Incomplete Information and Sensing. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Systems (AIPS)*, pp. 212–222. Toulouse, France.

Pistore, M., Marconi, A., Bertoli, P., and Traverso, P. (2005). Automated Composition of Web Services by Planning at the Knowledge Level. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1252–1259.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57.

Reiter, R. (1991). *The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression*, pp. 359–380. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Academic Press, San Diego, CA.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA.

Richter, S., Helmert, M., and Westphal, M. (2008). Landmarks Revisited. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 975–982. Chicago, IL.

Rintanen, J. (2000). Incorporation of Temporal Logic Control into Plan Operators. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, (edited by W. Horn), pp. 526–530. IOS Press, Berlin, Germany.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence , 5*, 115–135.

Sanchez, R. and Kambhampati, S. (2005). Planning Graph Heuristics for Selecting Objectives in Over-Subscription Planning Problems. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 192–201. Monterey, CA.

Sardina, S., de Giacomo, G., Lespérance, Y., and Levesque, H. (2004). On the Semantics of Deliberation in IndiGolog – From Theory to Implementation. *Annals of Mathematics and Artificial Intelligence , 41*(2-4), 259–299.

Savitch, W. J. (1970). Relationships Between Nondeterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences, 4*(2), 177–192.

Scherl, R. and Levesque, H. (2003). Knowledge, Action, and the Frame Problem. *Artificial Intelligence* , *144*(1–2), 1–39.

Scherl, R. B. and Levesque, H. J. (1993). The Frame Problem and Knowledge-Producing Actions. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI)*, pp. 689–695.

Smith, D. E. (2004). Choosing Objectives in Over-Subscription Planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 393–401. Whistler, Canada.

Sohrabi, S., Baier, J., and McIlraith, S. A. (2009). HTN Planning with Preferences. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*. Pasadena, California. To appear.

Sohrabi, S., Prokoshyna, N., and McIlraith, S. A. (2006). Web Service Composition Via Generic Procedures and Customizing User Preferences. In *Proceedings of the 5th International Semantic Web Conference (ISWC)*, pp. 597–611. Athens, Georgia.

Son, T. C., Baral, C., Nam, T. H., and McIlraith, S. A. (2006). Domain-Dependent Knowledge in Answer Set Planning. *ACM Transactions on Computational Logic*, *7*(4), 613–657.

Son, T. C. and Pontelli, E. (2004). Planning with Preferences using Logic Programming. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, (edited by V. Lifschitz and I. Niemela), number 2923 in LNCS, pp. 247–260. Springer.

Son, T. C. and Pontelli, E. (2006). Planning with preferences using logic programming. *Theory and Practice of Logic Programming*, *6*(5), 559–607.

Son, T. C., Tu, P. H., and Baral, C. (2004). Planning with Sensing Actions and Incomplete Information Using Logic Programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR), Lecture Notes in Computer Science*, volume 2923, pp. 261–274. Springer, Fort Lauderdale, FL, USA.

Srivastava, B. and Koehler, J. (2003). Web Service Composition - Current Solutions and Open Problems. In *In: ICAPS 2003 Workshop on Planning for Web Services*, pp. 28–35.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of PDDL axioms. *Artificial Intelligence* , *168*(1-2), 38–69.

Thrun, S., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., Hähnel, D., Rosenberg, C. R., Roy, N., Schulte, J., and Schulz, D. (1999). MINERVA: A Second-Generation Museum Tour-Guide Robot. In *ICRA*, pp. 1999–2005.

van den Briel, M., Nigenda, R. S., Do, M. B., and Kambhampati, S. (2004). Effective Approaches for Partial Satisfaction (Over-Subscription) Planning. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI)*, pp. 562–569.

Vardi, M. Y. and Wolper, P. (1994). Reasoning about Infinite Computations. *Information and Computation*, *115*(1), 1–37.

Waldinger, R. (1977). Achieving Several Goals Simultaneously. In *Machine Intelligence 8*, pp. 94–136. Ellis Horwood, Edinburgh, Scotland.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending Graphplan to Handle Uncertainty & Sensing Actions. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pp. 897–904.

Yoon, S. W., Fern, A., and Givan, R. (2007). FF-Replan: A Baseline for Probabilistic Planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 352–359.

Yoon, S. W., Fern, A., Givan, R., and Kambhampati, S. (2008). Probabilistic Planning via Determinization in Hindsight. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*.

Zhu, L. and Givan, R. (2005). Simultaneous Heuristic Search for Conjunctive Subgoals. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pp. 1235–1241. Pittsburgh, Pennsylvania, USA.

Appendix A

Proofs for Chapter 3

A.1 Proof for Proposition 3.1

Now we prove identities independently.

1. We prove that $\psi \mathsf{U} \chi \equiv \chi \vee \psi \wedge \bigcirc(\psi \mathsf{U} \chi)$.
   From Def. 3.3, $\langle \sigma_i, \nu \rangle \models \psi \mathsf{U} \chi$ iff there exists a $j \in \{i, \ldots, n\}$ such that $\langle \sigma_j, \nu \rangle \models \chi$ and for every $k \in \{i, \ldots, j-1\}$, $\langle \sigma_k, \nu \rangle \models \psi$.

   In the right-hand side of the expression, we can substitute $j \in \{i, \ldots, n\}$ by "$j = i$ or $j \in \{i+1, \ldots, n\}$", which converts the expression into the disjunction of the following statements.

   (1) there exists a $j = i$ such that $\langle \sigma_j, \nu \rangle \models \chi$,

   (2) there exists a $j \in \{i+1, \ldots, n\}$ such that $\langle \sigma_j, \nu \rangle \models \chi$ and for every $k \in \{i+1, \ldots, j-1\}$, $\langle \sigma_k, \nu \rangle \models \psi$ and $\langle \sigma_i, \nu \rangle \models \psi$.

   (1) is simply equivalent to $\langle \sigma_i, \nu \rangle \models \chi$. On the other hand, (2) reduces to $\langle \sigma_i, \nu \rangle \models \psi \wedge \bigcirc(\psi \mathsf{U} \chi)$. This can be verified straightforwardly in two relevant cases: when $i < n$ or when $i \geq n$.

2. We prove that $\neg\bigcirc\varphi \equiv \mathsf{final} \vee \bigcirc\neg\varphi$. Indeed by Def. 3.3, $\langle \sigma_i, \nu \rangle \models \neg\bigcirc\varphi$ iff $i \geq n$ or $\langle \sigma_{i+1}, \nu \rangle \not\models \varphi$. Without loss of generality we assume $i \leq n$. By Def. 3.3 the statement is true iff $\langle \sigma_i, \nu \rangle \models \mathsf{final}$ or $\langle \sigma_{i+1}, \nu \rangle \models \neg\varphi$, from where the result follows immediately.

3. We prove $\psi \mathsf{U} (\exists x)\, \varphi \equiv (\exists x)\, (\psi \mathsf{U} \varphi)$. From Def. 3.3, the formula is true iff there exists a $j \in \{i, \ldots, n\}$ such that there exists a $a \in \mathcal{D}$ such that $\langle \sigma_j, \nu[x \to a] \rangle \models \varphi$ and for every $k \in \{i, \ldots, j-1\}$, $\langle \sigma_k, \nu \rangle \models \psi$. Because x does not appear free in ψ we can rewrite the expression in the equivalent form: there exists a $a \in \mathcal{D}$ such that there exists a $j \in \{i, \ldots, n\}$ such that $\langle \sigma_j, \nu[x \to a] \rangle \models \varphi$

and for every $k \in \{i, \ldots, j-1\}$, $\langle \sigma_k, \nu[x \to a] \rangle \models \psi$. The result follows straightforwardly from here.

4. $\psi \mathsf{R} (\forall x) \varphi \equiv (\forall x) (\psi \mathsf{R} \varphi)$ can be proved by rewriting the R in terms of an U, then applying the previous identity, and then going back to an expression containing R.

5. We prove $\psi \mathsf{R} \chi \equiv \chi \wedge (\mathsf{final} \vee \psi \vee \bigcirc(\psi \mathsf{R} \chi))$..

$$
\begin{aligned}
\psi \mathsf{R} \chi &\equiv \neg(\neg\psi \mathsf{U} \neg\chi) && \text{(by definition of R)} \\
&\equiv \neg(\neg\chi \vee \neg\psi \wedge \bigcirc(\neg\psi \mathsf{U} \neg\chi)) && \text{(by identity 1)} \\
&\equiv (\chi \wedge (\psi \vee \neg\bigcirc(\neg\psi \mathsf{U} \neg\chi))) \\
&\equiv (\chi \wedge (\psi \vee \mathsf{final} \vee \bigcirc(\neg(\neg\psi \mathsf{U} \neg\chi)))) && \text{(by identity 2)} \\
&\equiv (\chi \wedge (\psi \vee \mathsf{final} \vee \bigcirc(\psi \mathsf{R} \chi))) && \text{(by definition of R)}
\end{aligned}
$$

A.2 Proof for Theorem 3.1

Before starting with the proof, we formally define some concepts and abbreviations that we use in the proofs.

Definition A.1 ($Old^{-}(q)$) *We define $Old^{-}(q)$ as the set containing all the literals in $Old(q)$ or formulae of the form $(Qx)\varphi$, where φ is a first-order (atemporal) formula.*

If X is a set of temporal formulae, we denote by $\bigwedge X$ the conjunction the elements in X. $\bigwedge X$ reduces to $True$ if X is empty.

We define the abbreviation $\beta(q)$ in the following way:

$$
\beta(q) \stackrel{\text{def}}{=} \begin{cases} \bigcirc \bigwedge Next(q) & \text{if } Next(q) \neq \emptyset \\ True & \text{otherwise.} \end{cases}
$$

We are now going to prove some intermediate results that will allow us to prove the main result rather straightforwardly. In most of the intermediate lemmas, we assume that we construct an automaton for a quantifier-free formula, which may contain free variables. Intuitively, this quantifier-free formula is the actual parameter received by the translation algorithm. This, however, does not mean that the result of Theorem 3.1 holds for formulae with free variables. It is just convenient to do it this way to facilitate the proof.

Lemma A.1 (Analogous to Lemma 4.2 by Gerth *et al.* (1995)) *If nodes q_1 and q_2 are split from a node q (in lines 32-35), then the following property holds.*

$$\begin{aligned}(\bigwedge Old^{-}(q) \wedge \bigwedge New(q) \wedge \beta(q)) \equiv \\ (\bigwedge Old^{-}(q_1) \wedge \bigwedge New(q_1) \wedge \beta(q_1)) \vee \\ (\bigwedge Old^{-}(q_2) \wedge \bigwedge New(q_2) \wedge \beta(q_2))\end{aligned}$$

Similarly, when node q is updated to become a new node q' (in lines 20-31), the following holds

$$\begin{aligned}(\bigwedge Old^{-}(q) \wedge \bigwedge New(q) \wedge \beta(q)) \equiv \\ (\bigwedge Old^{-}(q') \wedge \bigwedge New(q') \wedge \beta(q'))\end{aligned}$$

Proof: Follows directly from the properties of f-FOLTL. ■

Following the proof by Gerth *et al.* (1995), we define an ancestor relation between nodes R, such that $(p,q) \in R$ iff $Father(q) = Name(p)$. Let R^* be the transitive closure of R. A node is *rooted* if it has no ancestors; i.e., $Father(q) = Name(q)$.

Lemma A.2 (Analogous to Lemma 4.3 by Gerth *et al.* (1995)) *Let p be a rooted node, and $Q_p = \{q_i \,|\, (p,q_i) \in R^*\}$. Let $\aleph$ be the set of formulas that are in $New(p)$, when it is created. Let $Next(q_i)$ be the values of the field Next for q_i at the end of the construction. Then, the following holds:*

$$\bigwedge \aleph \equiv \bigvee_{q_i \in Q_p} (\bigwedge \Delta^{-}(q_i) \wedge \beta(q_i)).$$

Proof: By induction in the construction using Lemma A.1. ■

Lemma A.3 *Let graph G_φ be generated by the algorithm for formula φ. Let $R = \{r_1, \ldots, r_n\}$ be the successors of node p. Moreover let $\psi = \bigwedge Next(p)$. Then the graph that results by invoking the algorithm with ψ, G_ψ, is isomorphic with the graph that results from removing every state from G_φ that is unreachable by the nodes in R. Furthermore, the isomorphism is such that if it maps q to q', then $\Delta(q) = \Delta(q')$.*

Proof: Since nodes in R are direct successors of p, we know that there exists a common ancestor r' in G of all nodes in R such that, at the beginning of its construction, $New(r')$ is equal to $Next(p)$.

Likewise, when invoking the algorithm with ψ, the starting node, say r'', will contain its *New* field equal to ψ. We start mapping node r' to r''. Each time we split a node into two nodes, we look at G_φ and map the two successors accordingly. We repeat this process recursively.

The resulting mapping is effectively an isomorphism, since graph G_ψ is constructed using the same procedure as the subgraph of G_φ rooted in r'. ■

For the following lemmas, φ is an f-FOLTL formula whose quantifiers do not scope over temporal formulae, and ν is any variable assignment for all free variables in φ.

Lemma A.4 *If there is an accepting run $\rho = q_0 q_1 \cdots q_n$ for σ in $A_\varphi \cdot \nu$, and $Next(q_0) = \psi$, then there is an accepting run for σ_1 in $A_\psi \cdot \nu$.*

Proof: Since q_1 is successor of q_0, the proof results directly from Lemma A.3. ■

Lemma A.5 *If $\rho = q_0 q_1 \cdots q_n$ is an accepting run for $\sigma = s_0 s_1 \cdots s_n$ in $A_\varphi \cdot \nu$, then $\langle \sigma_0, \nu \rangle \models \Delta^-(q_0)$.*

Proof: Since ρ is a run, $\langle s_0, \nu \rangle \models \bigwedge \Delta^-(q_0) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$. Also, since $\Delta^-(q_0)$ contains no temporal formulae, $\langle \sigma_0, \nu \rangle \models \bigwedge \Delta^-(q_0) \setminus \{\mathsf{final}, \neg\mathsf{final}\}$. Now we have two cases.

- $n = 0$. Since q_0 is final, we know $\neg\mathsf{final} \notin \Delta^-(q_0)$ (by definition of final state). Now, whether or not $\mathsf{final} \in \Delta^-(q_0)$, $\langle \sigma_0, \nu \rangle \models \bigwedge \Delta^-(q_0)$.
- $n > 0$. Since q_0 has a successor, then $\mathsf{final} \notin \Delta^-(q_0)$ (see condition in line 6 in the algorithm). Now, whether or not $\neg\mathsf{final} \in \Delta^-(q_0)$, $\langle \sigma_0, \nu \rangle \models \bigwedge \Delta^-(q_0)$.

■

Lemma A.6 *If there is an accepting run for σ in $A_\varphi \cdot \nu$, then $\langle \sigma, \nu \rangle \models \varphi$.*

Proof: By induction in the length of σ.

- Base case ($\sigma = s_0$). Then, there is an initial state q in $A_\varphi \cdot \nu$ which is also final. By Lemma A.5, $\langle \sigma, \nu \rangle \models \bigwedge \Delta^-(q)$. Since q is final, we have that $Next(q) = \emptyset$. From Lemma A.2, we conclude immediately that $\langle \sigma, \nu \rangle \models \varphi$.
- Induction. Suppose $\rho = q_0 q_1 \ldots q_k$ is an accepting run for σ in $A_\varphi \cdot \nu$. Then, by Lemma A.5, $\langle \sigma_0, \nu \rangle \models \Delta^-(q_0)$. Moreover, from Lemma A.4, there is an accepting run for σ_1 in $A_\psi \cdot \nu$, where $\psi = Next(q_0)$.

  By inductive hypothesis, $\langle \sigma_1, \nu \rangle \models Next(q_0)$. Therefore, by f-FOLTL equivalence, we have that $\langle \sigma, \nu \rangle \models \Delta^-(q_0) \wedge \bigcirc Next(q_0)$. Then, by Lemma A.2, we conclude immediately that $\langle \sigma, \nu \rangle \models \varphi$.

■

Lemma A.7 *If $\langle \sigma, \nu \rangle \models \varphi$, then there is an accepting run for σ in $A_\varphi \cdot \nu$.*

Proof: By induction in the length of σ.

- Base case ($|\sigma| = 1$). By Lemma A.2, we have that

$$\langle \sigma, \nu \rangle \models \bigwedge \Delta^-(q) \wedge \beta(q), \tag{A.1}$$

  for some initial state q.

  We can conclude the following.

1. $Next(q) = \emptyset$. This is because if $|\sigma| = 1$, then $\langle \sigma, \nu \rangle \not\models \bigcirc \xi$ for every ξ.
2. The condition above and (A.1) imply $\langle \sigma, \nu \rangle \models \bigwedge \Delta^-(q)$.
3. Moreover, since $\langle \sigma, \nu \rangle \not\models \neg\mathsf{final}$, we know $\neg\mathsf{final} \notin \Delta^-(q)$.

From 1. and 3. we conclude that q is final. From 2., we conclude that $\langle s_0, \nu \rangle \models \bigwedge \Delta^-(q)$. Hence σ has an accepting run in $A_\varphi \cdot \nu$.

- Induction. By Lemma A.2, we conclude that $\sigma \models \bigwedge \Delta^-(q) \wedge \beta(q)$ for some initial state q. We have two cases.

  – $Next(q) = \emptyset$. In this case $\sigma_0 \models \bigwedge \Delta^-(q)$. As before, this means that $s_0 \models \bigwedge \Delta^-(q)$, and therefore q can be the initial state of a run.

  Furthermore, since $|\sigma| > 0$, $\sigma \not\models \mathsf{final}$. Hence, there must be a transition from q to a state q', which by the algorithm construction is such that $\Delta^-(q') = \emptyset$, is final, and has a transition to itself. This means that σ has an accepting run in A_φ, in fact such run is $q(q')^n$.

  – $Next(q) \neq \emptyset$. Again, we have $s_0 \models \bigwedge \Delta^-(q)$. Since, as before, $\sigma \not\models \mathsf{final}$, we have a transition from q to q'. State q' was initially invoked with $New(q') = Next(q')$, so, by Lemma A.3 and the induction hypothesis, we have that any run for σ_1 from q' has an accepting run in A_ψ, with $\psi = \bigwedge Next(q')$. Since any path in A_ψ has an isomorphic path in A_φ, then σ has a run in A_φ.

■

We are now ready to prove the main result.

Proof (for Theorem 3.1) : Let φ by such that $\psi = Q_1 x_1 Q_2 x_2 \cdot Q_n x_n \varphi$ and such that no quantifiers in φ scope over temporal formulae. Now, by Lemmas A.7 and A.5 we observe that if ν is an assignment of the free variables in φ.

$$A_\varphi \cdot \nu \text{ accepts } \sigma \text{ iff } \langle \sigma, \nu \rangle \models \varphi \tag{A.2}$$

Now, let $\Im$ be such that $\sigma \models \psi$ iff $\langle \sigma, \nu \rangle \models \varphi$, for all $\nu \in \Im$. Note that by definition A_ψ accepts σ iff $A_\varphi \cdot \nu$ accepts σ, for all $\nu \in \Im$. By semantics of f-FOLTL and A.2, this last assertion can only hold true iff $\sigma \models \psi$. This concludes the proof. ■

A.3 Proof for Proposition 3.4

Let $P = \{p_1, p_2, \ldots, p_n\}$, and let $\sigma = s_1, s_2, \ldots, s_{2^n}$ be an arbitrary sequence of all subsets of P. Formally, each s_i contains an element in 2^P and no pair of states $s_i, s_j \in \sigma$ are equal if $i \neq j$. Moreover, for each

state s_i, we denote by s_{i^*} the state in σ such that $s_{i^*} = P \setminus s_i$. Finally, let $A = \langle Q, \Sigma(S,\mathcal{D}), \delta, \epsilon, \epsilon, Q_0, F \rangle$ be the PNFA for $\varphi \stackrel{\text{def}}{=} \Diamond p_1 \wedge \Diamond p_2 \wedge \ldots \wedge \Diamond p_n$.

Note that because A accepts the models of φ, processing $s_i s_{i^*}$ leads the automaton to an accepting state, for any i.

Claim: Let $\rho_i = q_o q q_f$ and $\rho_j = q_0 q' q'_f$ be (accepting) runs of A on, respectively, $s_i s_{i^*}$ and $s_j s_{j^*}$, and $s_i \neq s_j$, then $q \neq q'$. **Proof:** Let us assume the contrary, i.e., that there exist two different states in σ, s_i and s_j, such that there are (accepting) runs $\rho_i = q_o q q_f$ and $\rho_j = q_0 q q'_f$ in A for $s_i s_{i^*}$ and $s_j s_{j^*}$, respectively.

Because $s_i \neq s_j$, we divide the rest of the proof into two cases:

Case 1 ($s_{i^*} \subsetneq s_{j^*}$) In this case, $q_0 q q'_f$ is an accepting run for $s_j s_{i^*}$. This implies a contradiction because if A is a PNFA for φ, then $s_j s_{i^*}$ is not be accepted because $s_j \cup s_{i^*}$ does not include all propositions in P.

Case 2 ($s_{j^*} \subsetneq s_{i^*}$) In this case, $q_0 q q_f$ is an accepting run for $s_i s_{j^*}$. This implies a contradiction, analogous to the previous case.

Notice that the claim immediately implies that there are at least 2^n states in A, since 2^n different states are visited when accepting $s_i s_{i^*}$, for each $s_i \in \sigma$. ■

Appendix B

Proofs for Chapter 4

B.1 Proof for Proposition 4.1

In this section we prove Proposition 4.1. First, we prove three intermediate results that will be used by the final proof.

The first intermediate result says that if an NNF formula ϕ over P is true in a state s (denoted as $s \models_{\mathsf{rg}} \phi$), then ϕ will also be true in a relaxed state (F^+, F^-) if every proposition that is true in s is also true in such a relaxed state. This is proven in the following lemma.

Lemma B.1 *Let P be a set of propositions, ϕ be an NNF formula, and $s, F^+, F^- \subseteq P$ be states. Then if $s \models_{\mathsf{rg}} \phi$, and (F^+, F^-) is such that:*

1. $(F^+, F^-) \models_{\mathsf{rg}} p$, *for every* $p \in s$, *and*
2. $(F^+, F^-) \models_{\mathsf{rg}} \neg p$, *for every* $p \in s^c$,

then $(F^+, F^-) \models_{\mathsf{rg}} \phi$.

Proof: The proof that follows is by induction on the structure of ϕ.
Base cases ($\phi = p$ or $\phi = \neg p$) They both follow directly from the conditions of this Lemma.
Induction We have the following cases

- if $\phi = \psi \wedge \xi$, then $s \models_{\mathsf{rg}} \psi$ and $s \models_{\mathsf{rg}} \xi$. By inductive hypothesis, also $(F^+, F^-) \models_{\mathsf{rg}} \psi$ and $(F^+, F^-) \models_{\mathsf{rg}} \xi$. It follows from Definition 4.1 that $(F^+, F^-) \models_{\mathsf{rg}} \phi$.
- if $\phi = \psi \vee \xi$, then the proof is analogous to the previous case.
- if $\phi = \forall x.\psi$, then for every $o \in \mathit{Objs}$ we have that $s \models_{\mathsf{rg}} \psi(x/o)$. By inductive hypothesis, for every $o \in \mathit{Objs}$ then $(F^+, F^-) \models_{\mathsf{rg}} \psi(x/o)$, hence by Definition 4.1, we have that $(F^+, F^-) \models_{\mathsf{rg}} \phi$.

- if $\phi = \exists x.\psi$, the proof is analogous to the previous case.

■

The final intermediate result is actually a version of Proposition 4.1 but for simple facts.

Lemma B.2 *Let s be a planning state, $R = (F_0^+, F_0^-)(F_1^+, F_1^-)\cdots(F_{m-1}^+, F_{m-1}^-)(F_m^+, F_m^-)$ be the relaxed planning graph constructed from s up to a fixed point. Moreover, let s_n be the state that results after performing a legal sequence of actions $a_1 \cdots a_n$ in s, then there exists some $k \leq m$ such that $(F_k^+, F_k^-) \models_{\mathsf{rg}} f$, for every $f \in s$, and such that $(F_k^+, F_k^-) \models_{\overline{\mathsf{rg}}} \neg f$ for every $f \in s^c$.*

Proof: Since R has been constructed to a fixed point, $F_{m-1}^+ = F_m^+$ and $F_{m-1}^- = F_m^-$, and $m > 0$. Moreover, assume that the set of states generated by performing the action sequence over s is $s_1 \cdots s_n$ (i.e., state s_i is generated after performing the sequence of actions $a_1 \cdots a_i$ over s). The following proof for the lemma is by induction on the length of the action sequence, n.

Base Case ($n = 0$) We prove that in this case we can consider $k = 0$. In this case the sequence of actions performed on s is empty. By definition of the construction of R, $F_k^+ = F_0^+ = s$ and $F_0^- = F_k^- = s^c$. Let f be an arbitrary fact.

1. $f \in s$. Then, by Definition 4.1, $(F_k^+, F_k^+) \models_{\overline{\mathsf{rg}}} f$, for $k = 0$ concluding the proof for this case.
2. $f \in s^c$. Then, again by Definition 4.1, we obtain $(F_k^+, F_k^-) \models_{\overline{\mathsf{rg}}} \neg f$, for $k = 0$.

Induction Let us assume that the theorem is true for $n-1$. We now prove that it is also true for n. We divide this proof into four cases. Again, assume f is an arbitrary fact.

1. $f \in s_n$ and $f \in s_{n-1}$. This case is trivial, since by inductive hypothesis we have that $(F_k^+, F_k^-) \models_{\overline{\mathsf{rg}}} f$ for some $k \leq m$.
2. $f \notin s_n$ and $f \notin s_{n-1}$. Again, by induction hypothesis $(F_k^+, F_k^-) \models_{\overline{\mathsf{rg}}} \neg f$ for some $k \leq m$.
3. $f \in s_n$ and $f \notin s_{n-1}$. Then, a_n must have added fact f when performed in s_{n-1}. We now prove that action a_n is executable at some level $k' \leq m-1$ of the relaxed graph, and that it will add fact f to the relaxed graph at level $k'+1 \leq m$.

   Let us assume that the precondition of action a_n is φ_P and that the condition of the conditional effect that adds f is φ_c. Then since both formulae are satisfied in s_{n-1}, we have that

   $$s_{n-1} \models_{\overline{\mathsf{rg}}} \varphi_P \wedge \varphi_c. \tag{B.1}$$

   Moreover, by inductive hypothesis, we have that there exists a $k' \leq m$ such that

   $$(F_{k'}^+, F_{k'}^-) \models_{\overline{\mathsf{rg}}} p, \qquad \text{for every } p \in s_{n-1} \tag{B.2}$$

   $$(F_{k'}^+, F_{k'}^-) \models_{\overline{\mathsf{rg}}} \neg p, \qquad \text{for every } p \in s_{n-1}^c \tag{B.3}$$

At this point, we can safely assume also that $k' < m$, because if k' were equal to m, then (B.2) and (B.3) also hold for $k' = m-1$, because the graph has been constructed to a fixed point.

Now, we combine equations (B.2), (B.3), and (B.1) with Lemma B.1 to conclude that $(F^+_{k'}, F^-_{k'}) \models_{\mathsf{rg}} \varphi_P \wedge \varphi_c$. Action a_n is therefore executable at level k' of the relaxed graph, and the condition φ_c, which enables the conditional effect that adds f is also true at level k'. Therefore, f is added to the relaxed graph at level $k = k'+1 \leq m$, concluding the proof for this case.

4. $f \notin s_n$ and $f \in s_{n-1}$. Proof is analogous to previous case.

■

Now we are ready to prove our result.

Proof (Proposition 4.1) : By Lemma B.2, we know that there exists a $k \leq m$ such that for each $p \in s_n$, $(F^+_k, F^-_k) \models_{\mathsf{rg}} p$, and for each $p \in s^c$ then $(F^+_k, F^-_k) \models_{\mathsf{rg}} \neg p$. Because $s_n \models_{\mathsf{rg}} \phi$ it follows from Lemma B.1 that $(F^+_k, F^-_k) \models_{\mathsf{rg}} \phi$. ■

B.2 Proof for Theorem 4.2

Before we start our proof we prove a lemma which establishes that, under the conditions of Theorem 4.2, if two nodes with exactly the same state have different *B*, *D*, or *O* metric value, then their lengths must also differ analogously.

Lemma B.3 *Let N_1 and N_2 be two search nodes that correspond to the same planning state s. Furthermore, let the metric M of the instance be NDVPL and depend on* (`total-time`). *If $R(N_1) \leq R(N_2)$, and:*

1. *R is either O or B, or*
2. *M is ATT and R is D.*

then $\texttt{length}(N_1) \leq \texttt{length}(N_2)$.

Proof: We divide the proof in two cases.

Case 1: R is either O or B. Then $R(N_1) = M(N'_1)$, where N'_1 is a hypothetical node with the same `length` as N_1 but in which possibly more preferences are satisfied. Analogously, $R(N_2) = M(N'_2)$ for a node N'_2 with the same `length` as N_2. Therefore,

$$M(N'_1) \leq M(N'_2). \tag{B.4}$$

Because the planning state associated to N_1 and N_2 are identical, we know that N'_2 and N'_1 are such that they satisfy exactly the same preferences, i.e., if Γ is the set of preferences of the planning instance,

for all $p \in \Gamma$ we have that $\texttt{is-violated}(p, N_1') = \texttt{is-violated}(p, N_2')$. Now, using the contra-positive of implication (2) in the NDVPL definition (Def. 4.3) and Equation B.4, we have that $\texttt{length}(N_1') \leq \texttt{length}(N_2')$. This implies that $\texttt{length}(N_1) \leq \texttt{length}(N_2)$, and concludes the proof for this case.

Case 2: R is D and M is ATT. Because M is ATT, then by Equation 4.1, $D(N_1) = M(N_1) + R_1$, where R_1 is an expression that does not depend on (`total-time`), i.e. it *only* depends on N_1's state. Likewise, $D(N_2) = M(N_2) + R_2$, where R_2 only depends on the state of N_2. Since both the states corresponding to N_1 and N_2 are equal, we have that $R_1 = R_2$. Hence, because $D(N_1) \leq D(N_2)$ we have that $M(N_1) \leq M(N_2)$, which by the contra-positive of implication (2) in the NDVPL definition (Def. 4.3) implies that $\texttt{length}(N_1) \leq \texttt{length}(N_2)$. This concludes this case, finishing the proof. ■

Now we are ready to prove our result. First, note that the search is restarted from scratch after the first plan is found. This also means that the closed list is reinitialized. Second, note that if two nodes N_1 and N_2 have the same state associated to them then both the G and the P functions evaluated on these nodes return the same value. Therefore, if $\text{USERHEURISTIC}(N_1) \leq \text{USERHEURISTIC}(N_2)$, then this means that the tie breaker functions used, say R, is such that $R(N_1) \leq R(N_2)$ where R is either O, B or D.

The sketch of the proof is as follows. We assume that a node N that leads to an optimal plan is discarded by the algorithm. Then we prove that if this happens then either the optimal was found or there is a node in the frontier that can be extended to another optimal plan.

Assume there exists an optimal plan $p_1 = a_1 a_2 \cdots a_n$ that traverses the sequence of states $s_0 s_1 \cdots s_n$. Let N_1 be a node formed by applying p_1 on s_0. Because the metric is NDVPL, we assume that this plan contains no cycles (otherwise, had the plan contained any cycles, by removing them we could not make it worse). Suppose further that at some point in the search, there is a node N that is generated by applying $a_1 a_2 \cdots a_j$ in the initial state (with $j < n$) and that is discarded by the algorithm in line 8. This means that there exists another closed node, say N_C that is associated the same state as N, and that is such that

$$\text{USERHEURISTIC}(N_C) \leq \text{USERHEURISTIC}(N). \tag{B.5}$$

Both nodes are associated the same state s_j, hence the `is-violated` counters are identical for each preference. This means that N_C is constructed from s_0 by a sequence of actions $b_1 b_2 \cdots b_k$. This sequence of actions gets to the same state s_j, hence the sequence $p_2 = b_1 b_2 \cdots b_k a_{j+1} \cdots a_n$ is also a plan.

Let N_2 be a node that would be constructed by applying p_2 in s_0. Now we prove that N_2 also corresponds to an optimal plan. We have two cases.

Case 1: The metric depends on (`total-time`). Because the Inequality B.5 implies that $R(N_C) \leq R(N)$, where R is either O, D or B, by Lemma B.3, we have that $\texttt{length}(N_C) \leq \texttt{length}(N)$, and therefore $k \leq j$. We clearly have that $\texttt{length}(N_2) \leq \texttt{length}(N_1)$, furthermore because all precondition counters are identical, it follows from the NDVPL condition that $M(N_2) \leq M(N_1)$. Given that N_1 represents an

optimal plan, we conclude that $M(N_2) = M(N_1)$, and therefore N_2 also represents an optimal plan.
Case 2: The metric does not depend on $(\texttt{total-time})$. Therefore, because node N_2 reaches the same state as N_1 does and M only depends on properties encoded in the state, $M(N_1) = M(N_2)$ and hence N_2 also represents an optimal plan. This concludes case 2.

Now, we know that since N_C, a predecessor of N_2 was expanded by the algorithm, one of the following things happen:

1. A successor of N_C is in *frontier*. In this case, the condition of Def. 4.5 follows immediately.

2. N_2 is in the closed list. This implies that the condition of Def. 4.5 is also satisfied.

3. A successor of N_C has been discarded by the algorithm. In this case, such a successor also leads to an optimal plan. This means that we could apply the same argument in this proof for such a node, leading to eventually satisfy the condition of Def. 4.5 since the algorithm has visited finitely many nodes.

Appendix C

Proofs for Chapter 5

We here provide the proofs of the two theorems, that is, we prove the correctness (sound and completeness) of our translations, and we prove the succinctness of the resulting PDDL planning instance.

C.1 Proof for Proposition 5.1

We first need to prove the following Lemma.

Lemma C.1 *Let σ_0 and σ' be programs. Then if*

$$[\sigma_0;\sigma',s_0] \vdash [\sigma_1;\sigma',s_1] \vdash \ldots \vdash [\sigma_n;\sigma',s_n]$$

then $[\sigma_0,s] \vdash^k [\sigma_k,s_k]$, *for all* $k \in [0,n]$.

Proof: By induction in n.
Base case ($n = 0$). The property is trivially true.
Induction. Let the property hold for $n = p$ we prove it for $n = p+1$. We know that $[\sigma_p;\sigma',s_p] \vdash [\sigma_{p+1};\sigma',s_{p+1}]$. By definition of *Tr* we have that $[\sigma_{p+1},s_{p+1}] \in Tr([\sigma_p,s_p],a)$, for some a. By definition of $\vdash$, the previous statement implies $[\sigma_p,s_p] \vdash [\sigma_{p+1},s_{p+1}]$, which concludes the proof. ■

Proof (Proposition 5.1) : We assume that the following holds:

$$q_0 = [\sigma_1;\sigma_2,s] \vdash q_1 \vdash q_2 \vdash \ldots \vdash q_{k-1} \vdash q_k = [nil,s'] \tag{C.1}$$

It is easy to see the following facts.

1. By definition of *Tr* if σ is a program that is not *nil*, the only possible transitions over $[\sigma;\sigma_2,r]$ produce a configuration $[\sigma';\sigma_2,r']$.

2. Since the last configuration in the sequence of Expression C.1 is $[nil, s']$ then, necessarily, at some intermediate configuration is of the form $q_p = [nil; \sigma_2, r''']$, for some $p \in [0, k-1]$.

From (1) and (2) we conclude that for all $j \in [1, p]$, $q_j = [\sigma_j; \sigma_2, s_j]$, for some s_j and σ_j.

Now, we apply Lemma C.1 and conclude from $[\sigma_1; \sigma_2, s] \vdash^p [nil; \sigma_2, s_p]$ that $[\sigma_1, s] \vdash^p [nil, s_p]$. Moreover, by definition of *Tr*, we have that $[nil; \sigma_2, s_p] \vdash [\sigma_2, s_p]$, and thus $q_{p+1}[\sigma_2, s_p]$. This concludes the proof.

■

C.2 Proof for Proposition 5.4

We divide the proof for each of three the cases.

1. The argument for this is similar to the one we use in the proof for Proposition 5.1. A transition on the while loop state $[\textbf{while}\,\phi\,\textbf{do}\,\sigma, s]$, will produce either $[nil, s]$, or $[\sigma; \textbf{while}\,\phi\,\textbf{do}\,\sigma, s]$. In the latter case, since we know that the while terminates (i.e., eventually transitions to $[nil, s']$), we can argue—by definition of *Tr*—that this can only happen if $[\sigma; \textbf{while}\,\phi\,\textbf{do}\,\sigma, s] \vdash^* [nil; \textbf{while}\,\phi\,\textbf{do}\,\sigma, s'']$, such that all states traversed in between such a computation are of the form $[\sigma_p; \textbf{while}\,\phi\,\textbf{do}\,\sigma, s_p]$. Furthermore, $[nil; \textbf{while}\,\phi\,\textbf{do}\,\sigma, s''] \vdash [\textbf{while}\,\phi\,\textbf{do}\,\sigma, s'']$. From state $[\textbf{while}\,\phi\,\textbf{do}\,\sigma, s'']$ we can apply the same reasoning, and finally conclude that all q_i are of the required form.

2. From 1. above it is straightforward to verify that $q_k = [\textbf{while}\,\phi\,\textbf{do}\,\sigma, r_k]$, for some r_k, since $[\sigma'; \textbf{while}\,\phi\,\textbf{do}\,\sigma, r_k]$ cannot transition to $[nil, s']$ in one step. Furthermore, let $q_i = [\textbf{while}\,\phi\,\textbf{do}\,\sigma, r_i]$, and let $i < k$. By definition of *Tr*, this can only happen iff q_{i+1} is not $[nil, s']$. In turn, by definition of *Tr*, $q_{i+1} \neq [nil, s']$ iff $r_i \models \phi$.

3. The proof for this follows straightforwardly from the form of the states in the sequence and Lemma C.1.

C.3 Correctness (Theorem 5.1)

We divide our proof into two parts: a soundness and a completeness result. Throughout the proof, we denote by $I_{\sigma,n,n'}$ the planning instance that results by first invoking $C(\sigma, n, [\,])$ and then following the remaining steps of the compilation, if such a call to C returns (L, L', n') for some L and some L'. Moreover, $I_{\sigma,n,n'}$'s initial state requires $state = s_n$ in the initial state, and the goal requires $state = s_{n'}$. Note that I_σ, as it is defined in the compilation section, corresponds to $I_{\sigma,0,n_{\text{final}}}$.

We start by proving three intermediate results.

Lemma C.2 *Let σ be a program, let I be a planning instance with initial state Init, and let $I_{\sigma,n,n'}$ be the instance generated by the compilation with the usual operator lists L and L'. Assume σ_1 is a subprogram of σ, such that $C(\sigma_1, n_1, E_1)$ was invoked during the top-level compilation, returning (L_1, L'_1, n'_1). Finally, let $\alpha = a_0 a_1 \cdots a_p$ be a plan for $I_{\sigma,n,n'}$. Let $a_0 \cdots a_j$ be a prefix of α such that $Succ(Init, a_0 \cdots a_j, s')$ and $s' \models state = s_k$, for some k such that $n_1 \leq k < n'_1$, then a_j is an instance of an operator in $L_1 \cdot L'_1$.*

Proof: Assume that a_j is an instance of an operator in $L \cdot L'$ but not in $L_1 \cdot L'_1$. Since all operators that where generated by C while compiling a subprogram of σ' are also in $L_1 \cdot L'_1$, there must be another subprogram of σ, say σ'', that is not a subprogram of σ' such that the compilation of σ'' generated an operator not in $L_1 \cdot L'_1$ that is possible when $state = s_k$. The recursive definition of the C operator does not admit this. If σ' and σ'' are two non-overlapping subprograms, the new preconditions that restrict the *state* variable are defined in such a way that they can never overlap for the same value of *state*. ■

Lemma C.3 *Let σ be a program with no program variables. Let I be a planning instance with initial state Init, and let $I_{\sigma,n,n'}$ be the instance generated by the compilation. Assume σ_1 is a subprogram of σ, such that $C(\sigma_1, n_1, [])$ was invoked during the top-level compilation, returning (L_1, L'_1, n'_1). Furthermore, let $\alpha = a_0 a_1 \cdots a_p$ be a plan for $I_{\sigma,n,n'}$ such that, when executed in Init, generates the sequence of states $s_0 s_2 \cdots s_p s_p$. Moreover, assume there exist two integers i and j, such $0 \leq i \leq j \leq p$ and such that $s_i \models state = s_{n_1}$, $s_j \models state = s_{n'_1}$ and for all r such that $i < r < j$, $s_r \models state = s_u$ with $n_1 \leq u < n'_1$.*

Finally, let I'_{σ',n_1,n'_1} be the instance that results from compiling σ' by calling $C(\sigma_1, n_1, [])$ on instance I', where I' is an instance with the operators from I, and such that its initial state is just like s_i but with no occurrence of the state fluent, and is such that its goal is empty.

Then $\alpha' = a_i a_{i+1} \cdots a_j$ is a plan for I'_{σ',n_1,n'_1}.

Proof: By Lemma C.2, actions in $a_i a_{i+1} \cdots a_j$ are instances of operators in I'_{σ',n_1,n'_1}. Moreover, since the initial state of I'_{σ',n_1,n'_1} is s_i, the sequence α' is also executable on I'_{σ',n_1,n'_1}, as while executing α' on I'_{σ',n_1,n'_1} the planning states traversed are identical those states traversed while performing the subsequence α' of α in $I_{\sigma,n,n'}$. Finally, after performing α', we reach a state where $state = s_{n'_1}$, and hence α' is a plan for I'_{σ',n_1,n'_1}. ■

We are now ready to prove the soundness part of the theorem.

C.3.1 Soundness Part

The statement we are now proving follows.

$\Rightarrow$ (Soundness):
Given a plan α for instance $I_\sigma = (D_\sigma, P_\sigma)$, show that $Filter(\alpha, D)$ is a plan for $I = (D, P)$ under the control of σ.

We prove this in several steps.

Lemma C.4 *Let σ be a program, $I = (D,P)$ a planning instance, and α a plan for planning instance $I_\sigma = (D_\sigma, P_\sigma)$. Then Filter$(\alpha, D)$ is a plan for I.*

Proof: Note that the preconditions of actions in D_σ are strictly more restrictive than their counterparts in D, as the original preconditions are conjoined with additional ones. Thus, whenever an action a of D_σ is executable in a state s and a is a domain action as opposed to any of the newly introduces bookkeeping actions, then the corresponding action a' in D is executable in s as well. Further, note that the additional effects of a in D_σ compared to a' in D only affect the new bookkeeping predicates and functions (bound, map, and state). Therefore, since the initial and goal state of I_σ differ from their counterparts in I only in terms of these bookkeeping predicates and functions, *Filter*(α, D) achieves the goal of P and thus *Filter*(α, D) is a plan for $I = (D,P)$. ■

To prove that the action sequence *Filter*(α, D) is also a plan under the control of σ, we have to show that the automaton $A_{\sigma,I}$ accepts it. We do this by induction over the structure of the program σ in the following two lemmata.

Lemma C.5 *Let σ be a program without the π(x-t) construct, $I = (D,P)$ a planning instance, and α a plan for planning instance $I_{\sigma,n,n'} = (D_\sigma, P_\sigma)$. Then Filter$(\alpha, D)$ is an execution of σ in I.*

Proof: Throughout this proof we will refer to the compilation result $C(\sigma, n, E) = (L, L', n')$ used to construct $I_{\sigma,n,n'}$. Since there are no π(x-t) constructs, we can assume that the E argument of C is always empty and can ignore any *bound* and *map* preconditions and effects upon these predicates for now. The program does not contain any program variables.

The proof proceeds by induction over the structure of σ as follows:

$\sigma = nil$**:** By definition of C, both L and L' are empty, and therefore no operators are included in D_σ. Thus the plan is empty. The empty sequence is accepted by $A_{\sigma,I}$, because $[nil, s]$ is a final state.

$\sigma = a, a \in \mathcal{A}$**:** By definition of the translation, the only operator in D_σ is action a. Thus, the only potentially possible action in any state where $state = s_n$ is a. Since the goal, by construction, requires $state = s_{n+1}$, then $\alpha = a$, and a must be possible in the initial state. From Eq. 5.4 we know that a is accepted by $A_{\sigma,I}$.

$\sigma = \phi?$**:** By definition of the translation, the only operator in D_σ is $test\_n\_n_1$, which is potentially possible in any state where $state = s_n$. Since the goal, by construction, requires $state = s_{n+1}$, $\alpha = test\_n\_n_1$, and since this is a plan, we know that its preconditions are satisfied in the initial state, hence $Init \models \phi$ and thus $A_{\sigma,I}$ accepts[1] $\varepsilon = Filter(\alpha, D)$ by Eq. 5.6.

[1] We denote the empty sequence of actions by ε.

These are the base cases. Now for the induction steps:

$\sigma = (\sigma_1;\sigma_2)$**:** Assume that $C(\sigma_1, n, E)$ and $C(\sigma_2, n_1, E)$ where invoked while compiling σ, for some n_1.

By construction of I, any plan $\alpha = a_0 a_1 \cdots a_n$ for I_σ can be partitioned into two parts α_1 and α_2 such that $\alpha = \alpha_1\alpha_2$, and such that $state = s_{n_1}$ in the state s' that results after performing α_1 over I_σ.

Let us define $I' = I$, then, by Lemma C.3, $\vec{a_1}$ is a plan for I'_{σ_1,n,n_1}. Moreover, let us define I'' as a planning instance whose initial state is s' but with no information about the state. By Lemma C.3, α_2 is a plan for I''_{σ_2,n_1,n_2}.

By induction hypothesis we know that the automaton $A_{\sigma_1,I'}$ accepts any plan for I'_{σ_1,n,n_1} for I'. Analogously, $A_{\sigma_2,I''}$ accepts any plan for I''_{σ_2,n_1,n_2}.

It now follows from the definition of Tr (Eq. 5.7) and a similar argument as in the proof for Lemma C.3 that $\alpha_1\alpha_2$ is also accepted by $A_{\sigma,I}$.

$\sigma = (\sigma_1|\sigma_2)$**:** From the definition of C we know that any plan for I_{σ,n,n_2+1} must start with either *noop_n_*$(n+1)$ or *noop_n_*(n_1+1). After that, by induction hypothesis and Lemma C.3, the only possible action sequences are those that are plans for $I_{\sigma_1,n+1,n_1}$ or I_{σ_2,n_1+1,n_2}. These sequences are accepted by their respective automata $A_{\sigma_1,I}$ and $A_{\sigma_2,I}$. By its definition, the language accepted by $A_{\sigma,I}$ is the union of the two languages of these automata, and the additional *noop* actions are filtered out.

$\sigma =$ **if** ϕ **then** σ_1 **else** σ_2**:** From the definition of C for this case we know that any plan for I_{σ,n,n_3} must start with either *test_n_n'* or *test_n_n''*, with $n' = n+1$ and $n'' = n_1+1$, depending on whether ϕ holds in the initial state. After that, by induction hypothesis and Lemma C.3, the only possible action sequences are those that are plans for I_{σ_1,n',n_1} or I_{σ_2,n'',n_2}. These sequences are accepted by their respective automata $A_{\sigma_1,I}$ and $A_{\sigma_2,I}$, by induction hypothesis. By its definition, the language accepted by $A_{\sigma,I}$ is the one accepted by the former if ϕ holds in the initial state, and otherwise the language of the latter. The *noop* and *test* actions are filtered out.

$\sigma =$ **while** ϕ **do** σ'**:** From the definition of C for this case we know that any plan for $I_{\sigma,n,n'}$, with $n' = n_1+1$, must start with either *test_n_n''*, with $n'' = n+1$, if ϕ holds in the initial state, or *test_n_n'*, otherwise. In the former case, by Lemma C.3, the only action sequence possible will start with a plan for I_{σ',n'',n_1} which, by induction hypothesis, is accepted by the automaton $A_{\sigma',I}$, followed by *noop_*n_1*_n* which, inductively, implies that it is followed by a plan for $I_{\sigma,n,n'}$. By definition of $A_{\sigma,I}$, in the case where $s \models \phi$, it accepts sequences which begin with sequences accepted by I_{σ',n'',n_1}, followed by any other sequence accepted by $A_{\sigma,I}$. Otherwise, if ϕ does not hold initially, *test_n_n'*, which is possible when ϕ doesn't hold, leads to a final state of $I_{\sigma,n,n'}$ and the filtered plan is empty. Analogously $A_{\sigma,I}$ accepts the empty language if ϕ doesn't hold. Thus, $A_{\sigma,I}$ accepts any plan for $I_{\sigma,n,n'}$.

$\sigma = \sigma'^*$**:** From the definition of C for this case and Lemma C.2 we know that any plan for I_{σ,n,n_2}, must either consist of $noop(n,n_2)$, which after filtering results in the empty plan which is trivially accepted by $A_{\sigma,I}$, or a plan for I_{σ',n,n_1} followed by $noop(n_1,n)$ and, recursively, any other plan for $I_{\sigma,n,n'}$. In the latter case, by induction hypothesis, any such plan is accepted by the sequence of automaton $A_{\sigma',I}$ and $A_{\sigma,I}$, which precisely meets the definition of $A_{\sigma,I}$.

■

Now for the case with program variables.

Lemma C.6 *Let σ be a program, possibly with $\pi(x\text{-}t)$ constructs, $I = (D,P)$ a planning instance, and α a plan for planning instance $I_\sigma = (D_\sigma, P_\sigma)$. Then Filter$(\alpha, D)$ is an execution of σ in I.*

Proof: The proof proceeds by induction over the number of $\pi(x\text{-}t)$ constructs in σ.

If σ is program variable free ($\pi(x\text{-}t)$ does not occur), then, trivially by Lemma C.5 the proposition holds.

Assume $\sigma = \pi(x\text{-}t)\sigma'$, and let $\alpha' = a_0 a_1 \cdots a_n$ such that $\alpha' \cdot free_{n_1}(x)$ is a plan for I_σ. First, we prove that there exists an $o \in Objs$ such that $a_0 a_1 \cdots a_n$ is a plan for $I_{\sigma'|x/o}$.

Let us assume that the state trajectory generated when performing $a_0 a_1 \cdots a_n$ in $Init$ is $s_0 s_1 \cdots s_n$. Observe the actions in the plan cannot delete $map(x)$ or delete $bound(x,o)$. Furthermore, if $bound(x,o)$ is true in a certain state, no action will add $bound(x,o')$ for any o' different from o. Hence, there exists a j ($0 \leq j \leq n$) such that

- $s_i \not\models map(x)$ and $s_i \not\models bound(x,o)$, for any $o \in Objs$ and any $i < j$.
- $s_i \models bound(x)$ and $s_i \models map(x,v)$ for all i s.t. $j \leq i \leq n$ and some $v \in Objs$.

We claim that $a_0 a_1 \cdots a_n$ is a plan for $I_{\sigma'|x/v}$. The proof for the claim is split in two parts: (a) we prove that the sequence $a_0 a_1 \cdots a_n$ is legally executable in $I_{\sigma'|x/v}$, then (b) we prove that it reaches the goal.

For proving (a), note that the only difference between I_σ and $I_{\sigma'|x/v}$ are the preconditions of some of its operators. For each occurrence of $bound(x) \rightarrow map(x,x_i)$ (for some x_i) in an operator in I_σ there is an occurrence of $x_i = v$ in $I_{\sigma'|x/v}$. It is easy to see that the preconditions of the first $j-1$ actions of the sequence, $a_0 a_1 \cdots a_{j-2}$, are satisfied in $I_{\sigma'|x/v}$. Indeed, note that because $bound(x)$ is not added by these actions in I_σ, by the definition of C, it means that the subformula of the precondition of the operator of I_σ that evaluated to true at that point is identical to that of the respective operator in $I_{\sigma'|x/v}$. Now let's focus on action a_{j-1}. This action *adds* $bound(x)$ and $map(x,v)$. By the construction of C this means that the precondition evaluated $bound(x) \rightarrow map(x,x_i)$ to be true in the state were a_{j-1} was performed (this happens because $bound(x)$ is false). Because after performing a_{j-1}, $map(x,v)$ is added, it means that

the parameter x_i of the operator took value v, while satisfying all additional preconditions. On the other hand, in $I_{\sigma'|x/v}$, the condition to be checked by the respective operator is instead $x_i = v$, which we know can be made true while satisfying additional preconditions of the operator, because a_{j-1} was executable in I_σ. For the remaining part of the sequence, $a_j a_{j+1} \cdots a_n$ the proof is analogous. When performed in I_σ, some of these actions will evaluate $bound(x) \rightarrow map(x, x_i)$ to true, with the side effect of making the parameter x_i equal to v. On the other hand, in $I_{\sigma'|x/v}$, the same effect is achieved but by the explicit $x_i = v$ in the precondition. Hence, the precondition in $I_{\sigma'|x/v}$ will also be satisfied.

The proof for (b) is straightforward. Since the goal does not mention any bookkeeping predicates, the sequence α' produces the same state in $I_{\sigma'|x/v}$ as $\alpha' \cdot free_{n_1}(x)$ in I_σ.

The proof now follows from Lemma C.5. ■

C.3.2 Completeness Part

The statement we are now proving follows.

> $\Leftarrow$ (Completeness):
> *Given a plan α for I under the control of σ, show that there exists a plan α' for I_σ, such that $\alpha = Filter(\alpha', D)$.*

The proof again proceeds by induction over the structure of the program σ, and again we first show the case for programs without $\pi(x\text{-}t)$ constructs, i.e. without program variables.

Lemma C.7 *Let σ be a program without the $\pi(x\text{-}t)$ construct, $I = (D, P)$ a planning instance, and α a plan for I under the control of σ, then there exists a plan α' for $I_{\sigma,n,n'}$ such that $\alpha = Filter(\alpha', D)$.*

Proof: We will again refer to the compilation result $C(\sigma, n, E) = (L, L', n')$ used to construct $I_{\sigma,n,n'}$, and occasionally also to variables occurring in the particular compilation case considered in the induction proof. Again, since there are no $\pi(x\text{-}t)$ constructs, we can assume that the E argument of C is always empty and can ignore any *bound* and *map* preconditions and effects upon these predicates for now. The program does not contain any program variables.

By assumption we know that $A_{\sigma,I}$ accepts the plan α. The induction over the structure of σ is as follows:

$\sigma = nil$: $A_{\sigma,I}$ only accepts the empty language, since there are no transitions defined for the *nil* program, but $[nil, s]$ is an accepting state for any state s over I. Thus $\alpha = \varepsilon$. Since both initial an goal state of $I_{\sigma,n,n'}$ only require $state = s_n$ on top of the original initial and goal state of I, and $n' = n$, $\alpha' = \varepsilon = \alpha$ is also a plan for $I_{\sigma,n,n'}$ and $\alpha = Filter(\alpha', D)$.

$\sigma = a, a \in \mathcal{A}$: In this case $\alpha = a$. Since in the compilation E is empty, the preconditions of the operator corresponding to a in $I_{\sigma,n,n'}$ are the same as those for a in I, except that $state = s_n$ has to hold. This condition already true in the initial state of $I_{\sigma,n,n'}$. Also, a goal state of $I_{\sigma,n,n'}$ is reached

after executing a in $I_{\sigma,n,n'}$, since the new operator, by definition of C has $state = s_{n+1}$ as an effect, which, by construction, is the only additional requirement in the goal state of $I_{\sigma,n,n'}$ compared to I. Thus α is a plan for $I_{\sigma,n,n'}$, and trivially $\alpha = Filter(\alpha, D)$.

$\sigma = \phi$**?:** Again, the plan has to be the empty sequence, since this is the only one accepted by $A_{\sigma,I}$. Also, by definition of $A_{\sigma,I}$, the initial state *Init* of I satisfies ϕ. Let $\alpha' = test\_n\_n'$. This is a plan for $I_{\sigma,n,n'}$, because by its construction in the definition of $Ctest$ its precondition is $state = s_n \wedge \phi$. This is satisfied since the initial state of $I_{\sigma,n,n'}$ is like that of I plus the assertion that $state = s_n$. Since ϕ cannot mention the new special fluent *state* its truth value does not differ between the initial state of $I_{\sigma,n,n'}$ and that of I itself. Further, $test\_n\_n'$ sets $state = s_{n'}$ as its only effect (E is empty), thus satisfying the goal of $I_{\sigma,n,n'}$. Finally, $\alpha = \varepsilon = Filter(test\_n\_n', D)$.

These are the base cases. Now for the induction steps:

$\sigma = (\sigma_1; \sigma_2)$**:** We start this case by stating an intermediate result.

Claim: If α is accpeted by $A_{\sigma,I}$, then α can be decomposed into two parts α_1 and α_2, such that $\alpha = \alpha_1\alpha_2$, and such that $[nil; \sigma_2, s'] \in Tr([\sigma_1; \sigma_2, Init], \alpha_1)$, for some s' and such that $[nil, s''] \in Tr([\sigma_2, s'], \alpha_2)$. Intuitively, this means that the automaton's state $[nil; \sigma_2, s']$ is part of an accepting path of states for α. **Proof:** Straightforward (but lengthy) by induction on the structure of σ_1.

Let us assume that $\alpha = \alpha_1\alpha_2$, for α_1 and α_2 as defined above. Furthermore let us define I^1 as an instance just like I except that its goal is to get to state s' (as defined above). Moreover, we define I^2 to be just like I but such that its initial state is s'. Observe now that α_1 and α_2 are clearly accepted by A_{σ_1,I^1} and A_{σ_2,I^2}. Indeed, this follows straightforwardly from the claim and the fact that the transition function for A_{σ_1,I^1} and A_{σ_2,I^2} are subsets of the transition function for $A_{\sigma,I}$.

By induction hypothesis, there are plans α_1', α_2' for $I^1_{\sigma_1,n_1,n_1'}$ and $I^2_{\sigma_2,n_2,n_2'}$ for any two integers n_1, n_2, such that $\alpha_1 = Filter(\alpha_1', D)$ and $\alpha_2 = Filter(\alpha_2', D)$. Choosing $n_2 = n_1'$ as defined by the compilation of σ_1 with parameter $n = n_1$, we get that the initial state of $I^2_{\sigma_2,n_2,n_2'}$ is a goal state of $I^1_{\sigma_1,n_1,n_1'}$ and thus $\alpha' = \alpha_1' \cdot \alpha_2'$ is a plan for $I_{\sigma,n,n'}$. Since the concatenation does not introduce any new actions we get $\alpha = Filter(\alpha', D)$.

$\sigma = (\sigma_1 | \sigma_2)$**:** By definition, $A_{\sigma,I}$ accepts the union of the sets of plans for σ_1 and σ_2, i.e. α is accepted by either $A_{\sigma_1,I}$ or $A_{\sigma_2,I}$.

Assume it is a plan under the control of σ_1 (i.e., it is accepted by $A_{\sigma_1,I}$). By induction hypothesis there is a plan α_1 for $I_{\sigma_1,n_1,n_1'}$ for any integer n_1, such that $\alpha = Filter(\alpha_1', D)$. Then $\alpha' = noop\_n\_(n{+}1) \cdot \alpha_1' \cdot noop\_n_1\_(n_2{+}1)$ is a plan for I_{σ,n,n_2+1}, where n_2 is defined in the compilation, and since the *noop* actions are filtered again $\alpha = Filter(\alpha', D)$. The case when α is a plan under the control of σ_2 is analogous with the plan $\alpha' = noop\_n\_(n_1{+}1) \cdot \alpha_2' \cdot noop\_n_2\_(n_2{+}1)$, n_1, n_2 are defined by the compilation.

$\sigma = \textbf{if}\,\phi\,\textbf{then}\,\sigma_1\,\textbf{else}\,\sigma_2$**:** Depending on whether or not *Init* $\models \phi$, α is a plan under the control of σ_1 or σ_2, i.e. it is either accepted by $A_{\sigma_1,I}$ or $A_{\sigma_2,I}$. Assume *Init* $\models \phi$. Then, α_1 is accepted by $A_{\sigma_1,I}$, and by induction hypothesis, there is a plan α'_1 for I_{σ_1,n_1,n'_1} for any integer n_1 s.t. $\alpha = \mathit{Filter}(\alpha'_1, D)$. Then $\alpha' = \mathit{test\_n\_}(n{+}1) \cdot \alpha'_1 \cdot \mathit{noop\_n_1\_n_3}$ is a plan for $I_{\sigma,n,n'}$ and by definition of *Filter* we have $\alpha = \mathit{Filter}(\alpha', D)$. Analogously when *Init* $\not\models \phi$, $\alpha' = \mathit{test\_n\_}(n_1{+}1) \cdot \alpha'_2 \cdot \mathit{noop\_n_2\_n_3}$ is a plan for $I_{\sigma,n,n'}$ and again $\alpha = \mathit{Filter}(\alpha', D)$.

$\sigma = \textbf{while}\,\phi\,\textbf{do}\,\sigma'$**:** The induction step for this case is itself by induction. We refer to this induction as "inner induction", and to the other as "outer induction". The inner induction is on the length of the action sequence α.

As our inner base case, assume that *Init* $\not\models \phi$, then $\alpha = \varepsilon$ ($|\alpha| = 0$). Then $\mathit{test\_n\_n'}$ is a plan for $I_{\sigma,n,n'}$ for any integer n, because by construction the precondition for this test action is $\neg\phi \wedge \mathit{state} = s_n$, and its effect asserts $\mathit{state} = s_{n'}$. Also $\varepsilon = \mathit{Filter}(\mathit{test\_n\_n'}, D)$. This concludes the proof for the inner base case.

Now, as our inner induction hypothesis, we assume the theorem holds for all sequences of actions whose length is strictly less that k. Moreover, assume $|\alpha| = k$. In this case, we have that *Init* $\models \phi$, and then $\alpha = \alpha_{\sigma'} \cdot \alpha''$ is a plan for $I_{\sigma,n,n'}$, where $\alpha_{\sigma'}$ is a sequence accepted by $A_{\sigma',I}$, and α'' is accepted by $A_{\sigma,I'}$, where I' is like I except that the initial state is the state reached after executing $\alpha_{\sigma'}$ in *Init*. Then, by outer induction hypothesis there is a plan $\alpha'_{\sigma'}$ for I_{σ',n_3,n'_3} for any integer n_3, s.t. $\alpha_{\sigma'} = \mathit{Filter}(\alpha'_{\sigma'}, D)$, and by inner induction hypothesis there is a plan α''' for I'_{σ,n_2,n'_2} for any integer n_2 s.t. $\alpha'' = \mathit{Filter}(\alpha''', D)$. Choosing $n_2 = n$ and $n_3 = n+1$ we get that $\alpha' = \mathit{test\_n\_}(n{+}1) \cdot \alpha'_{\sigma'} \cdot \mathit{noop\_n_1\_n} \cdot \alpha'''$ is a plan for $I_{\sigma,n,n'}$, where n_1 is defined by the compilation for σ. Finally, again, $\alpha = \mathit{Filter}(\alpha', D)$.

$\sigma = \sigma'^*$**:** We again require an inner induction on the length of α. Assume that $\alpha = \varepsilon$, then $\mathit{noop\_n\_n'}$ is a plan for $I_{\sigma,n,n'}$ and trivially $\alpha = \mathit{Filter}(\mathit{noop\_n\_n'}, D)$. This concludes the proof for the base case of the inner induction. Assume now for the inner induction case that the theorem holds for all sequences of length less than k, where $|\alpha| = k$. In this case, $\alpha = \alpha_1 \cdot \alpha_2$ where α_1 is accepted by $A_{\sigma',I}$ and α_2 is accepted by $A_{\sigma,I'}$ where I' is like I except that the initial state is the state reached after executing $\alpha_{\sigma'}$ in *Init*. Then, by outer induction hypothesis there is a plan α'_1 for I_{σ',n_3,n'_3} for any integer n_3 s.t. $\alpha_1 = \mathit{Filter}(\alpha'_1, D)$, and by inner induction hypothesis there is a plan α'_2 for I'_{σ,n_2,n'_2} for any integer n_2 s.t. $\alpha_2 = \mathit{Filter}(\alpha'_2, D)$. Choosing both $n_3 = n$ and $n_2 = n$ we get that $\alpha' = \alpha'_1 \cdot \mathit{noop\_n_1\_n} \cdot \alpha'_2$ is a plan for $I_{\sigma,n,n'}$, where n_1 is defined by the compilation. Again, by the two induction hypotheses and the fact that $\mathit{noop\_n_1\_n}$ is filtered out, $\alpha = \mathit{Filter}(\alpha', D)$.

■

Now for the case with program variables.

Lemma C.8 *Let σ be a program over a planning instance $I = (D,P)$ (possibly containing π(x-t) constructs), and α a plan for I under the control of σ, then there exists a plan α' for $I_{\sigma,n,n'}$ such that $\alpha = Filter(\alpha', D)$.*

Proof: The proof proceeds by induction over the number of π(x-t) constructs occurring in σ. The base case, where this number is zero, is given by Lemma C.7.

Otherwise, assume $\sigma = \pi(x\text{-}t, \sigma')$ for some arbitrary other program σ' over I. By the definition of $A_{\sigma,I}$, α is accepted by some automaton $A_{\sigma|_{x/o},I}$ where in σ all occurrences of x are replaced by some (but in all occurrences the same) o such that $(o,t) \in \tau_D \cup \tau_P$. We show that (i) $\alpha' = \alpha \cdot free\_n_1(x)$ is a plan for $I_{\sigma,n,n'}$ for any integer n, where n_1 is defined in the compilation of σ using n as the integer parameter. We further need to show that (ii) in a state s' reached after performing α' in any state s that satisfies $\neg bound(x) \wedge \neg(\exists y).map(x,y)$, we again get $s' \models \neg bound(x) \wedge \neg(\exists y).map(x,y)$. Obviously, the initial state *Init* has this property for all program variables occurring in σ.

(i) By assumption α is accepted by $A_{\sigma|_{x/o},I}$ for some o, i.e. after replacing all occurrences of x in σ with o, and is a plan for I. By induction hypothesis and Lemma C.7 there exists a plan α'_1 for $I_{\sigma|_{x/o},n,n'}$ for any integer n such that $\alpha = Filter(\alpha'_1, D)$. We show that this is also a plan for $I_{\sigma,n,n'}$ after minor modifications to the occurring test actions, and which in particular do not result in a different result when applying *Filter*. Compile σ as defined using $C(\sigma, n, [\,]) = (L, L', n')$. For any test action occurring in α'_1 whose corresponding operator definition in L has x as a formal parameter, add o as an additional argument at the position where x appears in the operator definition, creating a new sequence α'_2. We show that this sequence is a plan for $I_{\sigma,n,n'}$: Let a_1 be the first action in α'_2 whose corresponding operator definition in L has x as a formal parameter. The corresponding actual parameter is o. Then, since in the initial state s of $I_{\sigma,n,n'}$ we have that $s \models \neg bound(x) \wedge \neg(\exists y).map(x,y)$, s satisfies the preconditions of a_1, because the only preconditions on top of those defined in $I_{\sigma|_{x/o},n,n'}$ are $bound(x) \rightarrow map(x,o)$. The action will further have as an effect $bound(x)$ $map(x,o)$. Hence, all following actions a_k in α'_2 whose corresponding operator in L has x as a formal parameter, will also be possible and have the same effects as in $I_{\sigma|_{x/o},n,n'}$ (by construction of $\sigma|_{x/o}$), because also they have o as actual parameter, and since α'_2 cannot mention any action $free\_n_i(x)$, for any i, we have for all states s'' visited later on during the execution of α'_2 that $s'' \models bound(x) \wedge map(x,o)$ which entails the preconditions of a_k in $I_{\sigma,n,n'}$. Since further only the truth value of *bound* and *map* are changed compared to the effects in $I_{\sigma|_{x/o},n,n'}$, the goal, which by construction doesn't mention either of these predicates, is reached at the end. Hence, α'_2 is a plan for $I_{\sigma,n,n'}$. Also $\alpha = Filter(\alpha'_2 \cdot free\_n_1(x), mD)$.

(ii) Clearly, since for any n_i, $free\_n_i(x)$ has $\neg bound(x) \wedge (\forall y).\neg map(x,y)$ as an effect, any state s' reached after executing $\alpha'_2 \cdot free\_n_1(x)$ in any other state satisfies this. ■

Theorem 5.1 then follows directly from Lemmata C.6 and C.8 for $n = 0$ and n_{final} as defined by the compilation $C(\sigma, 0, [\,]) = (L, L', n_{final})$.

C.4 Succinctness (Theorem 5.2)

Proof: Theorem 5.2

The compilation of each programming construct, as defined by C, introduces a constant number of new operators into I_σ or extends the definition of one of the operators of I with a constant number of additional preconditions and effects. In all cases, the size of the new preconditions and effects is bounded by a constant factor in the number of elements of E. From the definition of C for π it follows that the maximal length of E occurring during the compilation of σ is exactly the number of nested π constructs, k. Hence, if the program has size n, then there are no more than n programming constructs. Since also each construct is considered exactly once by C, there can be no more than n operators in I_σ, each of size $O(k+p)$, where p is the size of the largest operator in the original instance. Hence, overall I_σ has size $O(k \cdot n)$. ■

Appendix D

Proofs for Chapter 6

D.1 Proof for Lemma 6.2

Before proving this lemma, we prove an intermediate result.

Lemma D.1 *Let $\mathcal{D}$ be a theory of action containing the reflexivity axiom for K. Then,*

$$\mathcal{D} \models (\forall s).\mathbf{KWhether}(\phi, s) \supset \{(\phi[s] \supset \mathbf{Knows}(\phi, s)) \wedge (\neg\phi[s] \supset \mathbf{Knows}(\neg\phi, s))\}$$

Proof: Let $\mathcal{M} \models \mathcal{D}$. Now, assume $\mathcal{M} \models \phi[s]$. If $\mathcal{M} \models \mathbf{KWhether}(\phi, s)$, then $\mathcal{M} \models \mathbf{Knows}(\phi, s)$ or $\mathcal{M} \models \mathbf{Knows}(\neg\phi, s)$. However, $\mathcal{M} \not\models \mathbf{Knows}(\neg\phi, s)$ since otherwise, by reflexivity we would have $\mathcal{M} \models \neg\phi[s]$ which would be a contradiction. Hence it is the case that $\mathcal{M} \models \mathbf{Knows}(\phi, s)$.

On the other hand, if we assume $\mathcal{M} \models \neg\phi[s]$ we conclude by analogy that $\mathcal{M} \models \mathbf{Knows}(\neg\phi, s)$. This implies that any model of $\mathcal{D}$ satisfies the formula of the lemma, and concludes the proof. ■

Proof (Lemma 6.2) : For the ($\Leftarrow$) direction, observe that if $\mathcal{M}$ is a model of $\mathcal{D}$ and $\mathcal{D}$ contains the reflexivity axiom, $\mathcal{M} \models \mathbf{Knows}(\phi, s)$ implies $\mathcal{M} \models \phi[s]$, for any s. The rest of the proof is straightforward since in all cases of programs, the formula for Do_K clearly implies that for Do.

For the ($\Rightarrow$) direction, the proof proceeds by induction in the structure of δ. We use the definition of ssf, plus Lemma D.1 to show in all cases the formula that corresponds to Do implies the formula that corresponds to Do_K. ■

D.2 Proof for Theorem 6.3

We first prove the following Lemma.

Lemma D.2 *Let $\mathcal{D}$ be a theory of action such that $\mathcal{K}_{init}$ contains the reflexivity axiom. Let C be a set of Golog deterministic tree programs. Then, for all fluents F in the language of $\mathcal{D}$ that are not K, and for every $\delta \in C$ such that $\mathcal{D} \models ssf(\delta, s)$, theory $\mathsf{Comp}[\mathcal{D}, C]$ entails*

$$Do_K(\delta, s, s') \supset (F(\vec{x}, s') \equiv F(\vec{x}, do(prim_\delta, s)))$$

Proof: Let $\mathcal{D}' = \mathsf{Comp}[\mathcal{D}, C]$. Because $\phi[s]$ is regressable in s all its atoms can be reduced in to formulae that only refer to either situation-independent predicates or fluent predicates. Then, it suffices to prove that

1. $\mathcal{D}' \models Do_K(\delta, s, s') \wedge F(\vec{x}, s') \supset F(\vec{x}, do(prim_\delta, s)))$, and
2. $\mathcal{D}' \models Do_K(\delta, s, s') \wedge F(\vec{x}, do(prim_\delta, s)) \supset F(\vec{x}, s')$.

Proof for 1: Suppose $\mathcal{M}$ is a model of $\mathcal{D}'$ such that $\mathcal{M} \models (Do_K(\delta, s, S') \wedge F(\vec{x}, S'))$, for some situation denoted by S'. From Proposition 6.3 and Lemma 6.2, we have that $\mathcal{M} \models Do^-(\delta, s, S') \wedge F(\vec{x}, S')$, and that $\mathcal{M} \models \mu_i(s) \wedge F(\vec{x}, S')$, for some μ_i of Lemma 6.1 . Since regression is correct and $\mathcal{M}$ also satisfies axiom (6.37), it follows immediately that $\mathcal{M} \models F(\vec{x}, do(prim_\delta, s))$.
Proof for 2: Assume $\mathcal{M}$ is a model for $\mathcal{D}'$ such that $\mathcal{M} \models (Do_K(\delta, s, s') \wedge F(\vec{x}, do(prim_\delta, s)))$, for any situations s, s'. By the successor state axiom of F, and correctness of regression, we conclude that $\mathcal{M} \models F(\vec{x}, do(prim_\delta, s))$ iff

$$\begin{aligned} \mathcal{M} \models & (Do^-(\delta, s, S_1) \wedge F(\vec{x}, S_1)) \vee \\ & F(\vec{x}, s) \wedge (\forall s_2)\,(Do^-(\delta, s, s_2) \supset F(\vec{x}, s_2)), \end{aligned}$$

for some situation S_1. Since δ is deterministic and given that $\mathcal{M} \models Do_K(\delta, s, S')$, by Proposition 6.3 and Lemma 6.2, we have that $\mathcal{M} \models S_1 = s'$. The assertion above reduces to $\mathcal{M} \models F(\vec{x}, s') \vee F(\vec{x}, s) \wedge F(\vec{x}, s')$, from which we conclude that $\mathcal{M} \models F(\vec{x}, s')$. ■

Proof (Theorem 6.3) : The proof of the theorem is now straightforward by using Lemma D.2. ■

D.3 Proof for Theorem 6.4

First we need the following result.

Lemma D.3 *Let $\mathcal{D}$ be a theory of action such that $\mathcal{K}_{init}$ contains the reflexivity axiom. Furthermore, let δ be a Golog deterministic tree program.*

$$\begin{aligned} \mathcal{D} \models K(s', s) \wedge K(do([a_1, \ldots, a_n], s'), do([a_1, \ldots, a_n], s)) \supset \\ \{Do_K(\delta, s, do([a_1, \ldots, a_n], s)) \supset Do(\delta, s', do([a_1, \ldots, a_n], s'))\} \end{aligned}$$

Proof: We proceed by induction in the structure of δ. We first observe that from the successor state axiom for K

$$\mathcal{D} \models \bigwedge_{i=0}^{n} K(do([a_1,\ldots,a_i],s'),do([a_1,\ldots,a_i],s))$$

Now let $\mathcal{M}$ be a model of $\mathcal{D}$. Observe that for any situation-suppressed formula ϕ if

$$\mathcal{M} \models \mathbf{Knows}(\phi, do([a_1,\ldots,a_i],s))$$

for some $i \leq n$ then $\mathcal{M} \models \phi[do([a_1,\ldots,a_i],s')]$. The rest of the proof is straightforward. ■

Let $\mathcal{D}' = \mathsf{Comp}[\mathcal{D},C]$. It suffices to prove the theorem for any arbitrary situation-suppressed fluent symbol F different from K. By expanding the definition of **Knows**, it suffices to prove

$$\mathcal{D}' \models (\forall \vec{x},s,s_1).Do_K(\delta,s,s_1) \supset \{(\exists s'')\,(K(s'',s_1) \wedge F(\vec{x})[s'']) \equiv (\exists s'')\,(K(s'',do(prim_\delta,s)) \wedge F(\vec{x})[s'])\},$$

Proof: (⇒) We prove that

$$\mathcal{D}' \models (\forall \vec{x},s,s_1).Do_K(\delta,s,s_1) \supset \{(\exists s'')\,(K(s'',s_1) \wedge F(\vec{x})[s'']) \supset (\exists s'')\,(K(s'',do(prim_\delta,s)) \wedge F(\vec{x})[s''])\},$$

Suppose $\mathcal{M} \models \mathcal{D}'$ and that for some situation denoted by S'',

$$\mathcal{M} \models Do_K(\delta,s,s_1) \wedge K(S'',s_1) \wedge F(\vec{x})[S''],$$

Notice that $\mathcal{M} \models s \sqsubseteq s_1$, and since $\mathcal{M} \models K(S'',s_1)$, there exists situation denoted by S''' such that $\mathcal{M} \models S''' \sqsubseteq S''$ and such that

$$\mathcal{M} \models Do_K(\delta,s,s_1) \wedge K(S''',s) \wedge K(S'',s_1) \wedge F(\vec{x})[S''']. \tag{D.1}$$

Now observe that $\mathcal{M} \models Do^-(\delta,s,s_1)$ (from Lem. 6.2 and Prop. 6.3). By Lemma 6.1, there is a formula μ_i such that

$$\mathcal{M} \models \mu_i(s) \wedge s_1 = do([a_1,\ldots,a_n],s)$$

Now, we use Proposition 6.4 to conclude that:

$$\mathcal{M} \models \mu_i(s) \wedge \bigwedge_{i=1}^{n} \mathsf{SensedCond}(a_i,do([a_1,\ldots,a_{i-1}],s),do([a_1,\ldots,a_{i-1}],s')$$

Since $\mathcal{M}$ satisfies the SSA for K, and (D.1), we obtain that:

$$\mathcal{M} \models K(do(prim_\delta,S'''),do(prim_\delta,s)), \tag{D.2}$$

Finally, from Lemma D.3 we know that

$$\mathcal{M} \models Do(prim_\delta, S''', do([a_1, \ldots, a_n], S'''))),$$

and thus we can use part of the Proof for Theorem 6.3 to argue that also:

$$\mathcal{M} \models F(\vec{x}, do(prim_\delta, S''')). \tag{D.3}$$

($\Rightarrow$) follows from Eqs. D.2 and D.3.

($\Leftarrow$) Suppose that for some situation S'',

$$\mathcal{M} \models Do_K(\delta, s, s_1) \wedge K(S'', do(prim_\delta, s)) \wedge F(\vec{x})[S'']$$

From the successor state axiom of K, for some S''',

$$\mathcal{M} \models Do_K(\delta, s, s_1) \wedge K(do(prim_\delta, S'''), do(prim_\delta, s)) \wedge K(S''', s) \wedge F(\vec{x})[do(prim_\delta, S''')].$$

Since $\mathcal{M}$ satisfies the SSA for K, we have that

$$\mathcal{M} \models \mu_i(s) \wedge \bigwedge_{i=1}^{n} \mathsf{SensedCond}(A_i, do([A_1, \ldots, A_{i-1}], s), do([A_1, \ldots, A_{i-1}], s))$$

For some μ_i of Lemma 6.1, and some sequence of actions $A_1, \ldots, A_n$.

In addition, since the program is deterministic, we conclude that $s_1 = do([A_1, \ldots, A_{i-1}], s)$. Now, by using Proposition 6.4 and the fact that $\mathcal{M} \models K(S''', s)$ we obtain:

$$\mathcal{M} \models K(do([A_1, \ldots, A_{i-1}], S'''), do([A_1, \ldots, A_{i-1}], s)) \tag{D.4}$$

Now, the proof follows with an argument similar to that of Theorem 6.3. Since $\mathcal{M} \models F(\vec{x})[do(prim_\delta, S''')]$, then

$$\mathcal{M} \models Do^-(\delta, S''', S_1) \wedge F(\vec{x}, S_1)$$

However, since the program is deterministic, by Lemma we have that: $S_1 = do([A_1, \ldots, A_{i-1}], S''')$, and thus,

$$\mathcal{M} \models F(\vec{x}, do([A_1, \ldots, A_{i-1}], S''')) \tag{D.5}$$

($\Leftarrow$) follows from Eqs. D.4 and D.5. ■

Appendix E

Golog DCK for Experiments in Chapter 5

This section shows the Golog code utilized for generating the experimental results in Section 5.6. The code is written in Prolog syntax. Note that quantifiers and the `pi` construct receive *typed* variables (the type follows the variable in the declaration). Finally, `final_pred`$(\vec{c})$ is a new fact, added to the initial state at pre-processing time whenever `pred`$(\vec{c})$ is part of the goal.

E.1 Golog Control for The Trucks Domain

```
proc( trucks_control04,
      star(

            pi(current_location,location,
               [

                % get the current location
                ?(at(truck1,current_location)),


                % unload everything

                while( exists(area,truckarea,
                              exists(pack,package,
                                     in(pack,truck1,area))),
                       pi(area,truckarea,
                          pi(pack,package,
                             [
                              ?(in(pack,truck1,area)),
                             unload(pack,truck1,area,current_location)
                             ])
                         )
                     ),

                % deliver any thing you want
                while( exists(pack, package,
                              exists(loc, location,
                                     and( at(pack,loc),
```

```
                              final_location(pack, loc) ))),

          pi(pack,package,
             pi(loc,location,
                pi(t1,time,
                   pi(t2,time,
                      deliver(pack,loc,t1,t2)))))
       ),


                  % while there's a package here whose destination is
                  % elsewhere and there's space in the truck,
                  % load the truck with such a package


   while( and( exists(area,truckarea, free_(area,truck1)),
               exists(pack, package,
                      and(at(pack,current_location),
                          exists(loc, location,
                                 and(not(loc=current_location),
                                     final_location(pack, loc) ))))),

          pi(pack,package,
             pi(loc, location,
                [
                 ?(and(not(loc=current_location),
                       and(
                           at(pack,current_location),
                           final_location(pack, loc)))),
                 pi(area,truckarea,
                    load(pack,truck1,area,current_location))
                ]
               )
            )
       ),

   % if there is a package in the truck
   if(exists(pack,package,
             exists(area,truckarea,
                    in(pack,truck1,area))),

                   % then drive to its destination
      pi(pack,package,
              pi(area,truckarea,
                 [
                  ?(in(pack,truck1,area)),
                  pi(newloc,location,
                     [
                      ?(final_location(pack,newloc)),
                      pi(t1,time,
                         pi(t2,time,
                            drive(truck1,current_location,newloc,t1,t2)))
                     ])
                 ]
```

```
                        )
                    ),

                % else are there any packages not at its final destination?

                if(exists(loc1,location,
                          exists(pack,package,
                                 exists(loc2,location,
                                        and(at(pack,loc2),
                                            and(final_location(pack,loc1),
                                                not(loc1=loc2)))))),

                   % then drive where the truck is needed

                    pi(loc1,location,
                       pi(pack,package,
                          pi(loc2,location,
                             [
                              ?(and(at(pack,loc2),
                                    and(final_location(pack,loc1),
                                        not(loc1=loc2)))),
                              pi(t1,time,
                                 pi(t2,time,
                                    drive(truck1,current_location,loc2,t1,t2)))
                             ]
                            )
                         )
                      ),

                  % else stay here
                       [])
                )
             ]
            )
        )
    ).
```

E.2 Golog Control for The Storage Domain

```
proc(storage_control03,
     star(
             pi(cr,crate,
                pi(cs,storearea,
                   pi(d,depot,
                      [
                        % bind cr with a crate that should be (and is not at) depot d
                       ?(and(finally_in(cr,d),not(in(cr,d)))),
                        % bind cs with a store area inside some container
                       ?(exists(cont,container,
                                and(on(cr,cs),in(cs,cont)))),

                        % move to  (assume you are in a depot storage area)
```

```
                    if(not(exists(tr,transitarea,at(hoist0,tr))),
                       [
                        while(not(exists(cloc,storearea,
                                         and(at(hoist0,cloc),
                                             connected(cloc,loadarea)))),
                              pi(a1,storearea,
                                 pi(a2,storearea,
                                    move(hoist0,a1,a2)))),
                        % go out to the load area (if necessary)
                        pi(a1,storearea,go_out(hoist0,a1,loadarea))
                       ],
                       []),


                    pi(a,area,
                       pi(p,place,
                          lift(hoist0,cr,cs,a,p))), % lift the crate


                    pi(entry_point,storearea,
                       [
                        ?(and(connected(loadarea,entry_point),in(entry_point,d))),
                        if(and(clear(entry_point),
                               exists(free_store,storearea,
                                      and(connected(entry_point,free_store),
                                          clear(free_store)))),
                           [
                            go_in(hoist0,loadarea,entry_point),
                            star(pi(a1,storearea,
                                    pi(a2,storearea,
                                       move(hoist0,a1,a2))))
                           ],
                             % get into depot d
                           []
                          )
                       ]),

                    pi(sa,storearea,
                       pi(a,area,
                          drop(hoist0,cr,sa,a,d)))
                   ]
                  )))))).
```

E.3 Golog Control for The Rovers Domain

```
proc(rovers_control01,
     [
      while(exists(w,waypoint,and(finally_communicated_soil_data(w),
                                  not(communicated_soil_data(w)))),

            pi(soil_waypoint,waypoint,
               pi(r,rover,
```

```
            [
             ?(and(finally_communicated_soil_data(soil_waypoint),
                   not(communicated_soil_data(soil_waypoint)))),
             ?(equipped_for_soil_analysis(r)),

             % navigate until we get to the waypoint were the soil is
             while(not(at(r,soil_waypoint)),
                   pi(w1,waypoint,
                      pi(w2,waypoint,
                         navigate(r,w1,w2)))),
             pi(s,store,                       % take a soil sample
                [sample_soil(r,s,soil_waypoint),


                 star(pi(w1,waypoint, % navigate for a while
                         pi(w2,waypoint,
                            navigate(r,w1,w2)))),

                 pi(w1,waypoint, % communicate the data
                    pi(w2,waypoint,
                       pi(l,lander,
                          communicate_soil_data(r,l,soil_waypoint,w1,w2)))),
                 drop(r,s)       % drop the contents of the store
                ]
               )
            ]
           )
       )
     ),


while(exists(w,waypoint,and(finally_communicated_rock_data(w),
                            not(communicated_rock_data(w)))),

      pi(rock_waypoint,waypoint,
         pi(r,rover,
            [
             ?(and(finally_communicated_rock_data(rock_waypoint),
                   not(communicated_rock_data(rock_waypoint)))),
             ?(equipped_for_rock_analysis(r)),

             % navigate until we get to the waypoint were the rock is
             while(not(at(r,rock_waypoint)),
                   pi(w1,waypoint,
                      pi(w2,waypoint,
                         navigate(r,w1,w2)))),
             pi(s,store,                       % take a rock sample
                [sample_rock(r,s,rock_waypoint),


                 star(pi(w1,waypoint, % navigate for a while
                         pi(w2,waypoint,
                            navigate(r,w1,w2)))),
```

```
                    pi(w1,waypoint, % communicate the data
                       pi(w2,waypoint,
                          pi(l,lander,
                             communicate_rock_data(r,l,rock_waypoint,w1,w2)))),
                    drop(r,s)       % drop the contents of the store
                   ]
                  )
               ]
              )
           )
        ),


   while(exists(resolution,mode,
                exists(obj,objective,
                       and(finally_communicated_image_data(obj,resolution),
                           not(communicated_image_data(obj,resolution))))),
         pi(target_objective,objective,
            pi(target_resolution,mode,
               pi(r,rover,
                  pi(cam,camera,
                     [   % bind target_object and target_resolution
                      ?(and(finally_communicated_image_data(target_objective,
                                                            target_resolution),
                            not(communicated_image_data(target_objective,
                                                        target_resolution)))),

                      ?(and(equipped_for_imaging(r),
                            and(on_board(cam,r),
                                supports(cam,target_resolution)))),

                      % move rover to the calibration target

                      while(not(exists(o,objective,
                                       exists(w,waypoint,
                                              and(at(r,w),
                                                  and(visible_from(o,w),
                                                      calibration_target(cam,o)))))),
                            pi(w1,waypoint,
                               pi(w2,waypoint,
                                  navigate(r,w1,w2)))),

                      pi(obj,objective,
                         pi(w,waypoint,
                            calibrate(r,cam,obj,w))),

                      % move rover to a location where the objective is visible
                      while(not(exists(w,waypoint,
                                       and(at(r,w),
                                           visible_from(target_objective,w)))),
                            pi(w1,waypoint,
                               pi(w2,waypoint,
                                  navigate(r,w1,w2)))),
```

```
                    pi(wp,waypoint,              % take the image
                       take_image(r,wp,target_objective,cam,target_resolution)),


                    star(pi(w1,waypoint,         % navigate for a while
                            pi(w2,waypoint,
                               navigate(r,w1,w2)))),

                    pi(l,lander,                 % communicate image data
                       pi(w1,waypoint,
                          pi(w2,waypoint,
                             communicate_image_data(r,l,target_objective,
                                                    target_resolution,w1,w2))))
                ]
               )
            )
         )
      )
   ]
  ).
```