# Monitoring the Generation and Execution of Optimal Plans

by

Christian Fritz

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

# Abstract

Monitoring the Generation and Execution of Optimal Plans

Christian Fritz

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2009

In dynamic domains, the state of the world may change in unexpected ways during the generation or execution of plans. Regardless of the cause of such changes, they raise the question of whether they interfere with ongoing planning efforts. Unexpected changes during plan generation may invalidate the current planning effort, while discrepancies between expected and actual state of the world during execution may render the executing plan invalid or sub-optimal, with respect to previously identified planning objectives.

In this thesis we develop a general monitoring technique that can be used during both plan generation and plan execution to determine the relevance of unexpected changes and which supports recovery. This way, time intensive replanning from scratch in the new and unexpected state can often be avoided. The technique can be applied to a variety of objectives, including monitoring the optimality of plans, rather then just their validity. Intuitively, the technique operates in two steps: during planning the plan is annotated with additional information that is relevant to the achievement of the objective; then, when an unexpected change occurs, this information is used to determine the relevance of the discrepancy with respect to the objective.

We substantiate the claim of broad applicability of this relevance-based technique by developing four concrete applications: generating optimal plans despite frequent, unexpected changes to the initial state of the world, monitoring plan optimality during execution, monitoring the execution of near-optimal policies in stochastic domains, and

monitoring the generation and execution of plans with procedural hard constraints. In all cases, we use the formal notion of regression to identify what is relevant for achieving the objective. We prove the soundness of these concrete approaches and present empirical results demonstrating that in some contexts orders of magnitude speed-ups can be gained by our technique compared to replanning from scratch.

# Dedication

To my parents.

# Acknowledgements

First and foremost I want to thank my supervisor, Sheila McIlraith, who, from all possible perspectives, is the best supervisor I could imagine. This regards not only the scientific guidance she gave me, which was always characterized by a maximal degree of objectivity and rationality, but also the mentoring I have received from her. Sheila was always an active—and pro-active—supporter of my, and her other students', best interests, constantly looking out for and setting up new opportunities for us, both in research and in academic networking. I greatly appreciate her flexibility regarding the topics I wanted to work on, and her support and encouragement when pursuing my own ideas or collaborating with other students. It is due to what I have learned from her that I now feel ready to pursue independent research.

I thank Jorge Baier for the endless discussions, white board sessions, and collaborations. Jorge's analytical skills and his clear and calm way of pondering a problem have always been inspiring to me. Moreover, I greatly value his friendship.

Besides Jorge, I was fortunate to have a number of other collaborators whom I would like to thank for working with me: Meghyn Bienvenu, for our collaborations on planning with preferences; Richard Hull and Jianwen Su for giving me the chance to work with them at Bell Labs and beyond, and for broadening my horizon and triggering my interest in work-flow related issues.

For their extremely valuable feedback, comments, and suggestions I thank the members of my supervisory committee, Hector Levesque, Fahiem Bacchus, and Craig Boutilier. The presented thesis has greatly benefited from the thorough review and constructive criticism they have provided me with over the course of my program at the department. This also applies to my external examiner, Sven Koenig, who has done an extremely thorough and critical review of this thesis, providing a lot of useful comments and suggestions for future work.

I thank Gerhard Lakemeyer, who, as the supervisor of my Master's research in Aachen,

had given me the chance to apply my algorithms in the highly-dynamic RoboCup domain. This first-hand experience highlighted the need for and triggered my interest in the topic of this thesis. The chance to learn about the entire spectrum from sophisticated, state-of-the-art theory to pragmatic, assumptions-defying practice, has made a lasting impression on me and has always been a source of motivation for my research.

As a member of the Knowledge Representation Group, I have benefited numerous times from the feedback provided by its other members, and I thank them for attending and critiquing my practice talks and presentations of early stage ideas. I have always appreciated the friendly, constructive, yet critically objective atmosphere that has been characteristic for our group meetings. My gratitude also extends to the Department of Computer Science itself, for creating a stimulating environment and for hosting a long list of excellent and inspiring guest speakers.

Finally, I thank my girlfriend Tanya for her loving and understanding support in hard times.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

In artificial intelligence, the problem of classical *plan generation* is that of finding a sequence of actions that, upon execution, will transform a given initial state of the world into a state satisfying articulated goal conditions. In theory, if a model of the world, describing the way actions change the world, is given, one can solve this problem in a variety of ways, the most common of which involve some sort of bounded search through the set of all action sequences. The resulting plan could then be executed by any agent capable of executing the actions (e.g., a physical robot, or a human).

In practice, things can be quite different. In this thesis we study the problem of generating and executing optimal plans in highly-dynamic environments under real-time constraints. Such environments frequently change independent of the actions taken by the agent, and can often only be observed partially or through noisy sensors. This makes it hard, if not impossible, to create accurate and precise models of how the environment evolves over time and in particular in response to the actions executed by the agent. As a result, discrepancies between assumed and observed states of the world are frequent during both plan generation and plan execution. During plan generation, such discrep-

ancies mean that the planning effort thus far may have been invalidated. During plan execution, they bring into question whether the executing plan remains *valid* (i.e., projected to reach the goal) and where relevant, *optimal* with respect to a given measure of preference between possible plans.

To effectively monitor plan generation and execution, a system needs to continuously identify relevant discrepancies, and take ameliorative action when necessary. In many cases the ameliorative action is to replan starting in the current state, but often a significant amount of previous planning effort can be recovered by modifying existing planning data structures to reflect the new state.

Effective execution monitoring requires a system to quickly discern between cases where a detected discrepancy is relevant to the successful execution of a plan and those cases where it is not. Algorithms dating back as far as 1972 (e.g., PLANEX, the plan execution algorithm deployed on Shakey the Robot [Fikes *et al.*, 1972]) have exploited the idea of annotating plans with conditions that can be checked at execution time to confirm the continued *validity* of a sequential plan. In this thesis, we are interested in a variety of more difficult and unsolved monitoring tasks, including the monitoring of plan *optimality*, monitoring the generation of optimal plans in highly-dynamic environments, and monitoring the execution of policies, executing in stochastic domains. To this end, we generalize the existing approaches to monitoring plan validity, and formulate an abstract monitoring approach. From this abstract approach, we then derive a number of concrete implementations to these advanced monitoring problems. Effectively, we provide monitoring techniques to cover the complete life cycle of a plan, from the first moment of its generation process, to the point of complete execution.

The type of the objective plays a major role in the implementation of appropriate monitoring, and in practice users may have a variety of constraints the system needs to satisfy. These objectives often go beyond classical planning, and often involve temporally extended hard or soft constraints. System designers often wish to control their planning

agent (e.g., a robot) in more direct ways than through the simple statement of final state goals. Often it is desirable to prescribe or disallow certain types of behavior. Different approaches for achieving this exist in the literature, some of which are state-centric and often based on temporal logic (e.g., [Bacchus and Kabanza, 1998]), while others provide a procedural language for expressing such temporally extended constraints (e.g., [Levesque *et al.*, 1997]). These types of control constrain the search for a plan to only that subset of the space of all plans that conform to this control. When executing the generated plans, however, it is necessary to verify that these constraints continue to be met, when discrepancies occur. We provide a method that makes the developed suite of monitoring techniques applicable to this setting as well.

Our work is motivated in part by our practical experience with the fast-paced RoboCup[1] domain where teams of physical robots play soccer against each other. In RoboCup, the state of the world is typically observed 10 times per second, each time raising the question of whether to continue with the current plan or to replan. Verifying plan validity and optimality must be done quickly because of the rapidly changing environment. Currently, there are no techniques to distinguish between relevant and irrelevant discrepancies with respect to optimality, and so replanning is frequently done unnecessarily or discrepancies are ignored altogether, ultimately resulting in plan failure or sub-optimal performance.

In the widest sense, the described problems regard planning under uncertainty. The common approach for dealing with uncertainty in the literature is to create conditional plans that cover all possible contingencies (cf. Section 8.2). Contingency planning is time intensive, and its complexity generally grows with the number of contingencies. In particular in continuous domains the number of contingencies can be large or even infinite. We therefore argue that increasing the robustness of plans through effective monitoring is often a more viable alternative in these practically interesting domains. Particularly in situations where decisions need to be made in real time, considering all contingencies

[1]URL: `www.robocup.org`

may not be feasible. In many practical situations, the most effective approach may therefore be a combination of contingency planning and monitoring. We implement such an approach in Chapter 6.

In the next section we describe the larger framework in which we understand our algorithms to operate. This serves the purpose of both providing the necessary context and drawing connections to related problems that are not addressed in this thesis. Following that, we outline the structure of this document and enumerate our contributions.

## 1.2 Framework

We describe a framework for plan generation and plan execution including monitoring, and briefly compare it to previous framework specifications. In our framework, we distinguish two main components, one for plan generation and execution, and one for monitoring. This structure is depicted in Figure 1.1. The entities, all of which have

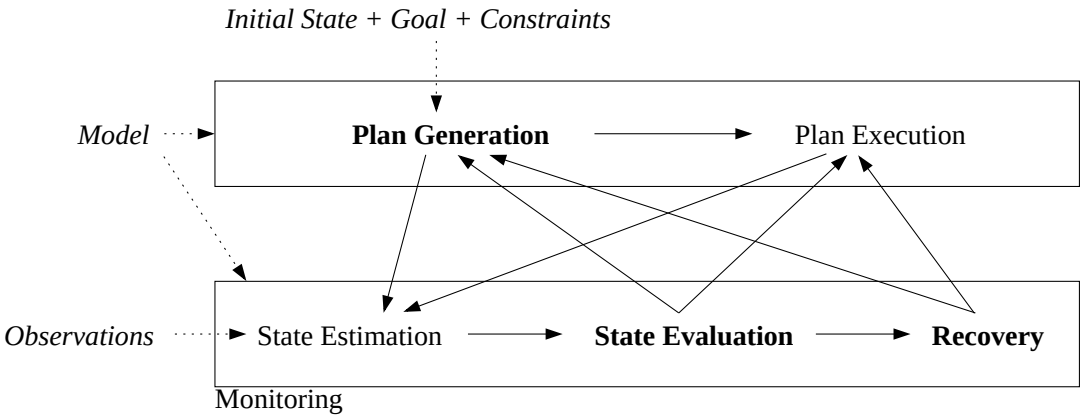



Figure 1.1: A framework for plan generation and plan execution including monitoring. Dotted arrows indicate data flow, solid arrows control flow, which may include data as well.

access to a model of the environment, are as follow:

**Plan Generation:** Given an initial state, a goal specification, and, optionally, additional soft and hard constraints, a planner generates a plan satisfying the objective, i.e., a plan that is expected to achieve the goal (valid), and which, if soft constraints (preferences) are given, is optimal with respect to these. The framework is agnostic about the precise planning paradigm, and, as we will see, the abstract monitoring approach we propose is applicable to a variety of paradigms and constraint specifications. In this thesis, we concentrate on forward search based planners. But, again, the proposed abstract approach is conceptually not limited to this plan generation technique.

**Plan Execution:** We assume that there is an agent capable of executing the actions prescribed by the plan. This agent could, for instance be a physical robot, a software bot acting over the semantic web, or a human. This thesis is not concerned with the precise details of how to execute each action, and we assume that the termination of actions is observable by the agent.

**State Estimation:** This is the task of estimating the actual state, based on the observation history and the model, possibly also determining a sequence of events that produced this state. We assume a situated agent, that is, an agent performing actions in a (dynamic) environment, thereby causing changes therein and receiving sensory input from this environment. This input can appear in response to active sensing requests or be delivered to the passive agent at a certain frequency, or upon the occurrence of certain events. Typically, these observations do not reveal the complete state of the world.

Which aspects of the current state one aims to estimate is determined by the subsequent steps, State Evaluation and Recovery. In the context of execution monitoring, state estimation is typically tailored towards detecting the discrepancies between what was predicted by the model and what was actually observed in the real world.

We do not directly contribute to this problem in this thesis, but for completeness provide an overview of existing approaches that are relevant to the problem of state estimation in Section 8.3.

**State Evaluation:** When executing a plan, this is the task of deciding whether the current state of the world has changed in a way that prevents the executing plan from achieving its objective (for instance, reaching the goal in an optimal way), or whether some sort of replanning is necessary. Similarly, during plan generation, the task is to decide whether the current planning effort remains valid in the currently estimated state of the world, or whether it needs to be revised.

The criteria for these decisions may vary. While in the literature almost exclusively the continued validity of the current plan was evaluated in the current state, it is often also desirable to decide whether the current plan is still optimal, or continues to adhere to additional constraints. This step effectively evaluates the relevance of any discrepancy noted between the expected state of the world and the one determined through state estimation. In the artificial example domains found in the planning literature, generally only those features of the world that matter to the problem are modeled. In these cases virtually any discrepancy between the expected and actual state of the world matters, i.e., affects the plan's ability to meet the objectives. But this is not so for many real-world domains. The reason is two fold: Since modeling action outcomes precisely can be very difficult and many state variables are real valued, it is often the case that expectations and observations only differ slightly and in particular are qualitatively the same. Secondly, we often face a lot of unpredictable exogenous events, most of which are unrelated to the problem the agent is facing and thus do not matter. But since still some of these may actually matter in certain situations, we cannot leave these details out of consideration by pruning them from the model entirely.

**Recovery:** Recovery is an alternative to replanning from scratch. It aims to modify and reuse the results of previous planning efforts, in order to minimize the time spent in devising a new plan that meets the objective. The questions involved here are how this can be performed as quickly as possible and what the constraints are. Many existing systems aim at repairing a failed plan quickly, ignoring the quality of the plan and following the intuition that plan-repair is faster than replanning from scratch. The adaptive plan generation method we propose generalizes from this problem, guarantees the produced plans to be optimal rather than just valid, and hence contributes to this problem as a special case as well. In the related work chapter we further review existing approaches to recovery.

In this thesis, we focus on the Plan Generation, State Evaluation, and Recovery steps, and their intimate relationships.

### 1.2.1 Comparison To Existing Frameworks

The monitoring component of the presented framework is similar and indeed inspired by that of [Bjäreland, 2001]. In our framework Observations replace Bjäreland's "Situation Assessment" function, and we replace his "Discrepancy Detection" and "Discrepancy Classification" steps into one State Estimation step. Bjäreland's approach is driven by the notion of a discrepancy between what was expected and what was observed. However, we believe that it is not always necessary to determine the exact discrepancy and also that often a significant amount of state estimation has to take place to determine it precisely. Our approach seems in particular in probabilistic settings more appropriate where a discrepancy may not be well defined. As a major difference, Bjäreland's framework lacks the State Evaluation step, a step we consider crucial for the overall success and performance of the system.

In [De Giacomo *et al.*, 1998] the authors propose a framework formalized in the situation calculus for monitoring the execution of high-level robot programs specified in

the Golog agent programming language [Levesque *et al.*, 1997] (also cf. Section 7.2). In this framework the authors assume to observe exogenous actions themselves, hence not addressing the problem of state estimation. The monitor is defined through a predicate $Monitor(\delta, s, \delta', s')$ and invoked after every step of the program. Intuitively, given the currently executing program $\delta$ and currently assumed situation $s$, $Monitor$ outputs a new situation term $s'$ to reflect the potential occurrence of an exogenous action, and modifies the program $\delta$ as appears adequate: If another predicate $Relevant(\delta, s, s')$ holds, a recovery predicate is called which may modify the remaining program $\delta$ to recover from the discrepancy. Otherwise the program remains unchanged. While the focus of [De Giacomo *et al.*, 1998] is on the formal specification of the framework, the authors do propose a specific, but simple, monitor. The *Relevant* predicate, corresponding to the State Evaluation step of our framework, is realized by simulating the remaining program $\delta$ in the new situation $s'$. If this simulation succeeds, that is the program can successfully be executed in the new situation, the discrepancy is irrelevant, otherwise it is relevant. Recovery is defined as finding a shortest repair program $p$ (sometimes called a *patch* (cf. [Eiter *et al.*, 2004])) for $\delta$, such that the sequence of the two programs, $p; \delta$, is expected to successfully execute as verified by simulation (forward projection). This suggests that all discrepancies are malignant.

Finally, [Fichtner *et al.*, 2003] formalized a similar framework based on the fluent calculus [Thielscher, 1998] and implemented it on a robot using the fluent calculus implementation FLUX [Thielscher, 2005].

Since the focus of this thesis is not on frameworks for formalizing the problem of execution monitoring, but on methods for addressing it, we do not consider the advantages and limitations of these frameworks in more detail.

## 1.3 Outline and Contributions

We employ Reiter's situation calculus [Reiter, 2001] to formalize our approach. Chapter 2 provides a review. Nevertheless, the majority of approaches developed in this thesis is applicable to any action language for which regression can be defined.

We begin the technical part of this document by reviewing existing systems for monitoring plan validity found in the literature, and formally characterizing a common technique deployed by these systems as goal regression (Chapter 3). We then generalize this technique into a broadly applicable abstract approach for monitoring that can be used to not only monitor plan validity, but plan optimality as well, forming the thesis of this document. Intuitively, the approach prescribes to first identify what is relevant to the achievement of the planning objective and annotate the plan with this information. Then, when a discrepancy between the expected and the actual state of the world occurs, this information can be used to determine the discrepancy's relevance very efficiently, and support recovery if needed. In subsequent chapters we substantiate this thesis by developing four concrete instances of this abstract approach for a list of advanced monitoring problems beyond the monitoring of basic plan validity. We describe these in more detail in the sub-section to follow. While in each of these chapters, we review previous work that directly relates to the problem addressed in that chapter, we describe a list of work that is more abstractly related to the general monitoring problem in Chapter 8. We conclude in Chapter 9 and identify possible directions for future work.

### 1.3.1 Contributions

**Monitoring plan optimality during execution (Chapter 4).**

First, we apply the approach to the notion of plan optimality and propose an algorithm to monitor plan optimality during execution that is sound but incomplete with respect to our specification. Prior to execution time we annotate each step

of our optimal plan by sufficient conditions for the optimality of the plan. These conditions correspond to the regression of the evaluation function (cost + heuristic) used in planning over each alternative to the currently optimal plan. At execution time, when a discrepancy occurs, these conditions can be reevaluated much faster than replanning from scratch by exploiting knowledge about the specific discrepancy. This corresponds to the State Evaluation step of our framework (Figure 1.1, p. 4), and regards the decision between continuing execution, or returning to plan generation, in case the plan has become sub-optimal or invalid. We have implemented our algorithm and tested it on simulated execution failures in well-known planning domains. Experimental results yield an average speed-up in performance of two orders of magnitude over the alternative of replanning, clearly demonstrating the feasibility and benefit of the approach.

**Generating optimal plans in highly dynamic environments (Chapter 5).**

Second, we show that the approach can also be used to monitor the plan generation process itself, which is necessary since generating optimal plans in highly dynamic environments is challenging as well. Plans are predicated on an assumed initial state, but this state can change unexpectedly during plan generation, potentially invalidating the planning effort. We make three contributions: (1) We propose a novel algorithm for generating optimal plans in settings where frequent, unexpected events interfere with planning. By implementing the abstract approach of this thesis, it is able to quickly distinguish relevant from irrelevant state changes, and to update the existing planning search tree if necessary. This corresponds to the Recovery step of the framework of Figure 1.1. We prove the correctness of this approach. (2) We argue for a new criterion for evaluating plan adaptation techniques: the *relative* running time compared to the amount of change. This is significant since during recovery more changes may occur that need to be recovered from subsequently, and in order for this process of repeated recovery to terminate,

recovery time has to *converge* to zero. (3) We show empirically that our approach can converge and find optimal plans in environments that would ordinarily defy planning because of their high dynamics.

**Monitoring policy execution in stochastic domains (Chapter 6).**

Third, we demonstrate the broad applicability of the approach, by providing the details of its use in decision-theoretic planning. In particular, we consider plan generation and execution in stochastic domains with (a) exogenous events, (b) an incorrect model, and/or (c) incomplete forward search (i.e., discarding low probability outcomes). We consider forward search-based planning algorithms for Markov Decision Processes (MDPs), exploring the reachable state space from a given initial state. Under such circumstances, an executing policy often finds itself in an unexpected state, bringing into question the continued optimality of the policy being executed (or near-optimality in the case where the optimal policy was approximated). Again, replanning in this unexpected state is a naive and costly solution, but is often unnecessary. Using our approach we identify the subset of each expected state that is critical to the optimality of a policy. With this information in hand, we can often avoid replanning when faced with an unexpected state. Our analysis offers theoretic bounds on optimality in certain cases as well as empirical results demonstrating significant computational savings compared to replanning from scratch.

**Generating and executing plans with procedural control (Chapter 7).**

Finally, we show how the approach can be applied to the problem of monitoring the continued satisfaction of procedural hard constraints during plan execution as well.

ConGolog is a logical agent programming language and is defined in the situation calculus. ConGolog agent control programs were originally proposed as an alterna-

tive to planning, but have also more recently been proposed as a means of providing domain control knowledge for planning. Unfortunately, in order to reason about the satisfaction of the hard constraints imposed by a ConGolog program, a system requires special-purpose machinery. This not only prevents state-of-the-art planners to easily use search control expressed in this language, but also hinders the direct application of the abstract monitoring approach of this thesis, in order to monitor the satisfaction of these constraints during execution.

We present a possible solution to these shortcomings. We present a compiler that takes a ConGolog program and produces a new basic action theory of the situation calculus whose executable situations are all and only those that are permitted by the program. This compilation has several theoretical and practical merits, and in particular allows systems to reason about ConGolog programs without any additional reasoning machinery. This includes the abstract monitoring approach of this thesis. In terms of the framework of Figure 1.1, the compilation generates the proper input to the planner (model, initial state, goal state), expressed in a basic action theory. Using this action theory as input, one can immediately apply all monitoring approaches proposed above. In particular, it is possible to generate plans that satisfy the procedural hard constraints expressed by the program, and monitor their continued satisfaction when discrepancies between expected and actual state of the world occur during execution.

# Chapter 2

# Background

## 2.1 The Situation Calculus

The situation calculus is a family of many-sorted logical languages for specifying and reasoning about dynamical systems. It was first proposed by McCarthy [1963] and later significantly extended by Reiter [2001], most importantly by providing a solution to the frame problem (see below). In this thesis we use Reiter's situation calculus.

Its basic elements are situations, primitive actions (sort $\mathcal{A}$), and fluents (sort $\mathcal{F}$). A situation is a *history* of the primitive actions performed from a distinguished initial situation $S_0$. The function $do(a, s)$ denotes the situation resulting from performing action $a$ in situation $s$, inducing a tree of situations rooted in $S_0$. Fluents are relations and functions that take a situation as their last argument (e.g., $F(\vec{x}, s)$), and are used to define the state of the world.

For readability, action and fluent arguments are generally suppressed. Also, $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$ is abbreviated to $do([a_1, \ldots, a_n], s)$ or $do(\vec{a}, s)$ and we define: $do([\,], s) \stackrel{\text{def}}{=} s$.

### 2.1.1 Basic Action Theories

A *basic action theory* in the situation calculus, $\mathcal{D}$, is comprised of the following sets of axioms [Levesque *et al.*, 1998]:

- $\Sigma$ the set of domain independent foundational axioms of the situation calculus, including one second-order induction axiom required to properly define the tree of situations. These axioms are as follows:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2,$$
$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s),$$
$$\neg s \sqsubset S_0,$$
$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'.$$

  Here the relation $\sqsubset$ provides an ordering on situations, and $s \sqsubseteq s'$ abbreviates $s = s' \vee s \sqsubset s'$.

- $\mathcal{D}_{ss}$, *successor state axioms*, provide a parsimonious representation of frame and effect axioms under an assumption of the completeness of the axiomatization. There is one successor state axiom for each fluent, $F \in \mathcal{F}$, of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $\vec{x}, a, s$. $\Phi_F(\vec{x}, a, s)$ characterizes the truth value of the fluent $F(\vec{x})$ in the situation $do(a, s)$ in terms of what is true in situation $s$. These axioms can be automatically generated from effect axioms, as described below.

- $\mathcal{D}_{ap}$, *action precondition axioms*, first-order axioms that specify the conditions under which actions are possible. There is one axiom for each action $a \in \mathcal{A}$ of the form $Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$ where $\Pi_a(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$.

- $\mathcal{D}_{una}$, a set of *unique name axioms* for actions;

- $\mathcal{D}_{S_0}$ a set of sentences relativized to situation $S_0$, specifying what is true in the initial state.

Although any situation calculus action theory is second-order, many reasoning tasks can be reduced to first-order theorem proving by using regression [Reiter, 2001] (also cf. Section 2.2). Properties that hold in all executable situations can be shown by induction over situations [Reiter, 1993].

### 2.1.2 The Frame Problem and A Solution for Deterministic Actions

To the user, specifying the effects of actions can be more natural when using *effect axioms*. For a relational fluent $F(\vec{x}, s)$, for example, positive and negative effect axioms can define the conditions under which the fluent becomes true ($\phi^+(\vec{x}, s)$), respectively false ($\phi^-(\vec{x}, s)$), after performing action $a$:

$$\phi^+(\vec{x}, s) \supset F(\vec{x}, do(a, s)),$$
$$\phi^-(\vec{x}, s) \supset \neg F(\vec{x}, do(a, s)).$$

These axioms describe the effects on the considered fluents, but they do not describe the *non-effects* on all other fluents. Axioms describing the latter are called *frame axioms*. The *frame problem* states the impossibility of stating and reasoning with all frame axioms explicitly, due to their cardinality: Even apparently nonsensical assertions, like "drinking water does not change one's hair color" would have to be captured by a frame axiom:

$$haircolor(do(drinkwater, s)) = y \leftarrow haircolor(s) = y.$$

Ray Reiter [Reiter, 1991] proposed a solution to the frame problem based on a completeness assumption, namely that the provided effect axioms specify *all* possible ways by which a fluent may change. In Reiter's solution, the set of all effect axioms, is hence syntactically transformed into the set of successors state axioms (one for each fluent).

In the following we describe how Reiter's solution applies to *functional fluents*, for relational fluents the computations are similar and can be found in [Reiter, 1991]. The effect axiom for a functional fluent $f$ and action $\alpha$ has the form:

$$\phi_f(\vec{t}, y, s) \supset f(\vec{t}, do(\alpha, s)) = y$$

where $\vec{t}$ are terms not mentioning situation terms. Note that, unlike relation fluents, for functional fluents there are no positive and negative effect axioms but only one axiom explicitly stating the new value ($y$) of the fluent. Above formula can be rewritten to:

$$\underbrace{a = \alpha \wedge \vec{x} = \vec{t} \wedge \phi_f(\vec{x}, y, s)}_{\Phi_f} \supset f(\vec{x}, do(a, s)) = y$$

and this can be done for all $n$ effect axioms for fluent $f$. These axioms can then be merged into a single *normal form* for this fluent:

$$\begin{aligned} \Phi_f^{(1)} \vee \cdots \vee \Phi_f^{(n)} &\supset f(\vec{x}, do(a, s)) = y, \qquad \text{or} \\ \gamma_f(\vec{x}, y, a, s) &\supset f(\vec{x}, do(a, s)) = y \end{aligned} \tag{2.1}$$

The completeness assumption expresses that if fluent $f$ changes its value from situation $s$ to situation $do(a, s)$, then $\phi_f(\vec{x}, y, a, s)$ must be true:

$$f(\vec{x}, s) = y' \wedge f(\vec{x}, do(a, s)) = y \wedge y \neq y' \supset \gamma_f(\vec{x}, y, a, s) \tag{2.2}$$

Together with the assumption

$$\neg(\exists \vec{x}, y, y', a, s).\gamma_f(\vec{x}, y, a, s) \wedge \gamma_f(\vec{x}, y', a, s) \wedge y \neq y'$$

Reiter shows that (2.1) and (2.2) are logically equivalent to:

$$\begin{aligned} f(\vec{x}, do(a, s)) = y \equiv\ & \gamma_f(\vec{x}, y, a, s) \vee \\ & f(\vec{x}, s) = y \ \wedge (\nexists y').\gamma_f(\vec{x}, y', a, s) \wedge y \neq y' \end{aligned} \tag{2.3}$$

which is the successor state axiom for functional fluent $f$.

## 2.2 Regression

The *regression* of a formula $\psi$ through an action $\alpha$ is a formula $\psi'$ that holds prior to $\alpha$ being performed if and only if $\psi$ holds after $\alpha$ is performed. Regression was first defined by Waldinger [Waldinger, 1977]. In the situation calculus, one step regression is defined inductively using the successor state axiom for a fluent $F(\vec{x}, s)$ as above:

$$Regr[F(\vec{x}, do(a, s))] \stackrel{\text{def}}{=} \Phi_F(\vec{x}, a, s)$$
$$Regr[\neg\psi] \stackrel{\text{def}}{=} \neg Regr[\psi]$$
$$Regr[\psi_1 \wedge \psi_2] \stackrel{\text{def}}{=} Regr[\psi_1] \wedge Regr[\psi_2]$$
$$Regr[(\exists x)\psi] \stackrel{\text{def}}{=} (\exists x) Regr[\psi]$$

We use $\mathcal{R}[\psi, \alpha]$ to denote $Regr[\psi(do(\alpha, s))]$, and $\mathcal{R}[\psi, \vec{\alpha}]$ to denote the repeated regression over all actions in the sequence $\vec{\alpha} = [\alpha_1, \ldots, \alpha_n]$ in reverse order, i.e.,

$$\mathcal{R}[\psi, \vec{\alpha}] \stackrel{\text{def}}{=} Regr\big(\ldots Regr\big(Regr\big(\psi(do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_1, s))))\big)\big)\big).$$

Note that the formula resulting from the regression has a free variable $s$ of sort situation. Hence, we can write $\mathcal{R}[\psi, \vec{\alpha}](S)$ to denote its instantiation in a particular situation $S$. Intuitively, the result of the regression is the condition that has to hold in $s$ in order for $\psi$ to hold after executing $\vec{\alpha}$ (i.e., in $do(\vec{\alpha}, s)$). It is predominantly comprised of the fluents occurring in the conditional effects of the actions in $\vec{\alpha}$. Due to the Regression Theorem [Reiter, 2001, pp.65–66] we have that $\mathcal{D} \models \psi(do(\vec{\alpha}, s)) \equiv \mathcal{R}[\psi, \vec{\alpha}]$ for all situations $s$.

As an example of regression involving functional fluents, consider a formula stating "at Union Square and cash = \$10". Regressing this formula over an action sequence ["take subway to Times Square", "buy ice cream"] yields a condition "cash = \$15", when a subway ride costs \$2 and an ice cream \$3, say.

Regression is a purely syntactic operation. Nevertheless, it is often beneficial to simplify the resulting formula for later evaluation. Regression can be defined in many action specification languages. In STRIPS [Fikes and Nilsson, 1971], regression is very

simply achieved by systematically adding and removing propositions from a set: the regression of a set of literals $L$ over an action $\alpha$ is defined based on the add and delete lists of $\alpha$:

$$\mathcal{R}^{\text{STRIPS}}[L, \alpha] = \begin{cases} \text{False}, & \text{if } L \cap \textsc{Del}(\alpha) \neq \emptyset; \\ L \setminus \textsc{Add}(\alpha), & \text{otherwise.} \end{cases}$$

Regression in ADL was defined in [Pednault, 1989]. ADL is also the foundation for the Planning Domain Definition Language (PDDL) [McDermott, 1998], which is used as the specification language in the biennial International Planning Competition. Rintanen [2008] defined a regression operator for ground PDDL operators.

In general, the regression of a formula $\psi$ over a sequence of actions $\vec{\alpha}$ may result in a formula whose size is exponential in the length of $\vec{\alpha}$. However, under certain restrictions, this blow up can be prevented. Reiter [2001], p.74, proposes one such restriction, which requires successor state axioms to be *context-free*. A successor state axiom for a fluent $F$ is context-free, roughly, if and only if the truth value of $F$ in $do(a, s)$ depends only on the truth value of $F$ in $s$ and is independent of other fluents' truth values. For context-free successor state axioms, regression has at most linear complexity in the length of the action sequence. As a special case, this restriction is satisfied by theories whose actions only have unconditional effects, as it is the case in STRIPS. In fact, regression in STRIPS is even simpler, since, due to the lack of conditional effects, the result of regressing a literal $l$ over some action can only be either *true*, *false*, or $l$ itself.

The specification of other restrictions is subject to ongoing research. Recently, van Ditmarsch *et al.* [2007] showed that, in the propositional case, formulae can be regressed with only a polynomial blow-up if one is willing to introduce new atoms to replace certain sub-formulae. Furthermore, regression in PDDL causes at most polynomial increase in size when the formula is represented in circuit form [Rintanen, 2008].

## 2.3 Notation and Definitions

Throughout this thesis we will use the following notational conventions and auxiliary definitions.

Lower case letters denote variables in the theory of the situation calculus, upper case letters denote constants. We use $\alpha$ to denote arbitrary but explicit actions and $S$ to denote arbitrary but explicit situations, that is $S = do(\vec{\alpha}, S_0)$ for some explicit action sequence $\vec{\alpha}$. Variables that appear free are implicitly universally quantified unless stated otherwise. By $\psi[x/y]$ we denote the formula resulting from substituting all occurrences of $x$ in $\psi$ with $y$. Further, $\vec{a} \cdot a$ denotes the result of appending action $a$ to the sequence $\vec{a}$.

For convenience, we define the following shortcut for talking about the preconditions of an action sequence $\vec{a} = [a_1, \ldots, a_n]$:

$$Poss([a_1, \ldots, a_n], s) \stackrel{\text{def}}{=}$$
$$Poss(a_1, s) \wedge Poss\big(a_2, do(a_1, s)\big) \wedge \cdots \wedge Poss\big(a_n, do([a_1, \ldots, a_{n-1}], s)\big)$$

For two situations $s, s'$, such that $\Sigma \models s \sqsubseteq s'$, we say that $s$ is a *sub-history* of $s'$, or $s'$ is a *continuation* of $s$.

We say that a situation $s$ is *executable*, denoted as *executable*$(s)$, if all actions in the history of $s$ have their preconditions satisfied in the situation where they are performed, formally:

$$executable(s) \stackrel{\text{def}}{=} (\forall a, s').do(a, s') \sqsubseteq s \supset Poss(a, s').$$

We use *fluents*$(\psi)$ to denote the set of all fluents occurring in formula $\psi$.

For convenience, for any $s, s'$ such that $\Sigma \models s \sqsubset s'$ we use $\mathcal{R}[F(\vec{x}, s'), s]$ to denote the regression of fluent $F$ back to $s$, i.e.,

$$\mathcal{R}\big[F(\vec{x}, do(\vec{\alpha}, s)), s\big] \stackrel{\text{def}}{=} \mathcal{R}\big[F(\vec{x}), \vec{\alpha}\big],$$

and we extend this to general formulae, possibly containing different situation terms:

$$\mathcal{R}[\neg\psi, s] \stackrel{\text{def}}{=} \neg\mathcal{R}[\psi, s]$$
$$\mathcal{R}[\psi_1 \wedge \psi_2, s] \stackrel{\text{def}}{=} \mathcal{R}[\psi_1, s] \wedge \mathcal{R}[\psi_2, s]$$
$$\mathcal{R}[(\exists x)\psi, s] \stackrel{\text{def}}{=} (\exists x)\mathcal{R}[\psi, s].$$

If $s$ is a free variable, then the resulting formula is again free in $s$. For convenience we also define $\mathcal{R}[\psi(s), s] \stackrel{\text{def}}{=} \psi(s)$.

As an example, consider the formula $F_1(do([\alpha_1, \alpha_2], s)) \wedge F_2(do(\alpha_1, s))$ and its regression back to $s$:

$$\begin{aligned}
&\mathcal{R}\big[F_1(do([\alpha_1, \alpha_2], s)) \wedge F_2(do(\alpha_1, s)), s\big] = \\
&\quad = \mathcal{R}\big[F_1, [\alpha_1, \alpha_2]\big] \wedge \mathcal{R}\big[F_2, [\alpha_1]\big] \\
&\quad = \mathit{Regr}\Big[\mathit{Regr}\big[F_1(do([\alpha_1, \alpha_2], s))\big]\Big] \wedge \mathit{Regr}\big[F_2(do(\alpha_1, s))\big].
\end{aligned}$$

# Chapter 3

# Monitoring Plan Validity

## 3.1 Introduction

The contributions of this chapter are threefold. We review a list of influential existing approaches for execution monitoring, whose commonalities we then generalize into an abstract monitoring approach that is more broadly applicable. We then describe a first concrete implementation of this abstract approach in the situation calculus, for monitoring plan validity during execution. This realizes the State Evaluation step of the framework of Section 1.2, for the case of plan validity.

A number of existing approaches for execution monitoring have proposed to annotate the plan with additional information during plan generation, which can be used during execution to determine the continued validity of the plan in case of unexpected events. The intuitive idea is to determine what is relevant for the continued validity of a plan at each step of plan execution, and annotate the plan with this information accordingly. We formally characterize the technique implicitly taken by these approaches for identifying this information as goal regression. We then generalize the approach into a more general, abstract monitoring approach, forming the thesis of this document. This monitoring approach, which continues to be based on relevance, allows for a broad variety of possible

objectives, including that of plan optimality, and can be applied during plan generation itself as well, rather than during plan execution only.

The approach for monitoring plan validity in the situation calculus, which we formalize at the end of this chapter, is a first example of a concrete implementation of this abstract approach. This formalization makes the approach usable by a large variety of planners, instead of being planner specific. It also allows us to prove the correctness of the approach, which has not been done in the reviewed literature inspiring the approach. Last but not least, it assists in understanding the more advanced uses of the developed abstract monitoring approach to more complex objectives, presented in the following chapters of this thesis.

## 3.2 Characterizing Existing Approaches

In this section we review significant previous approaches to execution monitoring, which will provide the motivation for the abstract approach we develop. We first review a number of early approaches towards the integration of planning and execution, which, treat the problem of execution monitoring and the sub-problem of state evaluation we are most interested in, rather implicitly. Even though implicit, these works are suggestive of a useful starting point towards the development of appropriate formal methods for state evaluation and replanning. We then discuss a few approaches that exhibit a more explicit treatment and awareness of this problem, but which are lacking concrete suggestions regarding its solution. Finally, we generalize the distinguished advantages of the reviewed works into a general abstract approach. This approach serves as the guiding principle for the methods we develop in the remainder of this thesis, which can be viewed as concrete implementations of this abstract approach to solve different problems of interest.

### 3.2.1 Integrated Planning and Execution

Work on execution monitoring reaches back to the earliest deployed systems. As one of the first Fikes *et al.* [1972] described, among other things, the execution system applied on Shakey the Robot. Even though the language used for planning, STRIPS, is very limited in expressiveness, the authors presented some interesting ideas for monitoring the execution of plans and reacting to execution failure. The core idea is to represent plans, which are limited to be sequential, in a way that supports monitoring by revealing what was termed the "plan structure". For this, the authors introduced *triangle tables*. Figure 3.1 shows a triangle table for the sequential plan $OP_1, OP_2, OP_3$, where $OP_i$ is a STRIPS operator. Each cell represents a set of clauses. The cells below operator

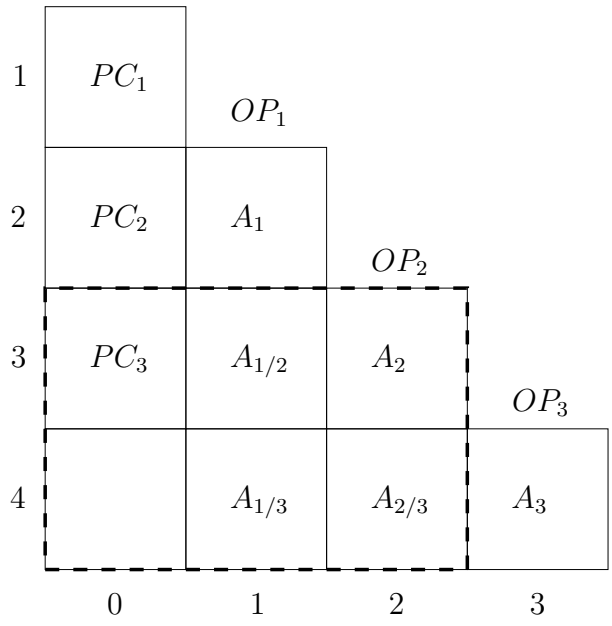



Figure 3.1: A triangle table for representing a plan with three operators $OP_1, OP_2, OP_3$, the dashed box defines the "3rd kernel".

$OP_i$, contain exactly those ADD effects of $OP_i$ that persist after the execution of the corresponding subsequent plan step. That is, $A_1$ contains exactly those clauses that are produced by $OP_1$ and $A_{1/2}$ contains that subset of $A_1$ whose clauses are not deleted by $OP_2$. The left-most column (number 0) holds those clauses that are preconditions to

the corresponding operators in the plan but are not produced by earlier operators in the plan and are thus required to be established by other means prior to execution. Taken together, these clauses ($PC_i$) define the overall preconditions for the entire plan. Further, Fikes et al. defined the *marked clauses* of a cell to be those clauses that are required by the operator of that row. Then the $i^{th}$*-kernel* is defined as the set of marked clauses in the rectangle that includes the bottom left cell and row $i$ (see the figure for an example). The interpretation of this is that whenever the clauses of the $i^{th}$-kernel are true in a given model of the state and dynamics of the environment, then the $i^{th}$-tail of the plan, starting with operator $i$, is applicable and projected to reach the goal. In our terminology we say the remaining plan is *valid*. Further, if the $(n+1)^{th}$-kernel is true, where $n$ is the number of operators in the plan, the goal is already achieved.

Based on the triangle table representation of a plan, the execution strategy that Fikes et al. proposed, called PLANEX, proceeds as follows:

1. Test the highest kernel $(n + 1)$.

2. If it is true, the goal is achieved, stop execution.

3. Otherwise, proceed checking lower kernels until one is found that is true and then execute the corresponding operator.

4. If no kernel in the triangle table is true, the plan has failed and replanning is required.

5. Otherwise, repeat.

By optimistically checking all higher kernels before executing the next operator in the plan, this strategy has the advantage that it discovers some serendipities. It also repeats parts of a plan that have failed to achieve their intended effects. However, this procedure may result in an infinite loop when the failure is due to a systematic, e.g., modeling,

error like for instance assuming that an action achieves a certain literal but simply does not, no matter how often the action is executed.

While the applied action description language, STRIPS, is fairly limited in its expressiveness, we make the following observation that allows us to lift the benefits of this approach to more expressive action description languages: Although the authors seem to have been unaware of this at the time of this article or they simply miss pointing it out, a kernel is exactly the regression of the goal (including action preconditions) through the remaining operators of the plan. Roughly, the regression of a formula over an action is a sufficient and necessary condition for the satisfaction of the formula following the execution of the action (cf. Section 2.2). The simplicity of the representation, merely based on clauses of the ADD effects of operators, is due to the limitation of STRIPS to non-negative, non-disjunctive preconditions and goals, and the required absence of conditional effects of actions. This regression is produced by the backward-chaining based planning algorithm deployed by the authors, and thus comes without computational overhead.

The utility of the plan structure for execution monitoring was later recognized by others as well. In these approaches the term *rationale* was coined, as the authors realized that not so much the structure of a plan but rather the rationale for the choice of its elements is what has to be preserved. This was the case with the hierarchical SIPE planning system [Wilkins, 1988] and the IPEM system [Ambros-Ingerson and Steel, 1988] based on the TWEAK partial order planner. The execution monitoring strategy of the former is described in [Wilkins, 1985]. The core idea is to represent the purpose of every action in the plan by stating the time until when its effects have to be maintained, e.g., until the goal is reached or a depending action has been executed, and to explicitly mark assumptions made during planning. These annotations are used during execution to decide whether a discrepancy is affecting the remaining plan or not. If so, SIPE offers a set of eight replanning rules that "often retain much of the original plan". These rules are incomplete in the sense that they do not necessarily produce a legal plan but rather a

task network with some unachieved sub-goals remaining, in which case the transformed plan is handed back to the planner.

In many applications, it is desirable to conservatively, i.e., only minimally, modify the current plan when needed, as this is generally assumed to minimize execution costs. An example of this is described by Myers [1998] where the SIPE approach was used to monitor the execution of air campaigns. We, however, believe that, if there are costs involved, these should be modeled explicitly and the monitoring be extended appropriately to monitor continued optimality of the plan according to these costs as opposed to only monitoring continued validity and soft-constraining the replanning implicitly through an unspecified preference criterion as Myers does.

Similarly to SIPE, Ambros-Ingerson and Steel [1988] approached execution monitoring in conjunction with the TWEAK partial order planner. They extend the set of plan transformation rules of the planner to accommodate for unforeseen events and to integrate the execution into the process. The actual execution monitoring is then performed through the application of predefined IF–THEN rules mapping *flaws* to *fixes* (plan transformation rules). A scheduler heuristically sorts the list of open flaws continuously during execution. While the paper does not go into detail about how to efficiently detect flaws, which is a major limitation, this work is still interesting as it shows how partial-order planning can naturally perform plan repair upon unforeseen events during execution. This is due to the fact that partial-order planning, much like backward-chaining, chooses actions to add to the plan based on open preconditions and this again makes the reasons for adding the actions, the rationale, explicit and usable for monitoring (and plan repair).

While PLANEX merely used the plan's rationale for evaluating whether the current plan or parts of it are still valid for achieving the goal, SIPE and IPEM also exploit this information for replanning.

Others have approached the problem of recovering from discrepancies during execution heuristically using predefined fault models and fixes. Beetz and McDermott [1994] de-

scribed how XFRM, a planner based on RPL, the Reactive Plan Language, handles discrepancies. In this framework, the model of the dynamics is probabilistic, i.e., actions can have stochastic effects. When a discrepancy arises during the execution of a plan, sample execution scenarios are projected to evaluate the impact of the discrepancy. This contrasts with the earlier presented approaches which, one way or the other, all implicitly regressed the goal to evaluate the impact of the discrepancy. Here instead the discrepancy is progressed. Since the projection is non-deterministic, the number of sample scenarios determines the probability of detecting a problem, but the authors do not investigate how many samples are required depending on the situation and the domain, neither do they provide a concise definition of a failure. When failures are projected to happen, these are classified into a domain dependent hand-crafted taxonomy of fault models that also maps them to plan transformation rules. The application of these transformations may improve the plan, but is not guaranteed to do so. All in all the approach is incomplete in at least two senses: due to the choice of samples certain future problems which are a result of a discrepancy in the current state may remain undetected, and due to the heuristic character of both diagnosis and replanning (plan transformation), the approach is not guaranteed to recover from contingencies. The authors later extended the approach to improve flexibility between planning and execution [Beetz and McDermott, 1996], but this did not address the issues raised above.

### 3.2.2 Expectation-Based Monitoring

Doyle *et al.* [1986] were the first to explicitly state the idea of basing the monitoring decision (continue execution or replan) on the materialization of certain plan-dependent expectations. Doyle et al. proposed an approach to the problem of monitoring the successful execution of a plan through the generation and use of perception requests to verify the nominal execution of the involved actions. Given a plan and a model of the environment, their method selects properties that need to be monitored and generates

perception requests that can verify the materialization of these properties. The latter are embedded into the plan. The properties are typically pre- and post-conditions of actions, and perception requests are sensing actions together with the expected sensing values. During execution the actual sensing values returned by the sensing actions can be compared to the expected values to detect discrepancies. While the significance of the actual implemented system described in the paper, called GRIPE (Generator of Requests Involving Perception and Expectations), may be questionable, the paper is notable for raising the awareness of relevant questions. These involve how often and accurately assertions should be verified and how this can be done best, i.e., introducing the problem of planning for perception requests. But first and foremost the question as to which properties should be monitored is raised and this question today is still awaiting a satisfactory answer and to which we contribute in this thesis. The authors propose four criteria involved with this choice:

- the *uncertainty* of properties due to several reasons (incomplete initial knowledge, stochastic action outcomes, etc.);
- the *dependency* of future actions on the property;
- the *importance* of the property; and
- the *ease of recovery*.

Of these four, the second has received the most attention as it was addressed, for instance, implicitly already in PLANEX, SIPE and IPEM, as described above. Doyle et al. realized that some action effects are irrelevant as no future action nor the goal require their presence, while others, which are said to lie on the *critical path*, are relevant. But the paper lacks a formal definition of this set. The notion of a critical path relates this work to the triangle tables we saw earlier since "elements of the critical path" is merely a synonym for "marked clauses in a kernel". Doyle et al., like Fikes and his colleagues

earlier, did not acknowledge that this is again just the regression of the goal through the remaining actions of the plan. Characterizing this technique as goal regression allows us to generalize this technique to other action formalisms, planning paradigms, and preference criteria as we demonstrate in the remainder of this thesis.

Inspired by this work, Musliner et al. discussed some aspects of time in execution monitoring for actions with durations and in particular addressed the questions of how to verify critical assertions most effectively [Musliner *et al.*, 1991]. By using a planner that deploys simple depth-first backward-chaining from the goal to the initial state, they construct the critical path. Then, at every stage in the plan where a post-condition is established that is used in the precondition of a later action, a verification action is inserted to monitor the condition over the required time interval. Musliner et al. argue that these actions themselves may require planning as they have pre- and post-conditions, and durations as well. To accommodate this, they propose to augment the theory with models for these sensing actions. The paper however lacks formal details and leaves certain questions open, e.g., whether and how the models for the normal actions are modified to motivate the introduction of verification actions.

Another example of expectation-based monitoring is the work by Earl and Firby [1997]. Instead of planning from first principles, the presented "Routine Activity Management and Analysis" system (RAMA) chooses from predefined *dynamics* when given a goal to achieve. These dynamics combine actions and expectations about observations to be made during their execution.

Expectations were also the basis for the method we previously proposed for monitoring the execution of plans generated from Golog programs [Ferrein *et al.*, 2004]. Roughly, Golog is a programming language that combines explicit agent programming with planning, by allowing both deterministic as well as non-deterministic programming constructs (cf. Section 7.2 for a more detailed review of Golog). In this approach, the produced plan is explicitly annotated with the assumptions that were made during planning. These as-

sumptions represent the at–planning–time projected truth values of conditions included in the Golog program (e.g., in if–then–else constructs or while–loops). If, during execution of the plan, these assumptions are violated or the next action is not possible, the plan is abandoned and replanning from scratch is performed. This seems to be reasonable in a domain that is as dynamic as RoboCup, where we applied this work.[1] This contrasts with the earlier mentioned work of De Giacomo *et al.* [1998] as it does not monitor the execution of the Golog program itself, but the resulting (conditional) plan after performing decision-theoretic planning over a program, maintaining the expectations involved through the hard constraints in the original program. While the approach worked in practice, it cannot claim any kind of correctness or optimality. For instance, it does not anticipate failure of future actions in the plan, like others do. Nevertheless it serves as a rare example of monitoring where the constraints underlying the construction of the plan, which in this case were procedural hard constraints represented through a Golog program, are taken into account in monitoring the execution of the plan. This is crucial, because when the the satisfaction of these constraints change, there is no reason to believe that the plan at hand continues to be optimal or even valid.

Consider the following "Tree Example" of Lespérance *et al.* [2000], which we modify slightly for our purposes. In this domain, an agent is trying to fell a tree and carry it home. To not faint from exhaustion she has to rest when feeling fatigue. She thus can decide between three actions, `chop`, `rest`, and `carry-tree`. We can express this planning problem including the hard constraint of resting when feeling fatigue, using the following Golog program

(**if** ¬fatigue **then** (`chop` | `carry-tree`) **else** `rest` **endIf**)$^*$ ; tree-at-home?

where $a|b$ denotes the non-deterministic choice between two sub-programs $a$ and $b$, $a;b$ denotes their sequence, and $\delta^*$ denotes the non-deterministic iteration of sub-program $\delta$.

[1] We used the presented approach in several RoboCup tournaments including the world-cups of 2003 and 2004.

A tree-chopping expert said that it usually takes five chops to fell a tree and our agent has an idea of how much labor, chopping and carrying, she can do until she needs a rest. From this model information and the hard constraints, a Golog interpreter may produce the following plan: `chop, chop, chop, rest, chop, chop, carry-tree`. Assume, however, that when executing this plan it happens that the agent feels fatigue already after two chops. The model was imprecise. The remaining plan is still executable but would violate the given hard constraints and would cause the agent to faint. The problem is that the hard constraints got lost in planning because the planner assumed correctness of the model. To account for such modeling errors and facilitate correct replanning, the method proposed by Ferrein *et al.* [2004] would produce the following annotated plan instead: $\mathfrak{M}(\neg\text{fatigue})$, `chop`, $\mathfrak{M}(\neg\text{fatigue})$, `chop`, $\mathfrak{M}(\neg\text{fatigue})$, `chop`, $\mathfrak{M}(\text{fatigue})$, `rest`, $\mathfrak{M}(\neg\text{fatigue})$, `chop`, $\mathfrak{M}(\neg\text{fatigue})$, `chop`, $\mathfrak{M}(\neg\text{fatigue})$, `carry-tree` and the proposed execution engine discards the plan if during execution a marker ($\mathfrak{M}$) is encountered whose condition fails to hold, contrary to what was expected.

While most work on monitoring is concerned with monitoring the execution of a plan, it is also possible that even during planning itself pieces of the tentative plan become invalid, for example when a precondition that was true earlier becomes false. This aspect of planning and monitoring in dynamic domains is addressed by Veloso *et al.* [1998] where the resulting approach is implemented in the PRODIGY framework [Veloso *et al.*, 1995]. In particular the authors propose to generate two kinds of monitors during planning: plan-based monitors targeting preconditions of actions in the tentative plan, and alternative-based monitors. The latter are relevant when preferences over different possible plans exist: during planning it may happen that the most preferred alternative is not possible for some reason. If this reason changes and the alternative becomes available, optimality requirements demand to pursue this alternative instead, so monitoring this reason is vital for optimality. While this presents an important aspect of generating optimal plans in practice, which we further investigate in Chapter 5, in the described

approach the problem is strongly simplified. It assumes that the quality of an alternative can be decided a-priori, that is, without exploring the corresponding part in the search tree first. It also assumes that the quality of a plan is purely determined by its comprising actions and in particular independent of the traversed states. This, and its lack of a formal foundation, are the main limitations of this approach.

### 3.2.3 State Evaluation

Recall that in our framework (cf. Figure 1.1) the State Estimation step is followed by a State Evaluation step to determine the relevance of the discovered and diagnosed discrepancy between the expected and the estimated state, to decide whether any kind of replanning is required or advisable.

One can generalize the technique implicitly used by several of the described approaches to answer this question, as that of annotating the plan at every step with the regression of the goal and remaining preconditions through the remainder of the plan. When the regressed goal holds in the state actually encountered during execution, the remainder of the plan is expected to succeed, according to the model. PLANEX uses STRIPS as the action description language and since STRIPS allows for neither conditional effects nor disjunctive preconditions nor uncertain action outcomes (disjunctive action effects), regression is very simple. This regression is implicitly done in the backward-chaining search performed by the planner and the annotation is part of the triangle tables used for representing plans. Our generalization also applies to SIPE, which is based on hierarchical planning, and IPEM, based on a partial-order planner. In both approaches the choice of actions to add to the plan is based on open preconditions (subgoals), starting from the goal, and in both approaches the dependencies between these sub-goals and the actions in the plan they are established by are represented in the plan. Again these representations are used during execution to verify the continued validity of the remaining plan and to support replanning at a basic level.

Neither of the recent logic-based frameworks for execution monitoring [De Giacomo *et al.*, 1998; Fichtner *et al.*, 2003; Bjäreland, 2001] mentions or formalizes this matter. Both Fichtner *et al.* [2003] and Bjäreland [2001] lack an evaluation step entirely. The specific monitor suggested by De Giacomo *et al.* [1998] performs evaluation not by regression at planning time, but through projection at run-time. This is done in the *Relevant* predicate (cf. Section 1.2). This is also the method used in [Soutchanski, 2003b]. We will discuss the benefits of a regression based approach in detail in later chapters of this thesis. Intuitively, though, regression exhibits the relevance structure of the problem, which allows quick determination of the relevance of discrepancies as they are observed, which can generally be done much faster than through projection, as we will demonstrate experimentally (cf. e.g., Chapter 4).

## 3.3 An Abstract Monitoring Approach

The stated generalization of existing approaches for monitoring plan validity can be taken even further, and stated as an informal, but general abstract approach for monitoring. It can be applied to both plan generation and plan execution with respect to a variety of objectives, including but not limited to plan validity. The approach can be described in two stages:

1. during plan generation, annotate the planning data structures with all information relevant to the achievement of the objective, then

2. when a discrepancy between assumed and actually estimated state of the world occurs, use this information to determine the degree of relevance of the discrepancy.

Given this description of the general abstract monitoring approach we propose, we can state the thesis of this document as follows:

*Annotating planning data structures with additional relevance information*

*during plan generation enables sound and efficient monitoring. In particular, it allows for sound and quick determination of the degree of relevance of state discrepancies.*

As has been demonstrated in the literature, the formal notion of regression is a powerful tool for determining the required relevance information. Hence, one possible and more concrete realization of the stated approach is as follows:

1. Determine all decision criteria that affect the choice of plan. Regress these criteria over the actions of the plan, and annotate the plan with the resulting formula.

2. To verify that the criteria continue to be met when an unexpected state is encountered, it is sufficient to verify the regressed formulae in the new state.

Hence, depending on what is being monitored (e.g., validity or optimality in terms of certain preferences), different plan annotations are required and different algorithms are needed in order to verify this information. In the remainder of this thesis we will develop several concrete instances of this abstract approach. In each case we will begin by identifying what needs to be included in the plan annotation, and how to exploit this annotation when state discrepancies are detected. We consider both state evaluation, i.e., determining whether the discrepancy is relevant to the objective, and recovery, i.e., the problem of adjusting existing planning data structures to the new state (cf. Section 1.2).

### 3.3.1 Other Uses of Relevance in Planning

Other uses of the notion of relevance in the planning literature exist, most notably in the context of conformant planning. The problem of conformant planning is that of devising a plan that achieves the goal for a set of possible initial states. This is necessary when only incomplete information regarding the initial state of the world is available. Many early approaches to this problem were based on the possible world semantics, in

which reasoning is done with respect to all possible states. This approach does not scale well to big problems as the set of possible states is exponentially larger than the set of states itself. Fortunately it is not always necessary to consider all possible states, since often certain approximations of this set are sufficient [Son and Baral, 2001]. Such approximations, may however be incomplete, which in the context of conformant planning means that no plan may be found, even when one exists. In addressing this issue, the notion of relevance has proved useful. Son and Tu [2006] propose a partitioned, partial representation of the initial belief state that achieves completeness under the particular approximation considered by the authors. A central role in identifying the set of partial states to consider is played by the relevance relationship between literals. For each literal the authors identify the set of all other literals it depends on. This allows them to reduce the set of considered partial initial states to only those that may lead to relevant differences over the course of executing sequences of actions.

Palacios and Geffner [2006] describe an approach for compiling conformant planning problems into classical planning problems, roughly, by means of replacing fluents with their epistemic counterparts and adjusting action preconditions and effects accordingly. However, again, the resulting representation is incomplete and solving the resulting classical planning problem may fail even when valid conformant plans exist. Therefore, similar to Son and Tu, Palacios and Geffner [2007] consider an extension of their earlier approach that considers relevant sub-sets of the set of all possible initial states in the compilation, in order to gain completeness regarding a given conformant planning problem.

Another existing use of relevance in planning aims at the simplification of the planning problem prior to planning [Nebel *et al.*, 1997]. The intuition is that, depending on the goal, certain fluents and/or operators are irrelevant and can be pruned from the problem description, which can significantly improve forward search based planning.

## 3.4 Monitoring Plan Validity in the Situation Calculus

In this section we formalize the notion of plan validity and provide a knowledge-level algorithm for monitoring plan validity, which is a direct result of applying the stated abstract approach.

Recall that a situation is simply a history of actions executed starting in $S_0$, for instance $do([\alpha_1, \ldots, \alpha_m], S_0)$ denotes the situation reached after executing action $\alpha_1$ through $\alpha_m$ in this order, starting in the initial situation $S_0$.

**Definition 1** (Plan Validity)**.** Given a basic action theory $\mathcal{D}$ and a goal formula $G(s)$, a plan $\vec{\alpha} = [\alpha_1, \ldots, \alpha_m]$ is *valid* in situation $S$ if

$$\mathcal{D} \models G(do(\vec{\alpha}, S)) \wedge Poss(\vec{\alpha}, S).$$

As such, a plan continues to be valid if, according to the action theory and the current situation, the precondition of every action in the plan will be satisfied, and at the end of plan execution the goal is achieved.

Applying the abstract monitoring approach of the previous section to the problem of monitoring plan validity provides us with the following strategy: The planner annotates each step of the plan with a sufficient and necessary condition for the validity of the plan. During plan execution these conditions are checked to determine whether plan execution should continue. The conditions can be generated through goal regression. The provision of the formal characterization of this monitoring approach makes it usable with a variety of planners, such as very effective heuristic forward search planners, rather than being restricted to the backward-chaining based planners which first implemented the basic idea.

**Definition 2** (Annotated Plan)**.** Given initial situation $S_0$, a sequential plan $\vec{\alpha} = [\alpha_1, \ldots, \alpha_m]$, and a goal formula $G(s)$, the corresponding annotated plan for $\vec{\alpha}$ is a se-

quence of tuples

$$\pi(\vec{\alpha}) = (G_1(s), \alpha_1), (G_2(s), \alpha_2), \ldots (G_m(s), \alpha_m)$$

where

$$G_i(s) = \mathcal{R}\big[G(do([\alpha_i, \ldots, \alpha_m], s)) \wedge Poss([\alpha_i, \ldots, \alpha_m], s), s\big]$$

That is, each step is annotated with the regression of the goal and the preconditions over the remainder of the plan.

**Proposition 1.** A sequence of actions $\vec{\alpha}$ is a valid plan in situation $S$ iff $\mathcal{D} \models \mathcal{R}\big[G(do(\vec{\alpha}, S)) \wedge Poss(\vec{\alpha}, S), S\big]$.

*Proof:* The proof is by induction using the Regression Theorem [Reiter, 2001, pp.65–66].

We can now provide a knowledge-level algorithm that characterizes the approach to monitoring plan validity described above. It is a generalization of the algorithm defined in [Fikes *et al.*, 1972]. We assume that the "actual" situation of the world, denoted $S^*$, is provided by the earlier state estimation step of the execution monitoring framework described in Section 1.2. The action theory $\mathcal{D}$ remains unchanged. For instance, $S^*$ may differ from the expected situation $S_i = do([\alpha_1, \ldots, \alpha_{i-1}], S_0)$ by containing unanticipated exogenous actions, or variations of actions executed by the agent. To better distinguish variables of the theory and meta-variables, used by the pseudo code, we print the latter using bold face. We refer to our algorithm as knowledge-level because the test to determine that a condition holds in the current state is characterized in terms of classical entailment, i.e., $D \models G_i(S^*)$. We assume that each individual planning system has a specific means of accomplishing this test, the details of which are not germane to the algorithm itself. In STRIPS, for instance, this is simple set inclusion of literals in the set describing the current state. For the situation calculus efficient Prolog implementations exist.

The correctness of this approach—only valid plans are continued and whenever a plan is still valid it is continued—is provided by the following theorem.

**Algorithm 1**: Algorithm for Monitoring Plan Validity

```
Input: action theory 𝒟, annotated plan π(α⃗) of length m
1  begin
2      obtain actual situation S*;
3      while 𝒟 ⊭ G(S*) do
4          i = m ;
5          while 𝒟 ⊭ G_i(S*) and i > 0 do
6              i = i − 1;
7          if i > 0 then
8              execute α_i;
9          else
10             abort and replan;
11         obtain S* ;
12 end
```

**Theorem 1** (Correctness)**.** The algorithm executes action $\alpha_i$ in the given situation $S^*$ iff the goal has not yet been reached, the remaining plan $[\alpha_i, \ldots, \alpha_m]$ is valid in situation $S^*$, and $i$ is the greatest index in $[1, m]$ with that property.

*Proof:* If $\alpha_i$ is executed, $\mathcal{D} \models \neg G(S^*)$ holds, or else the outer **while**–loop would not have been entered, and also $\mathcal{D} \models G_i(S^*)$ holds, or else the inner **while**–loop could only have been exited by the condition $i \not> 0$, but then the condition of the **if**–statement would have failed to hold. Given $\mathcal{D} \models G_i(S^*)$, plan validity follows by Propositions 1. On the other hand, if the goal has not been reached yet, the outer **while**–loop is entered, and if there is a maximal index $i$ such that $[\alpha_i, \ldots, \alpha_m]$ is a valid plan in situation $S^*$, then by Propositions 1 also $\mathcal{D} \models G_i(S^*)$. Hence the inner **while**–loop terminates once this value of $i$ is reached, and since this value is greater than zero by assumption, the condition of the **if**–statement holds. Thus $\alpha_i$ is executed. □

As mentioned previously, this approach of annotating the plan in each step with the regression of the goal and remaining preconditions, and exploiting this annotation during execution to determine continued validity of the plan, has been used implicitly by several previous execution monitoring systems. However, to the best of our knowledge,

we are the first to formally characterize it using the notion of regression. This provision enables its use with other planners, i.e., not only the ones used in those systems, and it allowed us to proof its correctness. But more importantly in the context of this thesis, the characterization provides us with a more abstract, conceptual understanding of this monitoring approach, and allows us to generalize it and apply it to more complex monitoring problems, as will be demonstrated by the following chapters.

Note also that the approach itself does not require complete knowledge of the initial state of the world. As long as the annotated conditions can be evaluated, the agent is able to verify plan validity.

# Chapter 4

# Monitoring Plan Optimality During Execution

## 4.1 Introduction

In the previous chapter we described how the abstract monitoring approach can be used to monitor plan validity. This effectively formalized the approach taken by a number of existing systems implicitly. However, not only the validity of a plan, but also its *optimality* may be affected by unexpected execution time discrepancies between the expected and actual state of the world. Optimality appears in cases where the user not only specifies a goal to define valid plans, but also wishes to discriminate between all possible valid plans, by designating a measure of utility or preference over plans.

Monitoring optimality amounts to deciding whether, given a discrepancy, the current plan continues being the best feasible plan according to these preferences, or whether due to the change a different plan has become superior. Unlike the previously described monitoring of plan validity, there are virtually no approaches for monitoring plan optimality.

### 4.1.1 Contributions

Given an optimal plan, our objective is to monitor its continued optimality, electing to replan only in those cases where continued execution of the plan will either not achieve the goal, or will do so sub-optimally.

In this chapter, we achieve this objective by again applying the presented abstract monitoring approach. We begin by defining a sufficient condition for the optimality of a given plan, i.e., a condition over situations that, if satisfied by a particular situation $S$, guarantees that the plan is optimal in that situation. From this we derive a plan annotation and an algorithm that can be used to verify continued optimality of the plan during execution, and prove its correctness. We have implemented our algorithm and tested it on simulated execution failures in well-known planning domains. Experimental results yield an average speed-up in performance of two orders of magnitude over the alternative of replanning, clearly demonstrating the feasibility and benefit of the approach.

While we argue that the conceptual approach developed here is broadly applicable to a variety of planners, in this and the following chapter we focus on $A^*$ search based planners and a metric function defined in terms of positive action costs. To this end, we first briefly review $A^*$ search based plan generation and formalize it within the framework of the situation calculus (also confer [Lin, 1999]).

## 4.2 $A^*$ Search Based Planning

To provide a formal characterization, we assume that the planning domain is encoded in a basic action theory $\mathcal{D}$. To keep the presentation simple, we also assume that the goal is a fluent $G(s)$ and can only be established by a particular action *finish*, contained in the action theory. Any planning problem can naturally be translated to conform to this by defining the preconditions of the *finish* action corresponding to the goal of the original planning problem: if the original goal was described by a formula $G'(s)$, then

define $Poss(\mathit{finish}, s) \stackrel{\text{def}}{=} G'(s)$.

Recall that in $A^*$ search [Hart *et al.*, 1968], the evaluation function is the sum of the heuristic function and the accumulated action costs. We denote functions in relational form. The cost $c$ of performing action $a$ in situation $s$ is denoted by the relational fluent $Cost(a, c, s)$. The search heuristic yielding value $h$ in $s$ is denoted by $Heur(h, s)$. We require this heuristic to be monotonic, i.e., in particular admissible. Furthermore, we assume that for all situations $s$ these two relations hold for exactly one value $c$ and $h$, respectively. We understand this to denote a formula provided by the user, for instance of the form

$$\begin{aligned} Heur(h, s) \quad \stackrel{\text{def}}{=} \quad & (\phi_1(s) \wedge h = h_1) \\ & \vee(\phi_2(s) \wedge h = h_2) \\ & \vdots \\ & \vee(\phi_n(s) \wedge h = h_n) \end{aligned}$$

where the conditions $\phi_i$ partition state space. Correspondingly the $A^*$ evaluation relation is specified as follows:

$$\begin{aligned} & Value(v, do([\alpha_1, \ldots, \alpha_n], s)) \stackrel{\text{def}}{=} \\ & \quad (\exists h, c_1, \ldots, c_n).Heur\big(h, do([\alpha_1, \ldots, \alpha_n], s)\big) \wedge Cost(\alpha_1, c_1, s) \wedge \\ & \quad \cdots \wedge Cost\big(\alpha_n, c_n, do([\alpha_1, \ldots, \alpha_{n-1}], s)\big) \wedge v = h + c_1 + \cdots + c_n \end{aligned}$$

By the definition of monotonicity we know that it is non-decreasing:

$$\mathcal{D} \models (\exists v).Value(v, s) \supset (\nexists s', v').s \sqsubset s' \wedge Value(v', s') \wedge v' < v. \tag{4.1}$$

In $A^*$ search, nodes that have been seen but not explored are kept in the so-called *open list*, sorted by their respective values. In plan generation, the open list is initialized to the initial situation. Plan generation proceeds by repeatedly removing the situation with the lowest value from the open list and inserting its feasible successor

situations. Search terminates successfully when this first element, $do(\vec{\alpha}, S_0)$, satisfies the goal, $\mathcal{D} \models G(do(\vec{\alpha}, S_0))$. The plan described by $\vec{\alpha}$ is guaranteed to be optimal because any alternative plan would be a continuation of one of the partial plans in the open list, but by Equation (4.1) and the fact that $do(\vec{\alpha}, S_0)$ achieves a lower value than any other element in the open list—or else it would not have been the first element in that list—no such plan could have lower accumulated costs than $\vec{\alpha}$.

## 4.3 A Sufficient Condition for Optimality

Standard goals enable a planner to distinguish between valid and invalid plans—those that achieve the goal and those that do not. By augmenting a plan specification with some measure of the quality of the plan, prescribed as user preferences, a utility or other metric function, a planner can discriminate between valid plans, and can generate a plan that is optimal with respect to this measure of quality.

Given an optimal plan, our objective is to monitor its continued optimality, electing to replan only in those cases where continued execution of the plan will either not achieve the goal, or will do so sub-optimally. To this end, we apply the abstract plan-annotation based monitoring approach proposed in Chapter 3 to monitor plan optimality. This is a critical problem for many real-world planning systems, and one that has been largely ignored in the literature.

We begin by defining plan optimality within our framework. Recall that $do(\vec{\alpha}, S)$ denotes the situation reached after performing the action sequence $\vec{\alpha}$ in situation $S$. The relation $Pref(s, s')$ is an abbreviation for a sentence in the situation calculus defining criteria under which $s$ is preferred to $s'$. We assume this formula is provided by the user.

**Definition 3** (Plan Optimality)**.** Given a basic action theory $\mathcal{D}$, a goal formula $G(s)$, and extra-logical relation $Pref(s, s')$, a plan $\vec{\alpha}$ is *optimal* in situation $S$ if $\mathcal{D} \models G(do(\vec{\alpha}, S)) \wedge Poss(\vec{\alpha}, S)$ and there is no action sequence $\vec{\beta}$ such that $\mathcal{D} \models Pref(do(\vec{\beta}, S), do(\vec{\alpha}, S)) \wedge$

$G(do(\vec{\beta}, S)) \wedge Poss(\vec{\beta}, S)$.

As such, a plan remains optimal in a new situation $S^*$ when it remains valid and there exists no other valid plan that is preferred. Hence, to monitor plan optimality, we require two changes compared to the previously presented approach for monitoring plan validity:

1. in addition to regressing the goal, we must regress the preference criteria to identify conditions that are necessary to verify optimality at each step of the plan, and

2. since optimality is relative rather than absolute, we must annotate each plan step with the regression of the preferences over alternative plans as well.

This approach can be applied to a wide range of preference representation languages and planners. In this chapter, we restrict our attention to preferences described by positive numeric action costs, and an $A^*$ search based forward planner with a monotonic evaluation function as described above.

The preference relation for $A^*$ search is defined as:

$$Pref_{A^*}(s_1, s_2) \stackrel{\text{def}}{=} (\exists v_1, v_2).\, Value(v_1, s_1) \wedge Value(v_2, s_2) \wedge v_1 < v_2.$$

By the definition of monotonicity we again know that it is non-decreasing, that is if $s_1$ is preferred to $s_2$, then no continuation of $s_2$ is preferred to $s_1$:

$$\mathcal{D} \models Pref_{A^*}(s_1, s_2) \supset (\nexists s_2').s_2 \sqsubset s_2' \wedge Pref_{A^*}(s_2', s_1). \tag{4.2}$$

This property allows us to formulate a *sufficient* condition for the continued optimality of a plan $\vec{\alpha}$ in terms of a comprehensive set of alternative plan prefixes, rather than the complete set of alternative plans. To do so, we can exploit the open list when search terminates, but need to extend this list by action sequences that were previously predicted to be impossible but are now possible (in $S^*$). If $do(\vec{\alpha}, S^*)$ is still most preferred with respect to the resulting, updated, open list, then the plan $\vec{\alpha}$ remains optimal. However,

even after the update this is not a *necessary* condition. It may be the case that another element $do(\vec{\beta}, S^*)$ in the (revised) open list is preferred to $do(\vec{\alpha}, S^*)$, but if $do(\vec{\beta}, S^*)$ is only a plan prefix, rather than a complete plan, and hence does not satisfy the goal, $do(\vec{\alpha}, S^*)$ may still turn out to be optimal. This could be the case if (i) no continuation of $do(\vec{\beta}, S^*)$ satisfies the goal, or (ii) no such continuation is preferred to $do(\vec{\alpha}, S^*)$. This can occur because the heuristic value can, and generally does, increase as further actions are performed. Formally, $\mathcal{D} \models Pref_{A^*}(s_1, s_2) \not\supset (\nexists s_1').s_1 \sqsubset s_1' \wedge Pref_{A^*}(s_2, s_1')$.

Neither of these issues can be resolved without further, time consuming, planning. For this reason, in this chapter, we limit ourselves to a tight sufficient condition, defined in terms of the fringe.

**Definition 4** (Sufficient Condition for Optimality)**.** Let $G$ be a goal formula, *Pref* a preference relation, and $\vec{\alpha}$ a valid plan for $G$ in situation $S$. A *sufficient condition for the optimality* of $\vec{\alpha}$ in $S$ is any condition $\varphi(S)$ such that if $\varphi(S)$ holds, $\vec{\alpha}$ is optimal in situation $S$.

In order to define a fringe, we introduce the notion of a situation cover.

**Definition 5** (Situation Cover)**.** Given an action theory $\mathcal{D}$ and a situation term $S$, a set $L$ of situations is a *cover of* $S$ if for all $S'$ such that $\Sigma \models S \sqsubseteq S'$ there exists $S'' \in L$ such that either $\Sigma \models S' \sqsubseteq S''$ or $\Sigma \models S'' \sqsubseteq S'$. The cover is *minimal* if no proper subset of $L$ is a cover.

**Definition 6** (Fringe)**.** Given an action theory $\mathcal{D}$ and a goal formula $G(s)$, a *fringe* of situation $S$ is a minimal cover $L$ of $S$, such that for all $S' \in L$, there is no $S''$ such that $\mathcal{D} \models S'' \sqsubseteq S'$ and $\mathcal{D} \models G(S'')$.

Consider the example search tree of Figure 4.1. There both of the following two sets are

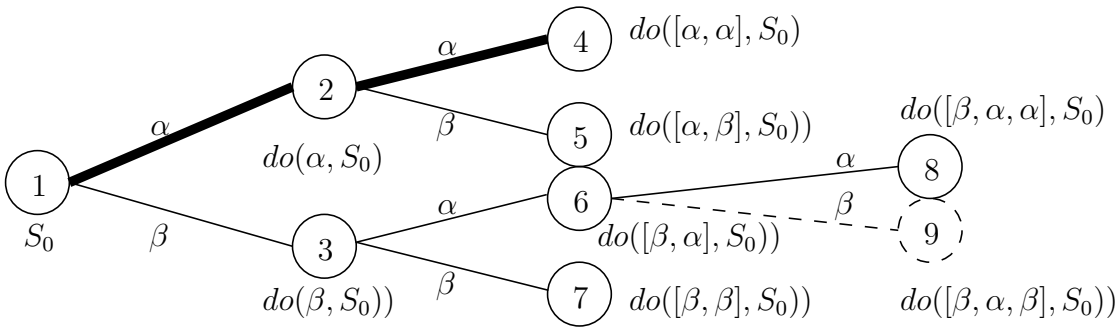



Figure 4.1: An example search tree. Dashed lines denote impossible actions, and $[\alpha, \alpha]$ is the assumed optimal plan.

fringes of $S_0$:

$$\left\{ \begin{array}{l} do([\alpha, \alpha], S_0), \\ do([\alpha, \beta], S_0), \\ do([\beta, \beta], S_0), \\ do([\beta, \alpha, \alpha], S_0), \\ do([\beta, \alpha, \beta], S_0) \end{array} \right\} \qquad \left\{ \begin{array}{l} do([\alpha, \alpha], S_0), \\ do([\alpha, \beta], S_0), \\ do(\beta, S_0) \end{array} \right\}$$

The former corresponds to the nodes 4, 5, 7, 8, and 9, and the latter corresponds to the nodes 4, 5, and 3.

Note the similarity between fringe and open list. The main difference is that a fringe can include infeasible situations. An important property of fringes is that any plan has exactly one prefix in any given fringe. This exhaustive character of fringes allows us to make optimality guarantees in conjunction with a heuristic function.

**Theorem 2** (Sufficiency)**.** Let $\mathcal{D}$ be an action theory, $G(s)$ a goal, $S$ a situation, and $\vec{\alpha}$ a valid plan for $G(s)$ in $S$. The existence of a fringe $L$ of $S$ such that for every $do(\vec{\beta}, S) \in L$, $\mathcal{D} \models (\forall v_a, v_b).\mathcal{R}\big[Poss(\vec{\beta}, s) \wedge Value(v_a, do(\vec{\alpha}, s)) \wedge Value(v_b, do(\vec{\beta}, s)), s\big](S) \supset v_b \geq v_a$, is a sufficient condition for the optimality of $\vec{\alpha}$ in $S$.[1]

*Proof:* Assume to the contrary that $\vec{\alpha}$ is not optimal in $S$, then, by applying the Regression

[1]Since $\vec{\beta}$ is a particular, known action sequence, regressing over it is not a problem and our abbreviation $Poss(\vec{\beta}, s)$ is well defined.

Theorem, there is a $S_b = do(\vec{\beta}, S)$ s.t. $\mathcal{D} \models (\exists v_a, v_b).\mathit{Value}(v_a, do(\vec{\alpha}, S)) \wedge \mathit{Value}(v_b, S_b) \wedge v_b < v_a$ and $\mathcal{D} \models G(S_b) \wedge \mathit{Poss}(\vec{\beta}, S)$. By definition of a fringe, $S_b$ is either in $L$ or is a sub-history or a continuation of some element in $L$. Since no element in $L$ is preferred to $\vec{\alpha}$ and no sub-history of any element in $L$ satisfies the goal, $S_b$ has to be a continuation of some element in $L$. But by definition of monotonicity (Equation (4.2)), no such continuation can be preferred to $\vec{\alpha}$.

□

This theorem establishes sufficient conditions for determining continued plan optimality. We must now translate these conditions into plan annotations that can be quickly checked during plan execution.

## 4.4 Algorithm

### 4.4.1 Annotation

With sufficient conditions for continued plan optimality formally characterized, we can now define the annotations we must add to an optimal plan to monitor continued optimality.

Plan annotations can be computed at plan generation time by our $A^*$ search forward planner, or after plan generation is completed. We assume our planner will output an optimal plan $\vec{\alpha}$, the open list $O$ that remained when plan generation terminated (e.g., nodes 5, 7, and 8 in Figure 4.1), and a list $O^-$ containing those situation terms found to be infeasible during plan generation. This is a list of situations $do(\alpha^-, S)$ such that $\alpha^-$ is not possible in situation $S$, i.e., $\mathcal{D} \models \neg\mathit{Poss}(\alpha^-, S)$ (cf. node 9 in the figure). The union $\{do(\vec{\alpha}, S_0)\} \cup O \cup O^-$ is a fringe of $S_0$.

Since monitoring optimality must be done relative to alternatives, each step of the optimal plan is annotated with the conditions that confirm its continued validity, as well as a list of alternative plans and their corresponding predicted evaluation function values

relativized to that plan step. Note that in the text that follows it is the description of the annotation, and not the annotation itself, that is defined in the situation calculus.

**Definition 7** (Annotated Plan (Optimality))**.** Given the initial situation $S_0$, the goal formula $G(s)$, the evaluation relation $Value(v, s)$, an optimal sequential plan $\vec{\alpha} = [\alpha_1, \ldots, \alpha_m]$, the open list $O$, and the list of infeasible situations $O^-$, the corresponding annotated plan for $\vec{\alpha}$ is a sequence of tuples

$$\pi(\vec{\alpha}) = (G_1(s), V_1, Alt_1, \alpha_1), \ldots, (G_m(s), V_m, Alt_m, \alpha_m)$$

where at each plan step $i$, $G_i(s)$ is as defined in Definition 2 and $V_i$ and $Alt_i$ are defined as follows, with $\vec{\alpha}_i = [\alpha_i, \ldots, \alpha_m]$ the remaining plan, and $S_i = do([\alpha_1, \ldots, \alpha_{i-1}], S_0)$ the expected situation:

$$\begin{aligned}
&V_i = (\phi_i^{Value}(v, s), \mathbf{v_i}), \quad \text{with} \\
&\phi_i^{Value}(v, s) = \mathcal{R}\big[\, Value(v, do(\vec{\alpha}_i, s)), s\big], \text{ and} \\
&\mathbf{v_i} \in \mathbb{R} \text{ such that } \mathcal{D} \models Value(\mathbf{v_i}, do(\vec{\alpha}_i, S_i)).
\end{aligned}$$

$Alt_i$ is the set of all tuples

$$Alt_{i_j} = \big(\vec{\beta}_{i_j}, \phi_{i_j}^{Poss}(s), \mathbf{p_{i_j}}, \phi_{i_j}^{Value}(v, s), \mathbf{v_{i_j}}\big) \text{ such that } do(\vec{\beta}_{i_j}, S_i) \in O \cup O^-,$$

where:

$$\begin{aligned}
&\phi_{i_j}^{Poss}(s) = \mathcal{R}\big[Poss(\vec{\beta}_{i_j}, s), s\big], \\
&\mathbf{p_{i_j}} = \begin{cases} 1, & \text{if } do(\vec{\beta}_{i_j}, S_i) \in O; \\ 0, & \text{if } do(\vec{\beta}_{i_j}, S_i) \in O^-, \end{cases} \\
&\phi_{i_j}^{Value}(v, s) = \mathcal{R}\big[\, Value(v, do(\vec{\beta}_{i_j}, s)), s\big], \text{ and} \\
&\mathbf{v_{i_j}} \in \mathbb{R} \text{ such that } \mathcal{D} \models Value(\mathbf{v_{i_j}}, do(\vec{\beta}_{i_j}, S_i)).
\end{aligned}$$

For plan step $i$, $V_i$ contains the regression of the evaluation relation over the remaining plan $\vec{\alpha}_i$ and its value $\mathbf{v_i}$ with respect to the expected situation $S_i$. $Alt_i$ represents the

list of alternative action sequences $\vec{\beta}_{i_j}$ to $\vec{\alpha}_i$, together with their respective regressed preconditions and their truth value ($\mathbf{p_{i_j}}$, represented as 0 or 1) in $S_i$, and regression of the evaluation relation and particular value in $S_i$ ($\mathbf{v_{i_j}}$).

That is, we annotate each step of the plan with the regression of the goal and preconditions, and the regression of the evaluation predicate over the remaining plan as well as over all considered alternative remaining plans $\beta_{i_j}$ at that point. For example, in the search tree of Figure 4.1, $\vec{\alpha} = [\alpha, \alpha]$ is the optimal plan, $Alt_1$ (cf. node 1) contains tuples for the action sequences $[\alpha, \beta], [\beta, \alpha, \alpha], [\beta, \alpha, \beta], [\beta, \beta]$, and $Alt_2$ (cf. node 2) contains only one tuple for $[\beta]$. A detailed example, including the actual content of the annotation, is discussed in Section 4.5 below.

Intuitively, the regression of the evaluation relation over a sequence of actions $\vec{\alpha}$ describes in terms of the current situation, the value the evaluation relation will take after performing $\vec{\alpha}$. As an example, consider the task of delivering a package to a location using a truck. Assume the heuristic yields a value $h = 0$ when the truck has the package loaded and is at the right location, and $h = 1$ otherwise. Then, regressing the heuristic through the action of driving the truck to the right location would yield a formula stating

$$(\text{package is on the truck} \wedge h = 0) \vee (\text{package is not on the truck} \wedge h = 1).$$

The *key* benefit of our approach comes from regressing conditions to the situations where they are relevant. Consequently when a discrepancy is detected during plan execution and the world is in a state that is different from the one assumed during plan generation, the monitor can determine the difference between these states and limit computation to reevaluating those conditions that are affected by the discrepancy (we will make this more precise below). This can result in significant computational savings, and is only possible because regression has enabled the definition of relevant conditions with respect to the situation before executing the remainder of the plan or any alternative.

### Space Complexity

We briefly study the space complexity of the annotation. We are interested in the complexity of the general approach, independent of the used action specification language and regression. We hence assume that the size of the $V_i$ and $Alt_{i_j}$ is expressed by a constant $c$. Then the space required to store the annotation is not significantly larger in the worst case than that for storing the open list itself, and in fact, asymptotically approaches that size for increasing cardinalities of actions. Let there be $n$ actions, and let $m$ denote the length of the optimal plan. In the worst case, the open list contains $n^m$ elements, namely when the search tree is a complete tree of depth $m$.

**Theorem 3.** Let the search tree be a complete tree of depth $m$ and branching factor $n$. Then the size of the annotated search tree is in $\mathcal{O}(\lambda n^m)$ with $\lambda = \frac{n-\frac{1}{n^{m-1}}}{n-1}$.

*Proof:* Recall the format of the annotation. In the first step of the plan, $Alt_1$ has cardinality $n^m$. In the second step, $Alt_2$ has cardinality $n^{m-1}$, and so on until eventually, $|Alt_m| = n^{m-(m-1)} = n$. The cardinality of $V_i$ is always equal to one. Hence, in total we store:

$$\begin{aligned} & cn^m + cn^{m-1} + \cdots + cn^{m-(m-1)} \\ = \; & cn^m(1 + \frac{1}{n} + \cdots + \frac{1}{n^{m-1}}) = cn^m \frac{n - \frac{1}{n^{m-1}}}{n-1} \end{aligned}$$

□

Because $\lim_{n\to\infty} \lambda = 1$, this means that we generally have no space requirements above those required to store the search tree in the first place.

Of course, normally in planning, the planner only returns the plan, and not the entire search tree. In that respect our method does require a lot more memory than a normal executing engine that simply executes the actions of the plan in sequence. However, if the planner was able to store the search tree at one point, the system must have adequately sized memory. If nonetheless these memory requirements are impractical, then

approximations of the proposed annotation could be explored. A variety of approximations could be devised. Rather than considering the proposed formulae completely, one could, e.g., only consider the contained (ground) fluents. This would lead to a sufficient condition less tight than the one we propose, but would be more economical in terms of the required space.

### 4.4.2 Execution Monitoring

Assume we are given an annotated, optimal plan $\pi([\alpha_1, \ldots, \alpha_m])$ and that we have executed $[\alpha_1, \ldots, \alpha_{i-1}]$ for some $i \leq m$ and thus expect to be in situation $S_i = do([\alpha_1, \ldots, \alpha_{i-1}], S_0)$. Given the situation estimate $S^*$ and the annotated plan as described in Definition 7, our task is to decide whether execution of the remainder of the plan, $\vec{\alpha}_i = [\alpha_i, \ldots, \alpha_m]$, is still optimal. We will do this by reevaluating all relevant conditions in $S^*$ to verify that the current plan is still valid and achieves maximal value among all alternatives.

Recall the representation of alternative plans $\vec{\beta}$ in $Alt_i$, containing the regressed preconditions, evaluation relation, and their respective values during plan generation, i.e., with respect to $S_i$. Also recall that $V_i = (\phi_i^{Value}(v, s), \mathbf{v_i})$ where $\phi^{Value}(v, s)$ is a formula over free variables $v$ and $s$, denoting the regression of the evaluation relation over $\vec{\alpha}_i$. A naive procedure for monitoring plan optimality at execution is shown in Algorithm 2.

The algorithm prescribes to continue execution as long as no feasible element from the list of alternatives achieves a better value in $S^*$ than the current plan. The time cost of this algorithm is greatly determined by the computation of the condition on line 7 as it reevaluates all annotated formulae anew in $S^*$. We can *significantly* reduce this time by only reevaluating those conditions that may have been affected by the discrepancy between the predicted situation $S_i$ and actual situation $S^*$. The precise set of affected conditions may be approximated in a variety of ways. We here focus on perhaps the most straightforward way: considering all conditions that mention any affected ground

**Algorithm 2**: Naive Optimality Monitoring Algorithm

**Input**: the annotated plan $\pi(\vec{\alpha}) = (G_1(s), V_1, Alt_1, \alpha_1), \ldots, (G_m(s), V_m, Alt_m, \alpha_m)$

```
1  begin
2      i ← 1;
3      while i ≤ m do
4          obtain S* ;                 // obtain current situation from state estimation module
5          (φ_i^Value(v, s), v_i) ← V_i ;
           // plan still valid?
6          if D ⊨ G_i(S*) then
               // plan still optimal?
               if ( ∀(β, φ_β^Poss(s), p_β, φ_β^Value(v, s), v_β) ∈ Alt_i :
7                   D ⊨ φ_β^Poss(S*) ⊃ (∀v_a, v_b).φ^Value(v_a, S*) ∧ φ_β^Value(v_b, S*) ⊃ v_b ≥ v_a )
8              then
9                  execute α_i ;
10                 i ← i + 1
11             else
                   // plan may be sub-optimal
12                 replan
13         else
               // plan no longer valid
14             replan
15 end
```

fluent. For this approximation to work, we here assume a finite set of ground fluents in the domain. Another possible approximation would be to only consider fluent names (predicates and function names). This approximation is less tight, but does not require the assumption of a finite set of ground fluents.

Let $\Delta_F(S_i, S^*)$ be the set of all ground fluents which appear in any of the regressed conditions and whose truth values differ between $S_i$ and $S^*$, i.e.,

$$\Delta_F(S_i, S^*) \stackrel{\text{def}}{=} \left\{ F(\vec{X}) \;\mid\; F \in \mathit{fluents}(\pi(\vec{\alpha})) \text{ and } \mathcal{D} \models F(\vec{X}, S_i) \not\equiv F(\vec{X}, S^*) \right\}$$

where $\mathit{fluents}(\pi(\vec{\alpha}))$ is the set of all fluents appearing in any regressed formula in the annotated plan $\pi(\vec{\alpha})$. If the regressed formulae contain quantifiers over fluent arguments, we can determine all its groundings. Only conditions mentioning any of these affected fluents need to be reevaluated, all others remain unaffected by the discrepancy. An im-

proved algorithm for monitoring plan optimality during execution is shown in Algorithm 3.

While the plan has not been executed to completion (line 3), the algorithm does the following:

**line 6:** it verifies continued validity of the plan;

**lines 7–9:** if the regression of the evaluation function over the plan ($\phi^{Value}(v, s)$) mentions any affected fluent, it is reevaluated, obtaining new value $\mathbf{v_i^{new}}$;

**lines 10–19:** the algorithm then checks for each alternative $\vec{\beta}$ at this point of plan execution: whether its preconditions are affected and need to be reevaluated, whether its value is affected and needs to be reevaluated, and whether this alternative is now possible and superior to the current plan. If an alternative has become superior, the algorithm aborts execution and calls for replanning. Otherwise the next action of the plan is executed.

Intuitively, the **foreach** loop revises relevant values—the truth of preconditions and the value of the evaluation function—generated for $S_i$ with respect to the actual situation $S^*$, aborting execution only when a viable and superior alternative is found. Line 17 is most crucial: Here the regression of the evaluation relation over alternative plan $\vec{\beta}$ is reevaluated with respect to the actual current situation $S^*$, yielding a new value $\mathbf{v}_\beta^{\mathbf{new}}$ for the evaluation relation (cf. $Alt_{i_j}$ in Definition 7). This reevaluation only occurs if the regression result (formula) mentions fluents that are affected by the difference between the expected situation $S_i$ and the actual situation $S^*$. Otherwise the value cannot have changed as a result of the discrepancy and does not need to be recomputed. In the algorithm we use $\mathit{fluents}(\phi) \cap \Delta_F(S_i, S^*)$ to determine whether formula $\phi$ might be affected by the discrepancy between the expected situation $S_i$ and the actually estimated situation $S^*$. When $\phi$ mentions fluents with free variables among its non-situation arguments, this intersection checks the unifiability of the terms. For instance $F(M, x, s)$ unifies with

**Algorithm 3**: The `Monoplex` algorithm

**Input**: the annotated plan $\pi(\vec{\alpha}) = (G_1(s), V_1, Alt_1, \alpha_1), \ldots, (G_m(s), V_m, Alt_m, \alpha_m)$

**begin**
  $\mathbf{i} \leftarrow 1$ ;
  **while** $\mathbf{i} \leq \mathbf{m}$ **do**
    obtain $S^*$ ; *// obtain current situation from state estimation module*
    generate $\Delta_F(S_i, S^*)$ ; *// determine discrepancy between expected and actual state*
    *// check continued plan validity*
    **if** $\mathcal{D} \models G_{\mathbf{i}}(S^*)$ **then**
      *// update value of current plan if necessary*
      $(\phi_i^{Value}(v, s), \mathbf{v_i}) \leftarrow V_i$ ;
      **if** *fluents*$(\phi^{Value}(v, s)) \cap \Delta_F(S_i, S^*) \neq \emptyset$ **then** *// plan quality may be affected*
        $\mathbf{v_i} \leftarrow \mathbf{v_i^{new}}$ with $\mathbf{v_i^{new}}$ such that $\mathcal{D} \models \phi^{Value}(\mathbf{v_i^{new}}, S^*)$
      *// update value of all alternatives where necessary*
      **foreach** $\left(\beta, \phi_\beta^{Poss}(s), \mathbf{p}_\beta, \phi_\beta^{Value}(v, s), \mathbf{v}_\beta\right) \in Alt_i$ **do**
        *// update truth-value of precondition if necessary*
        **if** *fluents*$(\phi_\beta^{Poss}(s)) \cap \Delta_F(S_i, S^*) \neq \emptyset$ **then**
          **if** $\mathcal{D} \models \phi_\beta^{Poss}(S^*)$ **then**
            $\mathbf{p}_\beta \leftarrow 1$
          **else**
            $\mathbf{p}_\beta \leftarrow 0$
        *// update value if necessary*
        **if** $\mathbf{p}_\beta = 1 \wedge$ *fluents*$(\phi_\beta^{Value}(v, s)) \cap \Delta_F(S_i, S^*) \neq \emptyset$ **then**
          $\mathbf{v}_\beta \leftarrow \mathbf{v_\beta^{new}}$ with $\mathbf{v_\beta^{new}}$ such that $\mathcal{D} \models \phi_\beta^{Value}(S^*, \mathbf{v_\beta^{new}})$
        *// finally, verify that the alternative remains inferior to current plan*
        **if** $\mathbf{p}_\beta = 1 \wedge \mathbf{v}_\beta < \mathbf{v_i}$ **then**
          replan ; *// plan may be sub-optimal, abort execution, replan*
      *// plan remains optimal, continue execution*
      execute $\alpha_i$ ;
      $\mathbf{i} \leftarrow \mathbf{i} + 1$
    **else** *// plan is invalid*
      replan
**end**

$F(M, X, s)$, where $M$ and $X$ are constants. Again, the realization of the entailment ($\mathcal{D} \models \varphi(s)$) depends on the implemented action language. The method is in particular not reliant on the situation calculus and can be used with any action language for which regression can be defined.

**Theorem 4** (Correctness)**.** Whenever `Monoplex` executes the next step of the plan (execute $\alpha_i$), the remaining plan $\alpha_i, \ldots, \alpha_m$ is valid and optimal in the actual current situation $S^*$.

*Proof:* The continued validity follows immediately from Theorem 1 and line 6 of Algorithm 3.

Regarding optimality, by construction, the set $K$ consisting of the plan $\vec{\alpha}$ and the alternatives $\vec{\beta}_{i_j}$ in $Alt_i$ describes a fringe of $S_i$, meaning that the set of situations $\{do(\vec{\alpha}, S_i)\} \bigcup_j \{do(\vec{\beta}_{i_j}, S_i)\}$ is a fringe of $S_i$. We show that the set $K$ also describes a fringe of $S^*$, i.e., $\{do(\vec{\alpha}, S^*)\} \bigcup_j \{do(\vec{\beta}_{i_j}, S^*)\}$ is a fringe of $S^*$, and if the **foreach** loop terminates, we have for all $\vec{\beta}_{i_j}$ that $\mathcal{D} \models (\forall v_a, v_b).\mathcal{R}\big[Poss(\vec{\beta}_{i_j}, s) \wedge Value(v_a, do(\vec{\alpha}, s)) \wedge Value(v_b, do(\vec{\beta}_{i_j}, s)), s\big](S^*) \supset v_b \geq v_a$. The optimality then follows from Theorem 2.

Notice that if a set of action sequences $C$ describes a minimal cover of a situation $S$, i.e., $L = \{do(\vec{a}, S) \mid \vec{a} \in C\}$ is a minimal cover of $S$, then it also describes a minimal cover of any other situation $S'$. Hence, $\{do(\vec{\alpha}, S^*)\} \bigcup_j \{do(\vec{\beta}_{i_j}, S^*)\}$ is a minimal cover of $S^*$. Recall that we assume that the only way the goal is achieved is by executing the special action *finish*. Since after reaching the goal, no more actions are considered by the planner, the set $K$ cannot include any action sequences that include *finish* and mention actions afterwards. Hence, the additional condition for the fringe-property always holds in cases where the assumption about this *finish* action holds. Hence, $K$ describes a fringe of $S^*$ as well.

We further show that for each alternative $\beta$, $\mathbf{p}_\beta$ is 1 iff $\mathcal{D} \models \mathcal{R}[Poss(\vec{\beta}, s), s](S^*)$, and $\mathbf{v}_\beta$ is such that $\mathcal{D} \models \mathcal{R}[Value(\mathbf{v}_\beta, do(\vec{\beta}, s)), s](S^*)$, and that $\mathbf{v_i}$ is such that $\mathcal{D} \models \mathcal{R}[Value(\mathbf{v_i}, do(\vec{\alpha}, s)), s](S^*)$. The first follows from the if case of line 11: if all fluents

occurring in the regressed preconditions $\phi_\beta^{Poss}$ assume the same value in both $S_i$ and $S^*$, then also $\mathcal{D} \models \phi_\beta^{Poss}(S_i) \equiv \phi_\beta^{Poss}(S^*)$ and hence, since by construction $\mathbf{p}_\beta$ is 1 iff $\mathcal{D} \models \phi_\beta^{Poss}(S_i)$, the property holds. Otherwise, if the (truth)-values of some fluents appearing in $\phi_\beta^{Poss}$ differ between $S_i$ and $S^*$, then the formula is reevaluated and the variable $\mathbf{p}_\beta$ set accordingly (Line 12).

The properties regarding $\mathbf{v}_\beta$ and $\mathbf{v_i}$ follow analogously from lines 16 and 8, respectively.

□

As it is stated, the algorithm continues the execution of the current plans only if it remains optimal. In certain situations this may not be practical and a more lenient policy for when to abort execution and replan may be desired. Note that this can be easily achieved, by minimally modifying the algorithm. For instance, one could slightly soften the decision on line 18 to allow plans to continue execution as long as no alternative has the potential of being more than, say, 10% better than the current plan. This would be achieved by simply replacing the condition in the **if**–statement of this line with $\mathbf{p}_\beta = 1 \wedge \mathbf{v}_\beta < 0.9 \cdot \mathbf{v_i}$.

### 4.4.3 Exploiting the Search Tree

In the description of the algorithm we focused on *what* needs to be annotated and verified during execution, but not *how* to do this most efficiently. Working with the fringe directly causes a lot of redundant work. This is because many alternative action sequences share the same prefix and so the costs and preconditions of these prefixes are annotated and potentially reevaluated multiple times. We can avoid this by exploiting the search tree structure in both the annotation and the algorithm. The plan annotation is then done by annotating copies of the search tree structure and modifying the algorithm to operate on this structure. Further, an index, mapping fluents to conditions appearing in the annotation is used to pin-point the conditions possibly affected by a discrepancy.

Technically, the only difference between the search tree method and the algorithms described above is the way the open list is updated and the values of the plan and the alternatives are changed to reflect the new, actual situation. The subsequent decision whether to continue execution or to replan is based on the same sufficient condition as described above, hence making the two methods equivalent in terms of this monitoring decision. The experimental result shown in the next subsection are based on an implementation which already uses these improvements.

## 4.5 An Illustrative Example

Consider the following simplified example from the TPP domain, where an agent drives to various markets to purchase goods which she then brings to the depot (cf. Section A.1 in the appendix). For simplicity, assume there is only one kind of good, two markets, and the following fluents: in situation $s$, $at(s) = l$ denotes the current location $l$, $totalcost(s) = t$ denotes the accumulated costs $t$ of all actions since $S_0$, $requested(s) = r$ represents the number $r$ requested of the good, $price(m, s) = p$ denotes the price $p$ of the good on market $m$, and $driveCost(src, dest, s) = c$ the cost $c$ of driving from $src$ to $dest$. Let there be two actions: $Drive(dest)$ moves the agent from the current location to $dest$, and $BuyAllNeeded$ purchases the requested number of goods at the market of the current location. Assume, the planner has determined the plan $\vec{\alpha} = [Drive(Market1), BuyAllNeeded, Drive(Depot)]$ to be optimal, but has as well considered $\vec{\beta} = [Drive(Market2), BuyAllNeeded, Drive(Depot)]$ as one alternative among others. To shorten presentation, we ignore the heuristic here, i.e., assume uniform cost search ($h = 0$). Then

$$\begin{aligned} V_1 \;=\; \Big( & v = totalcost(s) + driveCost(at(s), Market1, s) \\ & + \big(requested(s) \cdot price(Market1, s)\big) \\ & + driveCost(Market1, Depot, s), \quad \mathbf{v_1} \Big) \end{aligned}$$

and similarly $Alt_{1_1} = \left(\vec{\beta}, \phi_{1_1}^{Poss}(s), \mathbf{p_{1_1}}, \phi_{1_1}^{Value}(v,s), \mathbf{v_{1_1}}\right)$ where, very similar to above,

$$\begin{aligned}\phi_{1_1}^{Value}(v,s) \;=\; \Big(\; & v = totalcost(s) + driveCost(at(s), Market2, s) \\ & + \big(requested(s) \cdot price(Market2, s)\big) \\ & + driveCost(Market2, Depot, s)\Big)\end{aligned}$$

where we ignore preconditions for simplicity and $\mathbf{v_1}$ and $\mathbf{v_{1_1}}$ are the respective values of the regressed value function for the plan and the alternative with respect to the situation where we expect to execute this plan, $S_0$.

Assume that even before the execution of the plan begins, a discrepancy in the form of an exogenous action $e$ happens, putting us in situation $S^* = do(e, S_0)$ instead of $S_0$. The question is, whether $\vec{\alpha}$ is still optimal and in particular still better than $\vec{\beta}$. This clearly depends on the effects of $e$. If $e$ does not affect any of the fluents occurring in above annotated formulae, it can be ignored, the plan is guaranteed to remain optimal. This would, for instance, be the case when $e$ represents the event of a price change on a market that is not considered, as that price does not appear in the regressed formulae, which only mention relevant fluents.

But even when $e$ affects a relevant fluent, replanning may not be necessary. Assume, for instance, that $e$ represents the event of an increased demand, that is, increasing the value $r$ of $requested(s)$, formally $\mathcal{D} \models requested(S^*) > requested(S_0)$. Then $\Delta_F(S_0, S^*) = \{requested(s)\}$ and we need to reevaluate the annotated conditions, as $\vec{\beta}$ may have become superior. This could be the case, for instance, if the drive cost to $Market1$ is lower than to $Market2$, but the price at this market is higher. Then, a higher demand may make $Market2$ favorable, as the drive cost is compensated more than before by the lower price. This can quickly be determined by reevaluating the annotated conditions in $S^*$, obtaining new values $\mathbf{v_1}$ for $\vec{\alpha}$ and $\mathbf{v_{1_1}}$ for $\vec{\beta}$. If $\mathbf{v_{1_1}} < \mathbf{v_1}$ then replanning is necessary, otherwise the plan is guaranteed to remain optimal.

## 4.6 Empirical Results

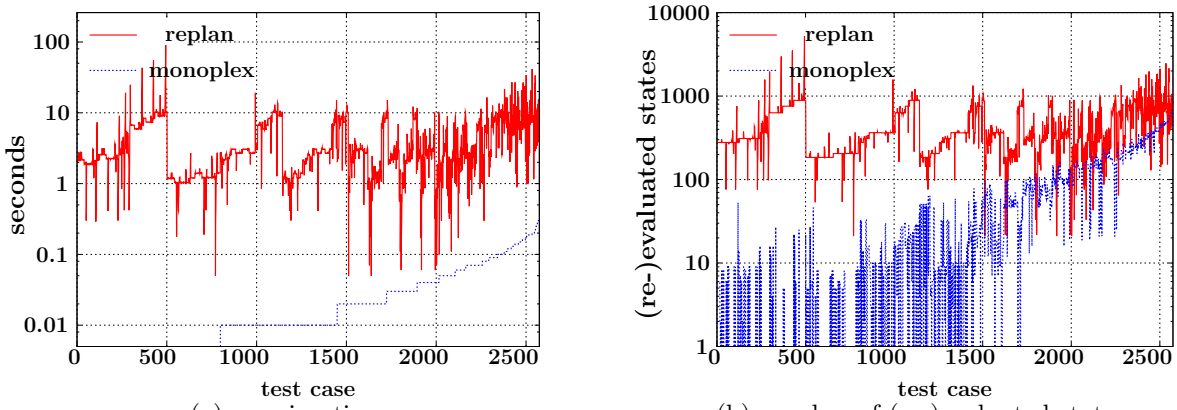
We have proved that our approach establishes conditions under which plan optimality persists in a situation. We were interested in determining whether the approach was time-effective—whether the discrepancy-based incremental reevaluation could indeed be done more quickly than simply replanning when a discrepancy was detected.

To this end, we compared a preliminary implementation of our `Monoplex` algorithm to replanning from scratch on 9 different problems in the metric `TPP` domain and two different problems of the `open stacks` domain with time (durative actions) of the $5^{th}$ International Planning Competition. In each case, we solved the original planning problem, perturbed the state of the world by changing some fluents, and then ran both `Monoplex` and replanning from scratch. To maximize objectivity, the perturbations were done systematically by multiplying the value of one of the numeric fluents by a factor between 0.5 and 1.5 (step-size 0.1), or by changing the truth value of a Boolean fluent.

### TPP

In the `TPP` domain (cf. Section A.1) this resulted in a total of 2574 unique test cases. Figure 4.2 shows the performance of both approaches on a logarithmic scale (all experiments were run on an Intel Xeon, 2.6GHz, 1GB RAM). To enhance readability we ordered the test cases by the running time of `Monoplex` . The determining factors for the running time (cf. Figure 4.2a) are predominantly the number of states for which the evaluation function had to be (re-)evaluated (4.2b), and the number of (re-)evaluated action preconditions. The discrepancy guidance of `Monoplex` found that on average only 1.18% of all preconditions evaluated during replanning were affected by the discrepancy and had to actually be reevaluated.

The results show that although `Monoplex` is sometimes slower than replanning (in 8 out of 2574 cases), it generally performs much better, resulting in a pleasing average speed-up of 209.12. In 1785 cases the current plan was asserted still optimal and therefore




(a) running time

(b) number of (re-)evaluated states

Figure 4.2: The time and number of (re-)evaluated states to determine continued plan optimality in the `TPP` domain, using replanning versus using `Monoplex` . For readability, the test cases were sorted by `Monoplex` time, and the y-axis has a logarithmic scale.

replanning unnecessary, in 105 it had become invalid. In 340 of the remaining 684 cases, replanning found the current plan to still be optimal. Notice in Figure 4.2b that sometimes the reevaluation of states can be entirely avoided, namely when the perturbation does not affect any relevant fluents. This happened 545 times and constitutes the greatest time savings potential, a result of our formal characterization of the situation-dependent relevance of fluents to the optimality of the plan.

### Open stacks

Less drastic, but similar in nature are the result for the `open stacks` domain (described in Section A.2 in the appendix), with an average speed-up of 139.84. Figure 4.3 shows the fraction of running time required by `Monoplex` compared to replanning for two different planning heuristics. In order to investigate the influence of the applied heuristic function we ran the same set of experiments with two different heuristics, 'A' and 'B', where 'B' is more informed than 'A' (cf. the respective lines in the figure). Again comparing `Monoplex` to replanning, we note that the performance of `Monoplex` improves with the use of a more informed heuristic and that it, in particular, preserves its superiority over

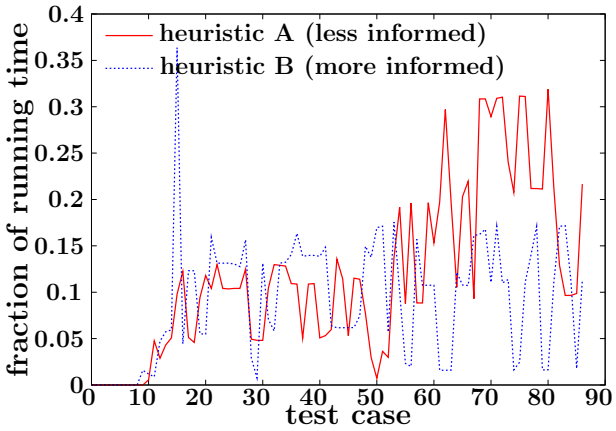



Figure 4.3: Ratio of running times "`Monoplex` / replanning" in `open stacks` domain. All test cases were repeated for two different heuristics, A and B, where B is more informed than A.

replanning. This meets the intuition that, like the planner, `Monoplex` benefits from a more focused search, resulting in a smaller search tree.

## 4.7 Discussion

When executing plans in dynamic environments, discrepancies between the expected and actual state of the world can arise for a variety of reasons. When such circumstances cannot be anticipated and accounted for during plan generation, they bring into question whether discrepancies are relevant, and whether they render the current plan invalid or sub-optimal. While there are several approaches for monitoring validity, no approaches exist for monitoring optimality. Instead it is common practice to replan when a discrepancy occurs or to ignore the discrepancy, accepting potentially sub-optimal behavior. Time-consuming replanning is impractical in highly dynamic domains and many discrepancies are irrelevant and thus replanning unnecessary, but to maintain optimality we have to determine which these are.

In this chapter we make several contributions to this problem. Starting from the abstract monitoring approach described in the previous chapter, we derive a concrete re-

alization of this monitoring approach for the notion of plan optimality. We have provided a sufficient condition for optimality and an algorithm that exploits knowledge about the actual discrepancy to quickly test this condition at execution. This approach guarantees plan optimality while minimizing unnecessary replanning.

Relevant properties of a situation as provided in the annotation, can further serve to focus on-line sensing when faced with limited sensing resources: the agent knows which features to sense and which can simply be ignored as they do not influence its objective.

We implemented our algorithm and tested it on systematically generated execution discrepancies in the `TPP` and `open stacks` domains. The results show that many discrepancies are irrelevant, leaving the current plan optimal, and that our approach is much faster than replanning, with an average speed-up of two orders of magnitude.

Theoretically, the requirements for the applicability of our method are easy to fulfill: The used action language has to be expressive enough to represent the user's preferences and heuristic function, and regression has to be defined. In practice however, the requirement of a regressable heuristic function is currently the main limitation of the approach. Many heuristics that have recently been shown very successful, like the popular FF heuristic [Hoffman and Nebel, 2001], are defined algorithmically and it is not obvious whether they can be described in a closed form, in a way that would be practical. Identifying heuristics that are regressable could hence be a promising direction for future work. Heuristics that are based on pattern databases (e.g., [Edelkamp, 2001]) might be a good starting point, since pattern databases in nature are static and hence more amenable to a logical representation as a regressable formula.

There are several other possible directions for future work as well. Certainly formalizing the approach for other planning paradigms would be worthwhile. We exemplify this for decision-theoretic planning in Chapter 6. One also could investigate the possibility of regressing the rationale in its entirety into one big formula instead of maintaining the tree structure. This formula could be brought into one of several possible normal forms, lend-

ing itself to possibly more sophisticated indexing schemes to further reduce reevaluations. Approximate representations of this formula may further be considered to account for limited memory of the execution engine. The normal forms and compilation techniques presented by [Darwiche and Marquis, 2002] may be a good starting point for this line of research. Also, when certain fluents occur in many of the annotated conditions, it may be feasible and beneficial to factor these fluents out, e.g., when the heuristic function is additive, to further minimize the number of conditions that need to be reevaluated when one of these fluents change unexpectedly.

We point out additional possible directions for future work in Section 9.3, in particular those that combine the presented approach with those developed in the remaining chapters of this thesis.

As we have seen, the proposed abstract monitoring approach based on explicit reasoning about relevance, can have great practical advantages during plan execution. However, given that plan generation itself requires time, the environment may change during plan generation as well. In the next chapter we study the problem of reacting to unexpected changes to the assumed initial state, occurring during the plan generation process itself. Again following the proposed abstract monitoring approach, we will devise a plan generation algorithm that is able to distinguish relevant from irrelevant state changes, and adjust the search tree accordingly, if necessary.

In practice, it is often the case that some degree of uncertainty can be predicted and quantified. When this is the case, a combination of conditional planning and execution monitoring seems most appropriate. We therefore continue our study of execution monitoring approaches in Chapter 6 with an implementation of the abstract monitoring approach for the execution of policies in domains with explicitly modeled stochastic action outcomes.

The restriction to final state goals can be a limitation in practice. In many cases,

agent designers, like robot programmers, wish to enforce or disallow certain types of behaviors, which often have temporal character. While languages have been developed that allow users to express such temporally extended constraints, the approach presented in this chapter is not immediately applicable for monitoring the execution of the generated plans. We will therefore describe a compilation approach in Chapter 7, which transform a planning problem with procedural constraints into a normal planning problem, which has the right format for our relevance based approach.

# Chapter 5

# Generating Optimal Plans in Highly Dynamic Environments

## 5.1 Introduction

In highly dynamic environments the initial state assumed by the planner frequently changes in unpredictable ways not only during plan execution, but during plan generation itself as well. Even when no plan or plan candidate has been identified yet, this may invalidate the current planning effort. We argue that neither boldly ignoring such changes nor replanning from scratch every time the initial state changes is an appealing option. While the plans produced by the former lack any guarantee of optimality or even validity, the latter may never finish plan generation if unexpected events keep interrupting. In this chapter, we propose an integrated plan generation and recovery algorithm which is a result of integrating the abstract monitoring approach proposed earlier into the planner itself. The planner is thus able to explicitly reason about the relevance and impact of discrepancies between assumed and observed initial state. As a result, it is able to "catch up with reality", meaning that it is generally able to finish plan generation eventually, while still guaranteeing optimal plans. Such an approach is particularly important in the

face of noisy sensors, where the agent's belief of the world frequently changes, though often only by small amounts, which often makes such changes irrelevant.

As a motivating example, consider a soccer playing robot in RoboCup, which, having the ball, deliberates about how to score. In RoboCup it is common to receive sensor readings 10 times per second. The game environment is very dynamic, resulting in frequent discrepancies between assumed and observed initial state. Such discrepancies may or may not affect the current planning process. But how can the robot tell? And how should the robot react when discrepancies are deemed relevant? For instance, assume that at some point during plan generation the current most promising plan starts with turning slightly to face the goal and then driving there, pushing the ball. If the ball unexpectedly rolls 10 centimeters away while deliberating, the initial turn action may cause the robot to lose the ball, so this discrepancy is relevant and another plan, starting by re-approaching the ball, should be favored. But if the ball rolls closer, the original plan remains effective and the discrepancy should be ignored and planning continued.

### 5.1.1 Contributions

The contributions of this chapter are three-fold:

1. We propose a novel algorithm for plan generation that monitors the state of the world during plan generation and recovers from unexpected state changes that impact planning. We prove that the algorithm produces plans that are optimal with respect to the state where execution begins. It is able to distinguish between relevant and irrelevant discrepancies, and updates the planning search tree to reflect the new initial state if necessary. This is generally much faster than replanning from scratch, as we demonstrate empirically, and works for arbitrary state changes that are representable in the domain specification.

2. We introduce a new criterion for evaluating plan adaptation algorithms: their *rel-*

*ative* running time compared to the "size" of the discrepancy. We argue that this measure is of greater practical significance than either theoretical worst case considerations or the absolute recovery time, for the following reason: In highly dynamic domains unexpected state changes occur during plan generation as well as during plan adaptation (recovery). In order to obtain a plan that is known to be optimal when execution commences, the cycle of plan generation and recovery has to terminate by a completed recovery before the state changes any further. This is possible when in practice the time for recovery is roughly proportional to the size of the change. Imagine plan generation takes 10 seconds and recovering from any state changes that occurred during that time takes 8 seconds. If we assume that in 8 seconds on average fewer changes happen than in 10, it seems reasonable to expect that we can recover from those in less than 8 seconds, say on average 6. This continues, until recovery has "caught up with reality". We informally say that an algorithm with this property *converges*. Repeated replanning from scratch obviously does not converge, as it does not differentiate between "big" and "small" discrepancies.

3. We show empirically that our algorithm can converge and find optimal plans in domains that were previously not amenable to planning, due to the high dynamics of the environment in which they are situated. Particularly "on-the-fly" recovery, i.e., recovering immediately upon discrepancy detection, has a higher chance of convergence than the alternative of completing the original planning task first and recovering only afterwards.

We explicitly assume that the number and extent of discrepancies increases over time, i.e., that greater discrepancies are incurred in longer time intervals. This seems reasonable to us and holds for many interesting application domains. This, together with the observation that our algorithm can recover from a few small changes faster than from many large ones, provides the convergence of our approach. We demonstrate this and the

resulting convergence of our approach empirically, on domain simulations which satisfy this assumption.

### 5.1.2 About Optimality in Dynamic Environments

We understand optimality to be defined in terms of what is currently known, and we want to execute plans only when they are considered optimal at the moment execution begins. This seems rational, since future events cannot generally be predicted. Nevertheless, we point out that this may lead to behavior that, in hindsight, is sub-optimal compared to a seemingly worse but quickly produced plan, namely when bad or catastrophic events in the environments can be avoided by planning and acting more quickly. We also explicitly assume that everything that matters for optimality is modeled in the theory. In particular, we assume that planning time itself does not directly affect optimality.

## 5.2 Algorithm

We continue focusing on a planner based on $A^*$ search that uses positive action costs as a metric. But as before, the conceptual approach is amenable to a variety of other forward-search based planning techniques and paradigms. We will elaborate on this in later chapters.

Intuitively, our approach annotates the search tree with all relevant information for determining the optimal plan. By regressing the goal, preconditions, and metric function over all considered action sequences, this information is expressed in terms of the current state. When unexpected events change the current state of the world, this information allows the planner to reason symbolically about the relevance of the changes and their potential impact on the current search tree and choice of plan—much faster then replanning from scratch. In particular, irrelevant changes can be discarded and do not trigger any kind of potentially costly replanning.

For instance, our soccer robot from above knows from regressing the goal that the plan ["turn", "drive to goal"] will succeed whenever "distance to ball $< 10cm$" holds. Hence it can determine the relevance of the aforementioned ball displacements, and also that, for instance, unexpected actions of its teammates can be ignored for now. A complication of this arises from our interest in an *optimal*, rather than just any valid plan. As before, we will need to also consider alternative action sequences, and also handle impacts on the regressed metric function.

At the highest level, the approach we present here consists of two components: A regression-based $A^*$ planner, and a recovery procedure. These can be used in at least two possible ways:

**At-the-end:** The planner generates an optimal plan for the assumed initial state. If no changes to the initial state occur, the resulting plan is optimal and execution can commence. Otherwise, the recovery procedure updates the final search tree and open list as necessary given any observed changes to the initial state. If the order of the open list does not change during the recovery, and hence still has the previously found plan as its first element, the plan is known to remain optimal and can be executed. Otherwise, the planner resumes plan generation given the updated structures.

**On-the-fly:** In the absence of any changes to the assumed initial state, the planner is proceeding in its search for an optimal plan. Whenever a change to the initial state is observed, plan generation is interrupted and the recovery procedure updates the current search tree and open list to reflect the changes. Plan generation then continues.

In both cases, this alternation continues until a plan generation cycle terminates without further interruptions, in which case the resulting plan is known to be optimal with respect to the currently assumed initial state.

In the next two subsections we propose a regression-based $A^*$ planner, which we call `RegBasA*`, and the recovery procedure which exploits the annotations created during plan generation.

### 5.2.1 Regression-Based $A^*$ planning

We present an $A^*$ planner which maintains a search tree annotated with any relevant regressed formulae, and which returns not only a plan, but also the remaining open list upon termination of search and the search tree. This planner is only a minor modification of the $A^*$ search planner described in Section 4.2. The main modification lies in the fact that all relevant formulae are regressed and stored in their respective nodes of the search tree for later reuse, and that an index is created, which maps ground fluents to the stored regressed formulae they appear in.

Our regression-based version of $A^*$ is shown in Algorithm 4. As before, we assume that the planning domain is described by a basic action theory $\mathcal{D}$, and that the user provided a goal formula $G(s)$, cost function $Cost(a, c, s)$, and monotonic heuristic $Heur(h, s)$, and, of course, an initial state described by a situation $S$. As before, we assume that the goal can only be achieved by a distinguished action *finish* (cf. Section 4.2). The algorithm interacts with $\mathcal{D}$ to reason about the truth-values of formulae. The algorithm takes as input the action theory $\mathcal{D}$, the assumed initial situation $S$, the cost and heuristic functions, *Cost*, *Heur*, an open list, and an annotated search tree, which is initially empty. The algorithm is initially invoked as `RegBasA*`$(\mathcal{D}, S, G, Cost, Heur, [(0, \infty, [\,])], nil)$. The elements of the open list are tuples $(g, h, \vec{\alpha})$, where $\vec{\alpha} = [\alpha_1, \ldots, \alpha_n]$ is an action sequence, $g$ are the costs accumulated when executing this sequence in $S$, and $h$ is the heuristic value, i.e., $\mathcal{D} \models Heur(h, do(\vec{\alpha}, S))$. When an element is expanded, it is removed from the open list and the following is performed for each agent action $\alpha'$: First, the preconditions of $\alpha'$ are regressed over $\vec{\alpha}$ (Line 11). If the resulting formula, stored in $T(\vec{\alpha}).P(s)$, holds in $S$ according to $\mathcal{D}$ (Line 12), the cost formula for $\alpha'$ is regressed over $\vec{\alpha}$, the heuristic is

**Algorithm 4**: `RegBasA*`: Regression-Based $A^*$ planning.

```
   Input: D, S, G, Cost, Heur, O, T
1  begin
2      if O = [ ] then
           // the open list is empty, no plan found
3          return ([ ], T)
4      else
5          [(g, h, α⃗) | O'] ← O ;        // divide open list into first element and remaining list
6          if D ⊨ G(do(α⃗, S)) then
7              return (O, T)
8          else
9              foreach α' ∈ A do
10                 α⃗' ← α⃗ · α' ;                                  // append action to sequence
11                 T(α⃗').P(s) ← R[Poss(α', s), α⃗] ;
12                 if D ⊨ T(α⃗').P(S) then
13                     T(α⃗').p ← 1 ;                              // action currently possible
                       // regress costs and heuristic
14                     T(α⃗').C(c, s) ← R[Cost(α', c, s), α⃗] ;
15                     T(α⃗').H(h, s) ← R[Heur(h, s), α⃗'] ;
                       // get current costs and current heuristic value
16                     T(α⃗').c ← c' with c' s.t. D ⊨ T(α⃗').C(c', S) ;
17                     T(α⃗').h ← h' with h' s.t. D ⊨ T(α⃗').H(h', S) ;
                       // insert new element into open list
18                     insert (g + c', h', α⃗') into O' according to g + c' + h' ;
19                 else
20                     T(α⃗').p ← 0 ;                              // action currently impossible
21             return RegBasA*(D, S, G, Cost, Heur, O', T)
22 end
```

regressed over $\vec{\alpha} \cdot \alpha'$, and the resulting formulae are evaluated in $S$ yielding values $c'$ and $h'$ (Lines 14–17). Intuitively, the regression of these formulae over $\vec{\alpha}$ describes in terms of the current situation, the values the respective functions will take after performing $\vec{\alpha}$. Finally, a new tuple $(g', h', \vec{\alpha}')$ is inserted into the open list (Line 18), where $g' = g + c'$. This insertion is done according to the sum of the first two elements of the tuples in the list $(g + h)$ to maintain the open list's order according to $Value(v, s)$ (cf. Section 4.2).

$A^*$ keeps expanding the first element of the open list, until this element satisfies the goal, in which case the respective action sequence describes an optimal plan. This is

because a monotonic heuristic never over-estimates the actual remaining costs from any given state to the goal, and we are assuming non-negative action costs. Due to the Regression Theorem [Reiter, 2001, pp.65–66], this known fact about $A^*$ also holds for our regression-based version. Similarly, the completeness of $A^*$ is preserved.

In service of our recovery algorithm described below, we explicitly keep the search tree, $T$, and annotate its nodes with the regressed formulae for preconditions ($T(\vec{\alpha}).P(s)$), costs ($T(\vec{\alpha}).C(c,s)$), and heuristic value ($T(\vec{\alpha}).H(h,s)$) and their respective values according to the (current) initial situation $S$ ($T(\vec{\alpha}).p, T(\vec{\alpha}).c$, and $T(\vec{\alpha}).h$). Roughly, when the initial state changes due to an unexpected event $e$, our recovery algorithm reevaluates $T(\vec{\alpha}).P(s), T(\vec{\alpha}).C(c,s)$, and $T(\vec{\alpha}).H(h,s)$ in $s = do(e,S)$, and updates their values and the open list accordingly. However, we can gain significant computational savings by only reevaluating those formulae actually affected by the state change. And since all formulae are *regressed*, we can determine which ones are affected, by simply considering the fluents they mention. For this purpose we create an index *Index* whose keys are ground fluent atoms (e.g., *distanceTo(Ball)*) and whose values are lists of pointers to all stored formulae that mention it. As in the previous chapter, the use of such an index requires us to assume a finite set of ground fluents in the domain. Again, other approximations of the affected conditions in the annotation exist.

### 5.2.2 Recovering from Unexpected Changes

While generating a plan for an assumed initial situation $S$, an unexpected event $e$, say "*distanceTo(Ball)* $\leftarrow$ 20", may occur, changing the state of the world and putting us into situation $do(e,S)$. When this happens, our approach consults the aforementioned index to pinpoint all formulae affected by this change (e.g., $T([$*turn, driveTo(goal), finish*$]).P(s)$). After reevaluating these formulae in $do(e,S)$ and updating their values, the search tree will be up-to-date in the sense that all its contained values are with respect to $do(e,S)$ rather than the originally assumed initial situation $S$. After propagating this change

to the open list, search can continue, producing the same result as if $A^*$ (or `RegBasA`$*$) had set out in $do(e, S)$ (cf. Theorem 5 below). *The regressed formulae never change.* Assuming that most unexpected state changes only affect a few fluents and thus often only affect a small subset of all formulae, our annotation allows for significant computational savings when recovering from changes compared to replanning from scratch, as we show empirically in the next section.

Algorithm 5 shows the recovery procedure. It receives as input the basic action theory $\mathcal{D}$, the assumed initial situation $S_1$ and actual situation $S_2$, as well as the cost function, heuristic function, open list $O$, and annotated search tree $T$, and the fluent index *Index*. The latter contains entries of the form $(\vec{\alpha}, type)$, where $\vec{\alpha}$ is a sequence of actions and *type* is either of `'p'`, `'c'`, or `'h'`, indicating that the fluent in question is mentioned in the regressed precondition, the regressed costs, or the regressed heuristic of the search tree node denoted by $\vec{\alpha}$, respectively. The algorithm modifies the values of the tree and the open list (ll. 25 and 29) to reflect their value with respect to the new situation $S_2$ (e.g., $do(e, S_1)$) rather than the originally assumed initial situation $S_1$. If the event changes the truth value of action preconditions, the content of the open list is modified accordingly (ll. 10, 19). When a previously impossible action has now become possible (Line 11) the annotation for this node is created and a new entry is added to the open list (ll. 13-19). The function $getGval(T, \vec{\alpha})$ computes the sum of all costs ($T(\cdot).c$) annotated in $T$ along the branch from the root to node $\vec{\alpha}$.

As said earlier, the algorithm can be used in one of at least two ways: during plan generation ("on-the-fly"), dealing with unexpected state changes immediately, or right after plan generation ("at-the-end"), dealing at once with all events that occurred during plan generation. The former has the advantage that the planning effort is focused more tightly on what is actually relevant given everything that has happened so far. This approach can be implemented by inserting code right before Line 21 of `RegBasA`$*$ that checks for events and invokes `Recover` if necessary, changing $S, O'$, and $T$ accordingly.

**Algorithm 5**: `Recover`: the recovery procedure.

**Input**: $\mathcal{D}, S_1, S_2, \mathit{Cost}, \mathit{Heur}, O, T, \mathit{Index}$

**begin**
  $\Delta_F \leftarrow \{F(\vec{X}, s) \in \text{keys}(\mathit{Index}) \mid \mathcal{D} \models F(\vec{X}, S_1) \not\equiv F(\vec{X}, S_2)\}$ ;
  $\Delta \leftarrow \bigcup_{f \in \Delta_F} \mathit{Index}(f)$ ; *// affected formulae*
  sort the elements $(\vec{\alpha}, \mathit{type})$ of $\Delta$ by length of $\vec{\alpha}$ ;
  **foreach** $(\vec{\alpha}, \texttt{`p'}) \in \Delta$ **do** *// update preconditions*
    **if** $T(\vec{\alpha}).p = 1$ *and* $\mathcal{D} \models \neg T(\vec{\alpha}).P(S_2)$ **then**
      $T(\vec{\alpha}).p \leftarrow 0$ ; *// action now impossible*
      **foreach** $(g, h, \vec{\alpha}') \in O$ **do** *// remove all affected elements from the open list*
        **if** $\vec{\alpha}$ *is prefix of* $\vec{\alpha}'$ **then**
          remove $(g, h, \vec{\alpha}')$ from $O$
    **else if** $T(\vec{\alpha}).p = 0$ *and* $\mathcal{D} \models T(\vec{\alpha}).P(S_2)$ **then**
      $T(\vec{\alpha}).p \leftarrow 1$ ; *// action now possible*
      $\vec{\alpha}' \cdot \alpha_{\text{last}} \leftarrow \vec{\alpha}$ ; *// get last action in sequence*
      $T(\vec{\alpha}).C(c, s) \leftarrow \mathcal{R}[\mathit{Cost}(\alpha_{\text{last}}, c), \vec{\alpha}']$ ;
      $T(\vec{\alpha}).H(h, s) \leftarrow \mathcal{R}[\mathit{Heur}(h), \vec{\alpha}]$ ;
      $T(\vec{\alpha}).c \leftarrow \mathbf{c}'$ with $\mathbf{c}'$ s.t. $\mathcal{D} \models T(\vec{\alpha}).C(\mathbf{c}', S_2)$ ;
      $T(\vec{\alpha}).h \leftarrow \mathbf{h}'$ with $\mathbf{h}'$ s.t. $\mathcal{D} \models T(\vec{\alpha}).H(\mathbf{h}', S_2)$ ;
      $g' \leftarrow \mathit{getGval}(T, \vec{\alpha})$ ;
      insert $(g', h', \vec{\alpha})$ into $O$ and update *Index*
  **foreach** $(\vec{\alpha}, \texttt{`c'}) \in \Delta$ **do** *// update accumulated costs*
    get $\mathbf{c}'$ s.t. $\mathcal{D} \models T(\vec{\alpha}).C(\mathbf{c}', S_2)$ ;
    **offset** $\leftarrow \mathbf{c}' - T(\vec{\alpha}).c$ ;
    **foreach** $(g, h, \vec{\alpha}') \in O$ **do**
      **if** $\vec{\alpha}$ *is prefix of* $\vec{\alpha}'$ **then**
        $g \leftarrow g +$ **offset** ; *// adjust accumulated costs in open list element*
    $T(\vec{\alpha}).c \leftarrow \mathbf{c}'$
  **foreach** $(\vec{\alpha}, \texttt{`h'}) \in \Delta$ **do** *// update heuristic values*
    **if** $(\exists g, h).(g, h, \vec{\alpha}) \in O$ **then** *// adjust heuristic estimate in open list*
      $h \leftarrow \mathbf{h}'$ with $\mathbf{h}'$ s.t. $\mathcal{D} \models T(\vec{\alpha}).H(\mathbf{h}', S_2)$ $T(\vec{\alpha}).h \leftarrow h$ ;
  **return** $(\mathit{sort}(O), T)$ ;
**end**

The appeal of the latter ("at-the-end") stems from the fact that recovering from a bulk of events simultaneously can be more efficient than recovering from each event individually. It may, however, be necessary to resume `RegBasA*` search afterwards, if, for instance, the current plan is no longer valid in the new initial state or a new opportunity exists, which may lead to a better plan. With both approaches, additional events may happen during recovery, making additional subsequent recoveries necessary (cf. our discussion about convergence in Section 5.1).

The following theorem states the correctness of `Recover` in terms of the "at-the-end" approach: calling `Recover` and continuing `RegBasA*` with the new open list and tree, produces an optimal plan and in particular the same as replanning from scratch in $S_2$. Recall that the first element of the open list contains the optimal plan. For "on-the-fly", correctness can be shown analogously (cf. Lemma 1 in Appendix B).

**Theorem 5** (Correctness)**.** Let $\mathcal{D}$ be a basic action theory, $G(s)$ a goal formula, $Cost(a, c, s)$ a cost formula, and $Heur(h, s)$ a monotonic heuristic. Then, for any two situations $S_1, S_2$ in $\mathcal{D}$ we have that after the sequence of invocations:

1. $(O_1, T_1) \leftarrow \texttt{RegBasA*}\big(\mathcal{D}, S_1, G(s), Cost(a, c, s), Heur(h, s), [(0, \infty, [\,])], nil\big)$,
2. create *Index* from $T_1$,
3. $(O_2, T_2) \leftarrow \texttt{Recover}\big(\mathcal{D}, S_1, S_2, Cost(a, c, s), Heur(h, s), O_1, T_1, Index\big)$,
4. $(O_3, T_3) \leftarrow \texttt{RegBasA*}\big(\mathcal{D}, S_2, G(s), Cost(a, c, s), Heur(h, s), O_2, T_2\big)$

the first element of $O_3$ will be the same as in $O'$ of

$$(O', T') \leftarrow \texttt{RegBasA*}\big(\mathcal{D}, S_2, G(s), Cost(a, c, s), Heur(s), [(0, \infty, [\,])], nil\big)$$

or both $O_3$ and $O'$ are empty.

*Proof:* Appendix B.

Due to the optimality of RegBasA$*$, this also means that any completed plan is guaranteed to be optimal with respect to $S_2$. This, in particular, works for any situation pair $S_1, S_2 = do(\vec{e}, S_1)$, for any sequence of events $\vec{e}$. Note that such events can produce arbitrary changes to the state of the world. The algorithm does not make any assumptions about the identity or cardinality of possible events. Any fluent may assume any value at any time.

In complex domains, many state changes are completely irrelevant to the current planning problem, overall or at the current stage of plan generation, and others only affect a small subset of elements in the search tree. During recovery, we exploit this structure to gain significant speed-ups compared to replanning from scratch. Practically more important than the mere speed-up, though, is that, on average, our algorithm recovers from small perturbations faster than from large ones, where "large" may refer to the number of fluents that changed or the amount by which continuous fluents changed. This is what allows our algorithm to *converge*, i.e., "catch up with reality", as we defined informally in the introduction. We verified this empirically.

## 5.3 Empirical Results

We present empirical results obtained using a current implementation of our algorithm to generate optimal plans for differently sized problems of the metric TPP and Zenotravel domains of the International Planning Competition (described in Sections A.1 and A.3 in the appendix). We begin by showing that the time required for recovering from unexpected state changes is roughly and on average proportional to the extent of the change. We then show that our approach is able to find optimal plans even when the initial state changes frequently. We compare the two mentioned recovery strategies on-the-fly and at-the-end, showing that the former clearly outperforms the latter in terms of likelihood of convergence. Finally, and not surprisingly, we show that our approach

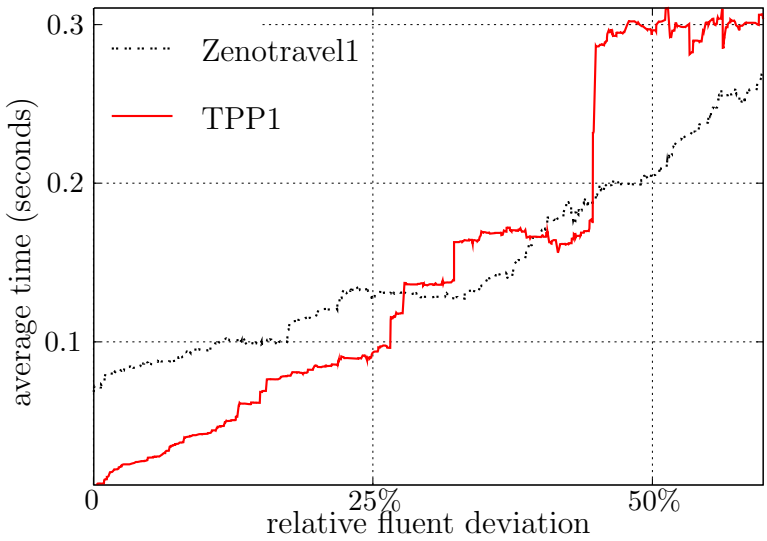



Figure 5.1: Recovery time relative to amount of change.

generally outperforms replanning from scratch. All experiments were run on an Intel Xeon 2.66 GHz with 2GB RAM.

Figure 5.1 plots the average time the combination of `Recover` + continued `RegBasA`$*$ search took to find a new optimal plan, after the value of a randomly selected continuous fluent was randomly modified after generating an optimal plan. A deviation $X\%$ means that the fluent was multiplied by $1 \pm \frac{X}{100}$, e.g., 50% means a factor of $1 \pm 0.5$.[1] The graph shows that recovering from a drastic change takes on average longer than recovering from minor deviations. While this does not seem surprising, we present it here since it provides the intuition for the convergence behavior of our approach, which we study next.

We assume that over longer periods of time more things change and in greater amounts than over shorter periods of time. Recovery generally takes less time than the original plan generation did (see below). Hence, we assume fewer or less drastic changes will happen during recovery than during plan generation. A second recovery—from the events that occurred during the first recovery—is thus predicted to take less time than the first. This process often continues until convergence. We studied the conditions under which

[1]Note that we used continuous fluents in our experiments only because they lend themselves better to a quantitative evaluation. Our approach is equally applicable to discrepancies on discrete valued, including Boolean, fluents.

our algorithm converges by simulating domains with frequent changes to the initial state. At high frequencies during plan generation and subsequent recoveries, we randomly perturbed some fluent by an amount of up to a certain maximum between 5-80%. We considered the two approaches described earlier: completing the original planning task and recovering only once a plan is found for the assumed initial state (at-the-end), followed by further `RegBasA*` search if needed, or reacting to state changes immediately (on-the-fly), pausing further `RegBasA*` expansion until `Recover` has brought the current search tree up-to-date. In both cases, several episodes of recovery and additional `RegBasA*` search were generally required before finding an optimal and up-to-date plan. Their number varied strongly, as a result of some discrepancies having larger impact than others. Table 5.1 shows the percentages of simulations in which an optimal plan was found, i.e., the algorithm converged within the time limit, for different frequencies and amounts of perturbation. As time limit we used 30 times the time required for solving the respective original planning problem without perturbations and using a conventional $A^*$ search planner. These were 0.52s for TPP1, 2.17s for TPP2, 3.03s for TPP3, and 0.34s, 0.82s, and 1.58s for Zenotravel 1, 2, and 3 respectively. The frequencies shown in the table are relative to these as well. For instance, the value 100 for Zenotravel1 on-the-fly, 5Hz, 40% states that even when every $0.34s/5 = 68$ milliseconds the value of a random fluent changed by up to 40% in the considered Zenotravel problem, the on-the-fly approach still converged 100% of the time. This simulates a quite erratic environment, possibly harsher than many practical application domains.

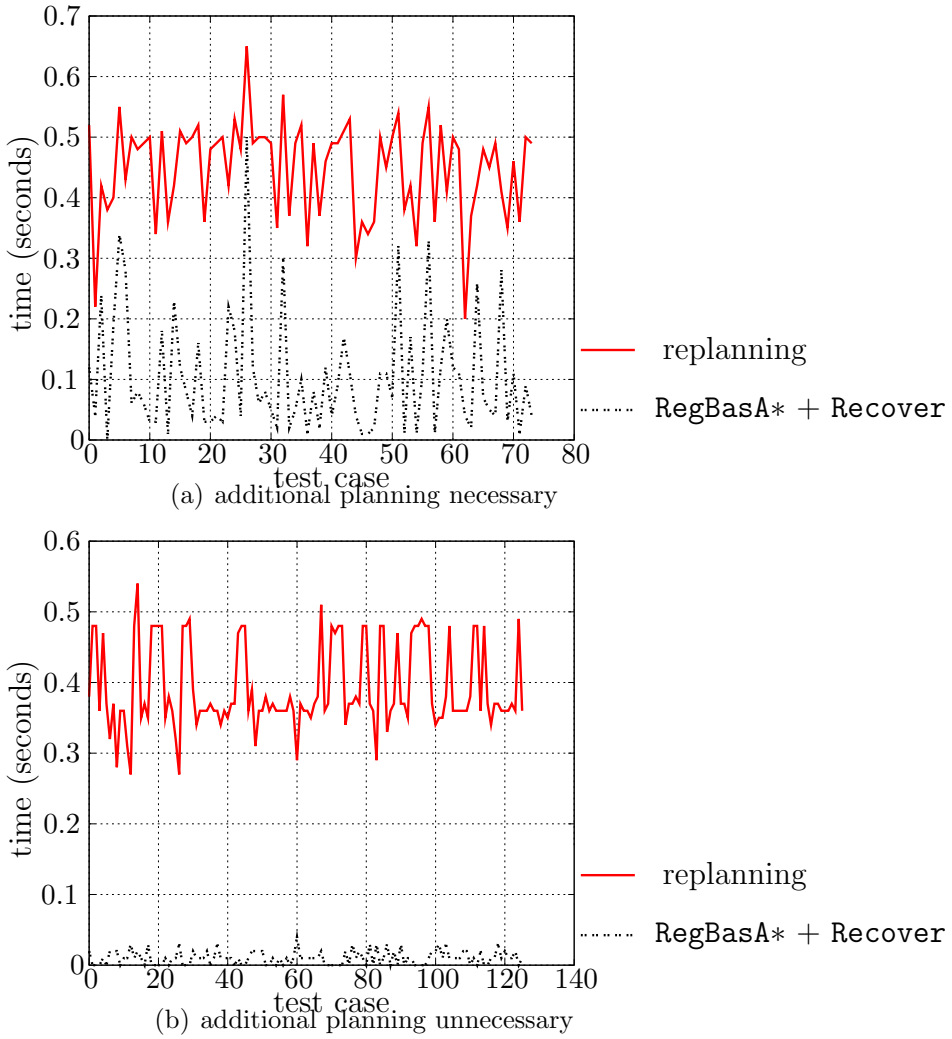
The on-the-fly recovery strategy clearly outperforms at-the-end recovery. This makes intuitive sense, as no time is wasted continuing plan generation for an already flawed instance. This also motivates an integrated approach like ours, showing its benefit over the use of plan adaptation approaches which are only applicable once a first plan has been produced (cf. Section 8.1). These approaches also generally do not guarantee optimal plans.

| Frequency: | 3Hz · planning time | | | | 5Hz · planning time | | | | 10Hz · planning time | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Deviation: | 5% | 20% | 40% | 80% | 5% | 20% | 40% | 80% | 5% | 20% | 40% | 80% |
| TPP1 at-the-end | 100 | 100 | 83 | 60 | 100 | 83 | 63 | 43 | 100 | 76 | 43 | 20 |
| TPP1 on-the-fly | 100 | 100 | **86** | **83** | 100 | **96** | **80** | **83** | 100 | **93** | **80** | **70** |
| TPP2 at-the-end | 96 | 60 | 63 | 43 | 100 | 51 | 44 | 34 | 89 | 34 | 24 | 10 |
| TPP2 on-the-fly | **100** | **86** | **86** | **83** | **96** | **75** | **86** | **82** | **96** | **86** | **79** | **82** |
| TPP3 at-the-end | 100 | 50 | 66 | 41 | 94 | 55 | 42 | 52 | 76 | 42 | 32 | 20 |
| TPP3 on-the-fly | 100 | **87** | **92** | **72** | **94** | **86** | **87** | **89** | **89** | **81** | **89** | **86** |
| Zenotravel1 at-the-end | 100 | 100 | 100 | 76 | 66 | 63 | 43 | 56 | 3 | 0 | 6 | 16 |
| Zenotravel1 on-the-fly | 100 | 100 | 100 | **100** | **96** | **100** | **100** | **86** | **66** | **70** | **93** | **93** |
| Zenotravel2 at-the-end | 66 | 30 | 26 | 3 | 30 | 6 | 6 | 6 | 10 | 0 | 0 | 0 |
| Zenotravel2 on-the-fly | **86** | **53** | **53** | **40** | **36** | **30** | **26** | **23** | **13** | 0 | **6** | **20** |
| Zenotravel3 at-the-end | 100 | 56 | 28 | 8 | **97** | 12 | 7 | 7 | 33 | 0 | 0 | 2 |
| Zenotravel3 on-the-fly | 100 | **60** | **56** | **66** | 90 | **60** | **30** | **43** | **43** | **33** | **25** | **21** |

Table 5.1: Percentage of test cases where our approach converged within the time limit, by event frequencies and deviation amounts. For better visibility, the bold faced numbers indicate which of the two usages (at-the-end, or on-the-fly) performed better.

The table also shows that convergence was much better on TPP than on Zenotravel. Interestingly, this was predictable given Figure 5.1: since the curve for Zenotravel1 intersects the y-axis at around 0.07 seconds, it seems unreasonable to expect convergence on this problem when the initial state changes at intervals shorter than that. This explains the low probability of convergence of the at-the-end approach when events occur at 10Hz times planning time, i.e., every 0.034s.

Since replanning from scratch takes the same amount of time, no matter how small the discrepancy is, assuming the problem does not get significantly easier through this, it has no chance of ever catching up with reality when events happen at time intervals shorter than the time required for plan generation. Our approach thus enables the application




(a) additional planning necessary

(b) additional planning unnecessary

Figure 5.2: Run-time comparison of our approach vs. replanning from scratch on the TPP domain.

of planning in domains where this was not previously possible. In practice, the approach needs to be paired with an equally effective execution monitor like the one we presented in the previous chapter of this thesis. During execution, a similar phenomenon to our convergence behavior can be observed. Intuitively, as plan execution proceeds and we are approaching the goal, the number of relevant domain features decreases, and so does the probability for relevant events—less things can go wrong. This benefits our relevance-based approach.

Not surprisingly, our approach generally outperforms replanning from scratch. To

demonstrate this, we compared the times required by both approaches for recovering from a single change of the initial state. The setup was as follows: We solved a planning problem, perturbed the state of the world by randomly changing some fluent's value, and then ran both (a) `Recover` followed by further `RegBasA*` search based on the modified open list if necessary, and (b) replanning from scratch using a conventional $A^*$ search implementation using the same heuristic. The fluent perturbations were done on continuous fluents only, and the amount of change was up to 50%.

Figure 5.2 shows the time both approaches require to recover from single events on our TPP1 problem. Recall that with both approaches the resulting plan is provably optimal. We separately show the times for cases where (a) additional `RegBasA*` was necessary, and (b) where it was not. The latter is the case when following `Recover`, the first element of the open list satisfied the goal. The average speed-up over replanning from scratch was 10.56 in the former case, and 33.64 in the latter. In all test cases, the simulated discrepancy was relevant in the sense that at least one formula appearing in the annotation was affected. Hence, calling `Recover` was necessary in all cases – the set $\Delta$ of Algorithm 5 was never empty.

We performed the same experiment on the Zenotravel1 problem. It is a reasonable question to ask whether the relative speed-up of our approach is just due to the use of a comparatively slow replanner, as it has been claimed about the empirical results of other plan repair approaches in the literature (cf. our discussion of the SPA system, page 142). Therefore we tested using two different, hand-coded heuristics, where the first is more informed (i.e., better) than the second. Using the first, which we also used in the earlier described experiments, the average recovery time was 0.14s, and the average replanning time was 0.51s, whereas with the second heuristic recovery time averaged to 0.35s and replanning to 1.07s. This shows that even when the planner, and thus replanner, is improved by the use of a better heuristic, our approach is still generally superior to replanning from scratch. This is because it equally benefits from a smaller search tree,

resulting from the use of a better heuristic.

## 5.4 Discussion

Generating optimal plans in dynamic environments where the assumed initial state of the problem frequently changes in unexpected ways is difficult. If such changes are ignored, no guarantees can be made about the optimality or even validity of the produced plan. But if one opts to replan from scratch every time such a change happens, the plan generation process may never terminate, when changes are frequent and regular.

In this chapter we made three contributions:

1. We presented a novel integrated plan generation and recovery algorithm for generating optimal plans in environments where the state of the world frequently changes unexpectedly during plan generation. At its core, the algorithm reasons about the relevance and impact of discrepancies, allowing the algorithm to recover from changes more efficiently than replanning from scratch.

2. We introduced a new criterion for evaluating plan adaptation approaches, called convergence, and argued for its significance.

3. We empirically demonstrated that our approach is able to converge even under high frequencies of unexpected state changes. Our experiments also show that an interleaved planning-and-recovery approach which recovers from such discrepancies on-the-fly is superior to an approach that only recovers once plan generation has completed.

Note that the convergence of our approach depends on two assumptions: (i) that, on average, over longer periods of time, more or more drastic changes to the state of the world occur, and (ii) that our algorithm is indeed able to recover from less drastic changes faster than from more drastic ones, again, on average. While it is hard to imagine environments

where the first assumption is violated, the satisfaction of the second assumption is hard to predict *a-priori*. We have hence opted to verify it empirically for the example domains we considered (cf. Figure 5.1 and its discussion).

The problem we address in this chapter has not received much principled attention in the literature. We review more abstractly related work in Chapter 8, and limit ourselves here to just a few brief remarks. To the best of our knowledge, [Veloso *et al.*, 1998] is the only other approach for monitoring the state of the world and reacting upon changes during plan generation. However, that approach applies only a very limited notion of optimality, where, in particular, the quality of sub-plans is known a-priori. Further the most characteristic distinction between ours and existing replanning approaches, which are devised for replanning only once a complete plan has been generated, and which often work by explicitly repairing that plan, is that:

1. our approach guaranteed optimal plans, while almost none of the other approaches do (with the notable exception of the SHERPA system [Koenig *et al.*, 2002] (discussed at length in Section 8.1, Page 146);

2. our approach is integrated with the planner, enabling replanning as soon as discrepancies occur, and not just after plan generation has completed.

There are several possible directions for future work. The approach needs to be tested in real-world applications for validation. It is hard to determine abstractly whether the simulated dynamics of the world are consistent with how real application domains may behave.

Further, the initially discussed question regarding the notion of optimality in a domain that may change in arbitrary ways deserves a more sophisticated, theoretical analysis. Here, insights from meta-reasoning come to mind, which we will review in Section 8.5. This in particular holds when removing the assumption that time does not affect the quality of the plan, which in many domains may be problematic.

In addition, we think that the ideas behind the presented approach may be beneficially applied to planning under initial state uncertainty as well, in particular when such uncertainty ranges over continuous domains.

# Chapter 6

# Monitoring Policy Execution in Stochastic Domains

While in the previous chapters we assumed that uncertainty in the environment cannot be predicted or quantified, we now consider the case when such quantification is possible to some extent. In particular, we consider stochastic domains, where certain actions can have any of a finite set of defined outcomes, and where the probabilities for these outcomes are known. Assuming that not all uncertainty is captured this way, it is still necessary to devise an execution monitoring strategy. We again follow the abstract approach of identifying what is relevant for the achievement of the objective—policy optimality in this case—and then devise an algorithm to determine the degree of relevance of unexpected state discrepancies, based on this information.

## 6.1 Introduction

The de-facto standard for modeling decision making in stochastic domains is Markov Decision Processes (MDPs). In most cases, MDPs are solved off-line by creating a policy that maps each state to an action that maximizes the expected accumulated reward over time. With this policy in hand, an agent knows how to act optimally in any state of the

world. Unfortunately, solving an MDP in such a way is computationally intensive and particularly difficult for large or infinite state spaces, or when time is limited. As such, a reasonable alternative is to compute an approximately optimal policy via state-based search from a known initial state. This generates a policy for (a subset of) the state space that is reachable from the current state within a bounded number of actions. Two questions arise from such an approach: how close to optimal is the resulting policy, and how robust is it.

In this chapter we address the issue of policy robustness. Since the MDP is only solved for a subset of the state space, an agent can find itself outside this subset during execution of the policy. This can happen for at least two reasons: i) the initial search ignored less likely outcomes of actions or ignored the possible occurrence of some unlikely exogenous events, or ii) the transition function erroneously neglected the possibility of certain outcomes or events. This is particularly true of sampling-based approaches, sometimes used to find an approximately optimal policy for a problem defined over a infinite state space (e.g., [Kearns *et al.*, 1999], also see Section 6.5.2). Regardless of the cause, a discrepancy between actual and anticipated outcomes requires the agent to decide how to proceed. A common response is to replan starting from the actual current state. We argue that in many cases of discrepancy detection it is not necessary to replan because the aspect of the state that is deviant has no bearing on the optimality of the remaining policy execution.

### 6.1.1 Contributions

To address this problem, we develop an approach for monitoring the execution of policies that forms yet another concrete application of the abstract monitoring approach we formulated in Chapter 3. In Section 6.2 we review the situation calculus representation of relational MDPs, and their solution using decision-tree search. Following the general scheme of using our abstract monitoring algorithm, in Section 6.3 we begin by defining

the required policy annotation, by identifying what is relevant for its optimality. We then present an algorithm that exploits this annotation to update the search tree and the value function estimate when discrepancies are observed during policy execution. The approach hence covers the state evaluation and the recovery steps of the framework of Figure 1.1, p.4, for the case of monitoring policy optimality in stochastic planning domains.

Thereafter we analyze properties of our approach including the space complexity of our annotation. We also consider two particular classes of MDPs: stochastic shortest-path problems and large or infinite MDPs that are solved approximately via sampling. We show that in both cases our algorithm can be used to verify the optimality of the current policy or to resurrect an approximately optimal policy together with an upper bound on the approximation error, under certain circumstances. Experiments with an implementation of our approach illustrate its potential to drastically speed-up the decision to continue execution of a policy or to replan.

## 6.2 Background

### 6.2.1 Representing MDPs

An MDP is described through a state space $\mathcal{S}$, a set of actions $A$, a transition function $T$, with $T(s, a, \cdot)$ denoting a distribution over $\mathcal{S}$ for all $s \in \mathcal{S}$, $a \in A$, a reward function $R : \mathcal{S} \to \mathbb{R}$, and a cost function $C : A \times \mathcal{S} \to \mathbb{R}$. The state is assumed to be fully-observable by an agent acting in such an environment and the agent's goal is to behave according to a policy $\pi : \mathcal{S} \to A$ that maximizes the *value function* defined using the infinite horizon, discounted reward criterion, defined in terms of the expectation $\mathcal{E}$:

$$V^{\pi}(s) = \mathcal{E}\left[\sum_{i=0}^{\infty} \gamma^i (r_i - c_i) \;\middle|\; s, \pi\right]$$

where $r_i$ is the reward obtained after performing policy $\pi$ for $i$ steps starting in $s$, $c_i$ is the cost incurred by the action performed at that stage, and $0 < \gamma < 1$ is the discount factor. Similarly, the *Q-function* is defined as

$$Q^{\pi}(a, s) = R(s) - C(a, s) + \gamma \mathcal{E}_{s' \sim T(s,a,\cdot)} [V^{\pi}(s')]$$

where the expectation $\mathcal{E}$ is over the transition probabilities $T(s, a, \cdot)$. Finally, the optimal value function and optimal Q-function are defined as $V^*(s) = \sup_{\pi} V^{\pi}(s)$ and $Q^*(a, s) = \sup_{\pi} Q^{\pi}(a, s)$ respectively. The optimal policy behaves greedily with respect to $Q^*$, i.e., $\pi^*(s) = \text{argmax}_a Q^*(a, s)$.

An MDP $\mathcal{M} = \langle \mathcal{S}, A, T, R, C \rangle$ can be represented through a basic action theory $\mathcal{D}$ in the situation calculus as follows (cf. [Reiter, 2001; Boutilier *et al.*, 2001]): Fluents describe the set of states relationally, thus $\mathcal{S}$ is the set of all, possibly infinite, combinations of fluent values. Further the user specifies for each (stochastic) action $a \in A$ a predicate $\mathit{Choice}(a, a'_i)$ describing a collection of primitive actions $a'_i$ that form the unique outcomes of executing $a$. Preconditions are defined for $a$, but successor state axioms are defined in terms of the primitive actions describing the outcomes. Using predicate $\mathit{Prob}(a'_i, a, p, s)$, the user specifies the probability $p$ for outcome $a'_i$ when $a$ is performed in situation $s$. *Prob* and *Choice* describe the transition probabilities $T$. The specification is completed by two more predicates: $\mathit{Reward}(r, s)$, describing the reward $r$ obtained in situation $s$, and $\mathit{Cost}(a, c, s)$, describing the cost $c$ of primitive action $a$ in $s$. The latter deviates slightly from the definitions used in [Reiter, 2001; Boutilier *et al.*, 2001] as it defines costs for the outcomes of stochastic actions rather than the stochastic actions themselves. This is slightly more expressive, as it allows to express uncertainty about action costs, and contains the alternative as a special case.

The relational representation is more efficient than simple state enumeration but more importantly it also allows us to regress preconditions, rewards, costs, and probabilities over actions. This is the key requirement for our algorithm.

### 6.2.2 Solving MDPs through Search

We assume we are given a decision-tree search planner, as described e.g., by Dearden and Boutilier [1994], which operates as follows. Starting with a search tree containing only one node labeled with situation $S_0$, describing the current state of the world, the planner works by repeatedly expanding nodes in the tree and adding their successors (Figure 6.1 shows an example tree). A node labeled with situation $s$, denoted $N[s]$ and called a *situation node*, has successors $N[a_i, s]$, labeled with actions $a_i \in A$, called *action nodes*. If $a_i$ is possible in $s$, i.e., $\mathcal{D} \models \mathit{Poss}(a_i, s)$, then $N[a_i, s]$ has successors labeled with the possible successor situations $do(a'_{i1}, s), \ldots, do(a'_{im}, s)$ where the $a'_{ij}$ are the outcomes (nature's choices) of action $a_i$ as defined by $\mathit{Choice}(a_i, a'_{ij})$. The edge between $N[a_i, s]$ and $N[do(a'_{ij}, s)]$ is denoted $E[a'_{ij}, s]$. Situation labeled nodes $N[s]$ have an associated reward $N[s].r$ as defined through $\mathit{Reward}(r, s)$, edges $E[a', s]$ denoting action outcomes $a'$ have associated costs $E[a', s].c$ and probabilities $E[a', s].pr$, defined by $\mathit{Cost}(a', c, s)$ and $\mathit{Prob}(a', a, pr, s)$ respectively. We make no assumptions regarding the expansion strategy of the planner, that is, how it chooses the next node to expand, nor about its cutoff criterion, used to determine when to stop expanding. But we do assume the existence of a predicate $\bar{V}(v, s)$ that provides a heuristic estimate $v$ of the value of the optimal value function ($V^*$) in situation $s$.[1] This heuristic is used to estimate the value in all leaf nodes.

Given a tree spanned this way, we can obtain a better estimate of the real value function for situations in the tree, by backing-up the values from the leaves to the root

[1] cf., e.g., [Dearden and Boutilier, 1994] for notes on how to obtain such a heuristic

using the standard update rules:

$$Q_i(a,s) = N[s].r + \sum_{a' \in successors(N[a,s])} E[a',s].pr\big(\gamma \cdot V_{i-1}(do(a',s)) - E[a',s].c\big)$$

$$V_i(s) = \begin{cases} \max_{a \in successors(N[s])} Q_i(a,s), & \text{if } i > 0; \\ v \;\text{ such that }\; \mathcal{D} \models \bar{V}(v,s), & \text{if } i = 0. \end{cases}$$

where we use $successors(N[s])$ and $successors(N[a,s])$ to informally denote the considered actions and considered action outcomes in the tree, respectively, and $i$ denotes the height in the tree, with the leaves being at $i = 0$. This value function estimate converges to the real value function as the search horizon increases, that is, the farther the search is performed, the closer the approximation will be to the real values. The best action to take in the initial situation $S_0$ according to this function is the greedy action $a^* = argmax_{a \in A} Q(a, S_0)$, and similarly for all subsequent actions. This produces a conditional plan representing the best (partial) policy according to this value function approximation, starting in $S_0$, of the form

$$\begin{aligned} \pi(S_0) = \quad & a; \\ & \textbf{if } \varphi(a'_1) \textbf{ then } \pi(do(a'_1, S_0)) \\ & \textbf{elseif } \varphi(a'_2) \textbf{ then } \pi(do(a'_2, S_0)) \\ & \vdots \\ & \textbf{elseif } \varphi(a'_m) \textbf{ then } \pi(do(a'_m, S_0)) \textbf{ fi} \end{aligned}$$

where $\varphi(a'_i)$ denotes a formula that holds if and only if the outcome $a'_i$ of $a$ has happened, and $\pi(do(a'_i, S_0))$ denotes the sub-policy for this outcome. Following the assumption of full-observability in the MDPs we consider, the formulae $\varphi(\cdot)$ are given and can always be evaluated.

But what if after performing action $a$ in $S_0$ we do not end in either of these situations $do(a'_i, S_0)$ but in some unexpected situation $S^*$, or after planning we observe that some

exogenous event has altered the world and we are no longer in $S_0$ but in $S^*$? In the highly-dynamic RoboCup domain, for instance, where real robots play soccer against each other, the robots' actions generally have infinitely many possible outcomes and the occurrence of exogenous events is practically impossible to predict. It seems one would need to perform time consuming replanning starting from $S^*$ in these cases. We argue that this can often be avoided, namely when the discrepancy between an expected state and the actual state is irrelevant to the remaining policy. To distinguish between relevant and irrelevant discrepancies, one can again apply the abstract monitoring approach proposed in Chapter 3, that is to regress the value function and other relevant information (including outcome probabilities) over the policy to derive a condition for its (near-)optimality in terms of the current situation. We assume that the planner again not only returns the policy, but also the search tree itself. This serves our approach and also enables the planner to further improve the remaining policy during execution.

## 6.3 Algorithm

Given a planner that provides a (near-)optimal policy according to the given heuristic function as described above starting from situation $S_0$, we assume, roughly, the general control flow of Algorithm 6, presented in pseudo-code. Recall that the policy $\pi$ is itself a tree. By $\pi(S)$ we denote the (sub-)policy rooted in the node corresponding to situation $S$. That is, first the planner generates a (near-) optimal policy and also returns the search tree. This policy is annotated by associating with each step an annotated copy of the corresponding sub-search tree (Section 6.3.1). Then the policy is executed until a termination condition is met, e.g., the goal is reached, if such a condition exists (cf. the condition of the **while**–loop). During execution the policy may be further improved by extending the current sub-search tree. However, if the actual current situation $S^*$ is unexpected and in particular not planned for, for reasons outlined above, the algorithm

**Algorithm 6**: General flow of control.

```
1  (π, T) ← plan(S_0) ;                                  // generate policy and search tree
2  π̂ ← annotate(π, T) ;                                  // annotate the policy
3  while termination condition not met do
4      if S* ∈ π̂ then          // the current situation has been planned for, continue
5          π̂ ← π̂(S*) ;
6      else                                              // the current situation was unexpected
7          choose S' ∈ π̂(S) ;
8          π̂ ← patch(π̂(S'), S*) ;                        // patch sub-tree from S' to S*
9      extend(π̂) ;                                       // (optional) plan further
10     execute-next-greedy-action(π̂) ;
```

first "patches" the policy from some (expected) situation $S'$ included in the original search tree to reflect this discrepancy (Section 6.3.2). We will later address the issue of how to choose $S'$, but intuitively it should be the result of one of the considered outcomes or $S$ itself. It turns out that patching a policy, by patching its associated sub-search tree and extracting a new greedy policy if necessary, can generally be done much faster than replanning from the new situation when exploiting both, the annotations we propose, and knowledge about the discrepancy itself. This is described in Section 6.3.2.

### 6.3.1 Annotation

We annotate each sub-policy with an annotated copy of the sub-search tree of the corresponding node of the overall search tree, $T$, returned by the planner. These annotated copies are described in Definition 8. Consider the example search tree of Figure 6.1. Recall that $N[s]$ denotes the (unique) node in the search tree labeled with situation $s$, $N[a, s]$ its successor node representing the execution of action $a$ in $s$, and $N[do(a'_i, s)]$ the node representing its $i^{th}$ outcome. $E[a'_i, s]$ denotes the edge between $N[a, s]$ and $N[do(a'_i, s)]$. In the figure, an example policy is indicated by bold lines. Our algorithm would, intuitively, annotate each of the situation nodes $N[S_0]$, $N[do(a'_{1,1}, S_0)]$, and $N[do(a'_{1,2}, S_0)]$, respectively with annotated copies of the respective sub-search trees: $\hat{T}_{S_0}$, $\hat{T}_{do(a'_{1,1},S_0)}$, and $\hat{T}_{do(a'_{1,2},S_0)}$, as defined below. Recall that for two situations $S$, $S'$, such

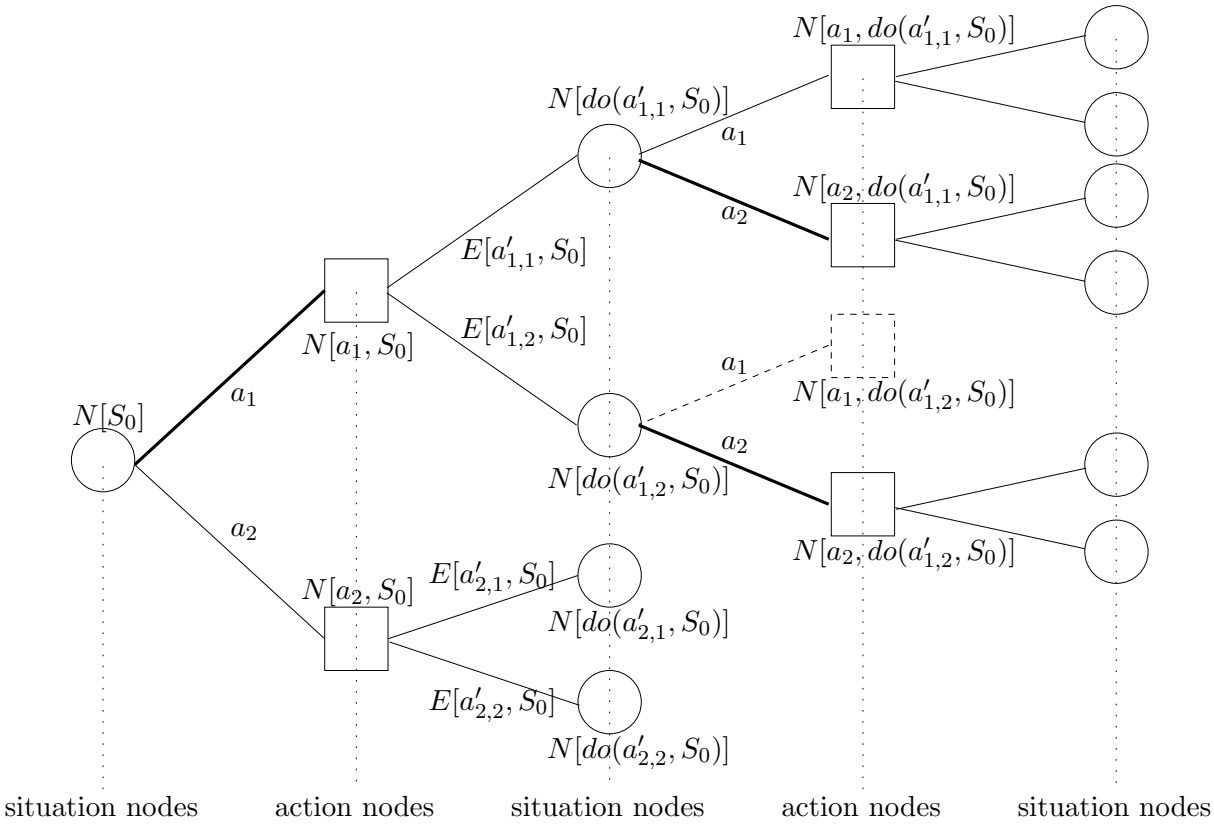



Figure 6.1: A sample search tree for initial situation $S_0$, in an environment with two actions ($a_1$, $a_2$), each of which has two possible outcomes ($a'_{1,1}/a'_{1,2}$, and $a'_{2,1}/a'_{2,2}$, resp.). Circles denote states/situation nodes, boxes denote nature's choices/action nodes. Dashed lines indicate infeasible actions. An example policy is indicated by bold lines.

that $S \sqsubset S'$, the expression $\mathcal{R}\big[\psi(S'), S\big]$ denotes the regression of $\psi(S')$ back to $S$. For instance, $\mathcal{R}\Big[\bar{V}\big(v, do(a'_{2,2}, do(a'_{1,2}, S_0))\big),\ do(a'_{1,2}, S_0)\Big]$ is defined as the one step regression over $a'_{2,2}$, and hence denotes a formulae specifying the heuristic estimate of the value function in the bottom-right node of the example search tree of Figure 6.1, in terms of fluents relativized to situation $do(a'_{1,2}, S_0)$, i.e., the situation reached after performing $a_1$ in the initial situation $S_0$, and observing outcome $a'_{1,2}$.

**Definition 8** (Annotated Search Tree)**.** Let $T_{S_0}$ be the search tree for an initial situation $S_0$, and let $S$ be any continuation of $S_0$, i.e., there is a situation node $N[S]$ in the search tree $T_{S_0}$. Let $T_S = (N_S, E_S)$ denote the sub-search tree rooted in that node. The

*annotated search tree* $\hat{T}_S = (\hat{N}_S, \hat{E}_S)$ for $T_S$ is defined as follows, where $S'$ denotes any continuation of $S$ appearing in $T_S$:

For each situation node $\hat{N}_S[S']$:

$$\left.\begin{array}{l} \hat{N}_S[S'].H(h,s) = \mathcal{R}\big[\bar{V}(h,S'),S\big] \\ \hat{N}_S[S'].h = \mathbf{h} \;\text{ such that }\; \mathcal{D} \models \bar{V}(\mathbf{h},S') \end{array}\right\} \quad \text{if it is a leaf node}$$

$$\left.\begin{array}{l} \hat{N}_S[S'].R(r,s) = \mathcal{R}\big[\mathit{Reward}(r,S'),S\big] \\ \hat{N}_S[S'].r = \mathbf{r} \;\text{ such that }\; \mathcal{D} \models \mathit{Reward}(\mathbf{r},S') \end{array}\right\} \quad \text{if it is not a leaf node}$$

For each action node $\hat{N}_S[a,S']$:

$$\hat{N}_S[a,S'].P(s) = \mathcal{R}\big[\mathit{Poss}(a,S'),S\big]$$

$$\hat{N}_S[a,S'].p = \begin{cases} 1 & \text{if } \mathcal{D} \models \mathit{Poss}(a,S') \\ 0 & \text{otherwise} \end{cases}$$

For each edge $\hat{E}_S[a',S']$:

$$\hat{E}_S[a',S'].C(c,s) = \mathcal{R}\big[\mathit{Cost}(a',c,S'),S\big]$$

$$\hat{E}_S[a',S'].c = \mathbf{c} \;\text{ such that }\; \mathcal{D} \models \mathit{Cost}(a',\mathbf{c},S')$$

$$\hat{E}_S[a',S'].Pr(p,s) = \mathcal{R}\big[\mathit{Prob}(a',a,p,S'),S\big]$$

$$\hat{E}_S[a',S'].pr = \mathbf{pr} \;\text{ such that }\; \mathcal{D} \models \mathit{Prob}(a',a,\mathbf{pr},S')$$

Nodes further have an associated value:

$$\hat{N}_S[S'].v = \begin{cases} \hat{N}_S[S'].h & \text{if leaf node} \\ \hat{N}_S[S'].r + \max_a(\hat{N}_S[a,S'].v) & \text{otherwise} \end{cases}$$

$$\hat{N}_S[a,S'].v = \sum_{a' \in \mathit{successors}(\hat{N}_S[a,S'])} \hat{E}_S[a',S'].pr(\gamma \cdot \hat{N}_S[\mathit{do}(a',S')].v - \hat{E}_S[a',S'].c)$$

where $a'$ are all the possible outcomes of action $a$, as defined by $\mathit{Choice}(a,a')$.

That is, we annotate each situation-node of the tree with the regression of the heuristic estimate of the real value function if it is a leaf node, or the regression of the reward

predicate otherwise. Action nodes are annotated with the regression of their preconditions, and edges with the regression of their respective cost- and probability-predicates. In all cases, the regression is performed back to the situation labeling the root of the tree, $S$, and their value according to $S$ is stored as well. The regressed predicates are all the information needed to determine the optimality of the policy given the search horizon, solely based on what is true in the current situation. This allows for greatly improved reevaluation if on-line we find ourselves in the unexpected situation $S^*$ rather than $S$: *Only those annotated conditions that mention fluents that are affected by the discrepancy between $S^*$ and $S$ need to be reevaluated.* Given that most discrepancies only affect a fraction of all fluents, this saves a lot of computation. After reevaluating the affected conditions, the values of Q-function and value function can be updated (back-up). This on-line behavior is subject of the next section.

Note the correspondence between $\hat{N}_S[S'].v$ and the value function $V(S')$, and between $\hat{N}_S[a, S'].v$ and the Q-function $Q(a, S')$.

### 6.3.2 Execution Monitoring

Assume we have executed a (possibly empty) prefix $a_1, a_2, \ldots, a_k$ of an earlier, in situation $S_0$, generated policy and the outcomes of these actions were $a'_{1i_1}, a'_{2i_2}, \ldots, a'_{ki_k}$ so that we expect to be in situation $S = do([a'_{1i_1}, a'_{2i_2}, \ldots, a'_{ki_k}], S_0)$, but in fact find ourselves in a different situation $S^*$ that the current search tree and policy do not account for. The naive solution is to either fall back to greedy behavior according to the heuristic estimate of the value function or to replan from $S^*$. We here propose a better solution, "patching" the search tree to reflect all values with respect to $S^*$ in place of $S$. We will do this as outlined earlier, by reevaluating all conditions that are affected by the discrepancy and backing-up the altered values to also update the Q-function and value function values.

Let *fluents*$(\hat{T}_S)$ denote the set of ground fluents[2] occurring in any of the regressed

[2] As in the previous chapters, we assume a finite set of ground fluents, cf. our discussion on page 51.

formulae appearing in the annotated tree rooted in the expected situation $S$, and let $\Delta_F(S, S^*)$ be those such fluents whose truth values differ between $S$ and $S^*$, i.e.,

$$\Delta_F(S, S^*) \stackrel{\text{def}}{=} \left\{ F(\vec{X}) \;\middle|\; F \in \mathit{fluents}(\hat{T}_S) \text{ and } \mathcal{D} \models F(\vec{X}, S) \not\equiv F(\vec{X}, S^*) \right\}.$$

Only conditions mentioning any of these fluents need to be reevaluated, all others remain unaffected by the discrepancy.

**Function** `redoSit`$(\hat{N}_S[S'],\ \hat{T}_S,\ S^*)$

```
Φ ← successors(N̂_S[S']);
if Φ = ∅ then                                                    // if leaf node
    if fluents(N̂_S[S'].H(h, s)) ∩ Δ_F(S, S*) ≠ ∅ then             // verify heuristic value
        N̂_S[S'].h ← h   such that D ⊨ N̂_S[S'].H(h, S*);
        N̂_S[S'].v ← N̂_S[S'].h
else
    foreach a ∈ Φ do                                             // recurse over all actions
        redoAction(N̂_S[a, S'], T̂_S)
    q_max ← max_{a∈Φ}(N̂_S[a, S'].v);
    if fluents(N̂_S[S'].R(r, s)) ∩ Δ_F(S, S*) ≠ ∅ then             // verify reward
        N̂_S[S'].r ← r   such that D ⊨ N̂_S[S'].R(r, S*)
    N̂_S[S'].v ← N̂_S[S'].r + q_max ;                              // update value
```

**Function** `redoAction`$(\hat{N}_S[a, S'], \hat{T}_S, S^*)$

```
if fluents(N̂_S[a, S'].P(S')) ∩ Δ_F(S, S*) ≠ ∅ then               // verify feasibility of action
    if D ⊨ N̂_S[a, S'].P(S*) then
        N̂_S[a, S'].p ← 1;
    else
        N̂_S[a, S'].p ← 0;
if N̂_S[a, S'].p = 1 then                                         // if it is possible
    Φ ← successors(N̂_S[a, S'], T̂_S);
    foreach a' ∈ Φ do                                            // recurse over all outcomes
        redoOutcome(Ê_S[a', S'])
    // update value using update rule
    N̂_S[a, S'].v ← Σ_{a'} Ê_S[a', S'].pr · (γ N̂_S[do(a', S')].v − Ê_S[a', S'].c)
else
    N̂_S[a, S'].v ← −∞;
```

**Function** `redoOutcome`( $\hat{E}_S[a', S'], \hat{T}_S), S^*$ )

1 **if** *fluents*$\big(\hat{E}_S[a', S'].C(c, S')\big) \cap \Delta_F(S, S^*) \neq \emptyset$ **then** *// verify costs*
2 $\quad \hat{E}_S[a', S'].c \leftarrow \mathbf{c}$ such that $\mathcal{D} \models \hat{E}_S[a', S'].C(\mathbf{c}, S^*)$
3 **if** *fluents*$\big(\hat{E}_S[a', S'].Pr(pr, S')\big) \cap \Delta_F(S, S^*) \neq \emptyset$ **then** *// verify outcome probability*
4 $\quad \hat{E}_S[a', S'].pr \leftarrow \mathbf{pr}$ such that $\mathcal{D} \models \hat{E}_S[a', S'].Pr(\mathbf{p}, S^*)$
5 redoSit( $\hat{N}_S[do(a', S')], \hat{T}_S)$ ; *// recursively update resulting situation node*

The function `redoSit` together with helper functions `redoAction` and `redoOutcome` implements the patching, using the node and edge annotation as defined above. The call `redoSit`$(\hat{N}_S[S], \hat{T}_S, S^*)$ patches the tree $\hat{T}_S$ from $S$ to situation $S^*$, possibly making a different action the best (greedy) choice, in which case the annotation needs to be redone.

These functions implement a rather straightforward way of reevaluating relevant formulae in the search tree: They traverse the search tree in a depth-first manner, and check for each annotated formula whether it might be affected by the discrepancy. This is accomplished by checking whether any of the affected ground fluents unifies with any fluent appearing in the formula. If so, the formula is reevaluated and the values annotated in the tree are updated as necessary. In the traversal, the function `redoSit` is used to process situation nodes, `redoAction` processes action nodes, and `redoOutcome` processes the edges going out of action nodes and denoting possible action outcomes.

More sophisticated implementations of achieving the necessary updates can be thought of. For instance, when only very few formulae are affected by the discrepancy, it is more efficient to only reevaluate these, and then propagate potentially changed values up the tree, as necessary.

**Proposition 2.** By construction of the algorithm we have that after calling `redoSit`$(\hat{N}_S[S], \hat{T}_S, S^*)$, all annotated costs, rewards, probabilities, and heuristic values are with respect to $S^*$ instead of $S$, i.e.,

- in all situation nodes, $\mathcal{D} \models \bar{V}(\hat{N}_S[do(\vec{a'}, S)].v, do(\vec{a'}, S^*))$ if it is a leaf node, and $\mathcal{D} \models \mathit{Reward}(\hat{N}_S[do(\vec{a'}, S)].r, do(\vec{a'}, S^*))$ otherwise;

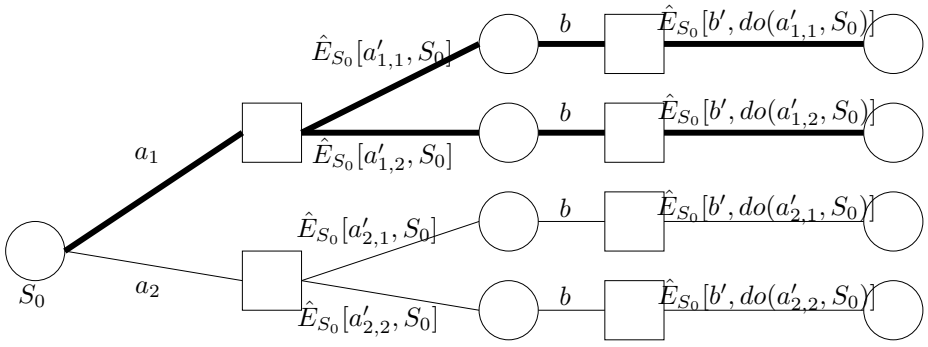



Figure 6.2: The annotated search tree for the root node of the example ($\hat{T}_{S_0}$). Circles denote states, boxes denote nature's choices. The policy is marked by bold lines. For readability we abbreviate: $a_1 = Drive(Market1)$, $a_2 = Drive(Market2)$, $a'_{1,1} = Drive10(Market1)$, $a'_{1,2} = Drive12(Market1)$, $a'_{2,1} = Drive10(Market2)$, $a'_{2,2} = Drive12(Market2)$, $b = BuyAllNeeded$, $b' = BuyAllNeededSuccess$.

- in action nodes, $\hat{N}_S[a, do(\vec{a'}, S)].p = 1$ if $\mathcal{D} \models Poss(a, do(\vec{a'}, S^*)$ and $\hat{N}_S[a, do(\vec{a'}, S)].p = 0$ otherwise; and
- in edges, $\mathcal{D} \models Cost\big(a'', \hat{E}_S[a'', do(\vec{a'}, S)].c, do(\vec{a'}, S^*)\big)$, and $\mathcal{D} \models Prob\big(a'', a, \hat{E}_S[a'', do(\vec{a'}, S)].pr, do(\vec{a'}, S^*)\big)$.

Further, also all annotated node values ($\hat{N}_S[s].v, \hat{N}_S[a, s].v$) are as defined in Definition 8.

We therefore refer to the resulting annotated search tree as $\hat{T}_{S^*}$.

## 6.4 An Illustrative Example

Consider the following simplified example from a modification of the previously introduced TPP domain, where an agent drives to various markets to purchase goods which she then brings to the depot (cf. Section A.1). In our modification of this domain the drive action is stochastic. For simplicity, assume there is only one kind of good, two

markets, and the following fluents: in situation $s$, $at(s) = l$ denotes the current location $l$, $request(s) = q$ represents the number $q$ requested of the good, $price(m, s) = p$ denotes the price $p$ of the good on market $m$, and $driveCost(src, dest, s) = c$ the cost $c$ normally incurred by driving from $src$ to $dest$. Let there be two actions: $Drive(dest)$ moves the agent from the current location to $dest$, and $BuyAllNeeded$ purchases the requested number of goods at the current (market) location. The drive action is stochastic and may result in one of two outcomes $Drive10(dest)$, and $Drive12(dest)$, where the only difference is that the latter incurs a cost of 1.2 times the drive cost specified by $driveCost(src, dest, s)$, whereas the former only incurs the normal cost (factor 1.0). This could, e.g., represent the risk of a traffic jam on the route. The buying action, on the other hand, always succeeds, represented by a single possible outcome $BuyAllNeededSuccess$.

Assume the planner has determined the plan $\vec{\alpha} = [Drive(Market1), BuyAllNeeded]$ to be optimal, but has as well considered $\vec{\beta} = [Drive(Market2), BuyAllNeeded]$ as one alternative among others. Note that we here simplified the representation of the plan by collapsing the two possible outcomes of drive actions into one straight line plan, as opposed to a conditional plan where the conditions are over the possible outcomes of the drive actions. This is possible, because the plan-suffixes for the two outcomes are identical in our example. For simplicity we also assume no rewards, i.e., $\mathcal{D} \models (\forall s).Reward(0, s)$. The search tree resulting from this problem is shown in Figure 6.2. For the first step of the policy, we annotate the search tree $T_{S_0}$ as follows, where for parsimony we ignore preconditions, rewards, and the heuristic value function used for evaluating leaf nodes. Recall the annotation of edges in the tree $\hat{E}_S[a', S'].C(c, s)$, $\hat{E}_S[a', S'].c$, $\hat{E}_S[a', S'].Pr(pr, s)$, and $\hat{E}_S[a', S'].pr$. We here only consider the annotated search tree for $S = S_0$.

$\hat{E}_{S_0}[Drive10(Market1), S']:$
$\quad .C(c,s) = (c = driveCost(at(S'), Market1, S'))$ $\quad .c = 381$
$\quad .Pr(pr,s) = (Prob(Drive10(Market1), Drive(Market1), pr, S'))$ $\quad .pr = 0.8$
$\hat{E}_{S_0}[Drive12(Market1), S']:$
$\quad .C(c.s) = (c = driveCost(at(S'), Market1, S') \cdot 1.2)$ $\quad .c = 457.2$
$\quad .Pr(pr,s) = (Prob(Drive12(Market1), Drive(Market1), pr, S'))$ $\quad .pr = 0.2$
$\hat{E}_{S_0}[Drive10(Market2), S']:$
$\quad .C(c,s) = (c = driveCost(at(S'), Market2, S'))$ $\quad .c = 458$
$\quad .Pr(pr,s) = (Prob(Drive10(Market2), Drive(Market2), pr, S'))$ $\quad .pr = 0.8$
$\hat{E}_{S_0}[Drive12(Market2), S']:$
$\quad .C(c,s) = (c = driveCost(at(S'), Market2, S') \cdot 1.2)$ $\quad .c = 549.6$
$\quad .Pr(pr,s) = (Prob(Drive12(Market2), Drive(Market2), pr, S'))$ $\quad .pr = 0.2$
$\hat{E}_{S_0}[BuyAllNeededSuccess, do(Drive10(Market1), S')]:$
$\quad .C(c,s) = (p = price(Market1, S') \cdot requested(S'))$ $\quad .c = 17$
$\quad .Pr(pr,s) = (pr = 1.0)$ $\quad .pr = 1.0$
$\hat{E}_{S_0}[BuyAllNeededSuccess, do(Drive12(Market1), S')]:$
$\quad .C(c,s) = (p = price(Market1, S') \cdot requested(S'))$ $\quad .c = 17$
$\quad .Pr(pr,s) = (pr = 1.0)$ $\quad .pr = 1.0$
$\hat{E}_{S_0}[BuyAllNeededSuccess, do(Drive10(Market2), S')]:$
$\quad .C(c,s) = (p = price(Market2, S') \cdot requested(S'))$ $\quad .c = 14$
$\quad .Pr(pr,s) = (pr = 1.0)$ $\quad .pr = 1.0$
$\hat{E}_{S_0}[BuyAllNeededSuccess, do(Drive12(Market2), S')]:$
$\quad .C(c,s) = (p = price(Market2, S') \cdot requested(S'))$ $\quad .c = 14$
$\quad .Pr(pr,s) = (pr = 1.0)$ $\quad .pr = 1.0$

Let us assume that even before we begin the execution of the plan, a discrepancy in form of an exogenous action $e$ happens, putting us in situation $S^* = do(e, S_0)$ instead of $S_0$. How does this affect the relevant values in the search tree and, as a consequence, the optimal policy? This clearly depends on the effects of $e$. If $e$ does not affect any of the fluents occurring in above annotated formulae, it can be ignored, the plan is guaranteed to remain optimal. This would, for instance, be the case when $e$ represents the event of a price change on a market not considered (e.g., $price(Market3)$), as that price does not appear in the regressed formulae, which only mention relevant fluents.

Consider, on the other hand, the case where $e$ represents the event of an increased demand, that is, increasing the value of $request(s)$ (i.e., $\mathcal{D} \models request(S^*) > request(S_0)$). Then we need to reevaluate all conditions that mention this fluent, that is, in our example, the costs of all occurrences of the $BuyAllNeededSuccess$ action. Afterwards any value that has changed needs to be propagated up the tree, in order to determine whether the

currently best policy has changed.

As a second example, imagine that the event $e$ represents an update on the traffic situation between Depot and Market1, stating that the risk of a traffic jam has increased to 0.5, i.e.,

$$\mathcal{D} \models Prob(Drive12(Market1), Drive(Market1), 0.5, S^*)$$
$$\wedge\, Prob(Drive10(Market1), Drive(Market1), 0.5, S^*).$$

Then we only need to recompute the backup values for the upper two branches without reevaluating any predicates except for these probabilities. After the backup, the currently best policy can again be greedily read off the tree.

## 6.5 Analysis

Three questions come to mind when trying to evaluate our approach. First, is the annotation of manageable size and does it scale? Second, what kind of guarantees can we make about the quality of the value function (approximation) derived from the resurrected search tree? Last but not least, how does the approach perform compared to simple replanning from scratch, does it offer computational savings? We here address these questions in this order.

### 6.5.1 Space Complexity

The space complexity of our approach is determined by the space required to store the annotation. This looks dramatic at first, as we annotate each action of the policy with a copy of the corresponding sub-search tree. Let there be $n$ actions ($n > 1$), let $m$ be the maximal number of outcomes an action has, and let $h$ be the maximal depth of the search tree. In the worst case, the tree is uniformly expanded to depth $h$, each action is possible in each situation-node, and each action has $m$ outcomes. Then the search tree

has size $(nm)^h$. It turns out that the annotation is not significantly larger and is, in fact, only constantly larger in $h$, despite the number of sub-tree copies at each subsequent level of the policy.

**Theorem 6** (Annotation Size)**.** If the search tree has size $(nm)^h$, then the size of the annotation is in $\mathcal{O}(\lambda(nm)^h)$ with $\lambda = \frac{n-\frac{1}{n^h}}{n-1}$.

*Proof:* Recall the format of the annotated policy from Section 6.3.1. In the root, $\hat{T}_S$ has size $c(nm)^h$ for some constant $c$ reflecting the per node annotation. We annotate each of the (at most) $m$ outcomes $a'$ of $a$ with a copy of the respective sub-search tree of the remaining horizon $h-1$, in total $cm(nm)^{h-1}$. This goes on for $h$ steps, eventually annotating $m^h$ leaves with a constant size value. In total we annotate

$$\begin{aligned} & c(nm)^h + mc(nm)^{h-1} + \cdots + m^h c(nm)^{h-h} \\ = \; & c(nm)^h(1 + \frac{1}{n} + \cdots + \frac{1}{n^h}) = c(nm)^h \frac{n - \frac{1}{n^h}}{n-1} \end{aligned}$$

□

Because $\lim_{n\to\infty} \lambda = 1$, this means that we generally have no space requirements above those required to store the search tree in the first place.

### 6.5.2 Optimality Considerations

What can we claim about the quality of the resurrected value function we obtain after patching a tree from situation $S$ to $S^*$? This depends on the usage, i.e., the problem being solved and the type of planner used. In this section we do the analysis for two classes of MDPs.

**Stochastic Shortest-Path Problems** A stochastic shortest-path problem [Bertsekas, 1995], is an MDP where all rewards are zero, all costs are positive, the discount factor $\gamma$ is equal to 1, and there is a distinguished set of *goal states* $g \in G$ that

are absorbing, i.e., $T(g, a, g) = 1$ for all actions $a \in A$, and $C(a, g) = 0$. In this type of problem, an optimal policy is one that minimizes the expected costs of actions performed until reaching a goal state.

In our formalization, assume the task is to devise an optimal policy for reaching a goal in a stochastic shortest-path problem, starting from the initial situation $S_0$, and that we are given a monotonic heuristic function.

Assume further that the planner has found an optimal policy that reaches the goal with probability one. Then, our algorithm can be used to verify the continued optimality of this policy in cases where an unexpected state is reached.

**Theorem 7.** An optimal policy $\pi(S_0)$ for a stochastic shortest-path problem starting from situation $S_0$ and annotated with $\hat{T}_{S_0}$ as in Definition 8, continues to be optimal in $S^*$ if after calling `redoSit`$(\hat{N}_{S_0}[S_0], \hat{T}_{S_0}, S^*)$, the greedy policy for $\hat{T}_{S^*}$ coincides with $\pi(S_0)$ and still reaches the goal with probability one.

The theorem follows from the monotonicity of the heuristic function and Proposition 2.

Recall the tree representation we used to represent policies. These do not refer to actual states, but only state the actions to perform and use explicit conditions to test which outcome of an executed stochastic action has actually occurred.

**Sampling in Large MDPs** Kearns *et al.* [1999] show that the time required to compute a near-optimal action from any particular state in an infinite horizon MDP with discounted rewards does not depend on the size of the state space. This is significant, as it allows, at least theoretically, for the computation of near-optimal actions even in MDPs with infinite state spaces. The authors propose a sampling based decision tree search algorithm denoted **A** and prove bounds on how many samples $C$ and what horizon $H$ is necessary in order for this algorithm to determine an $\epsilon$-optimal action. The algorithm works like the one described in Section 6.2.2, but only explores $C$ outcomes from the set

of outcomes of each action, where the samples are chosen according to the probability distribution over the outcomes. The bounds $C$ and $H$ only depend on $\epsilon$, the discounting factor $\gamma$, and a bound $R_{max}$ on the absolute value of the reward function. The bound on the horizon is $H = \lceil \log_\gamma(\epsilon(1-\gamma)^3/(4R_{max})) \rceil$. In the terminology of this chapter, the algorithm runs in the initial situation $S$, returning a best action $a$. After executing $a$ and observing the outcome $a'_i$, the algorithm needs to run again in $do(a'_i, S)$, because (i) the error-bound of the value function approximation for node $do(a'_i, S)$ for any outcome $a'_j$ of $a$ is greater than $\epsilon$, but more severely (ii) it is unlikely that the actual action outcome $a'_i$ is among the $C$ samples considered in planning.

The latter issue can in many cases be accounted for using our approach. Simple term manipulation of the above horizon bound yields the error-bound for considered successor situation $do(a'_j, S)$ of the initial situation as $\epsilon' \leq 4\gamma^{H-1}R_{max}/(1-\gamma)^3$ (since $C$ remains unchanged). This error bound on the value function can be resurrected by applying our algorithm in cases where neither action outcome probabilities nor preconditions have been affected by the discrepancy.

**Theorem 8.** Let $T_S$ be the search tree as spanned by the algorithm **A** [Kearns *et al.*, 1999] with horizon $H$ and sample width $C$ from initial situation $S$, $\pi(S)$ be the greedy policy extracted from $T_S$ with best first action $a$, and $\hat{T}_S$ be the annotated search tree (cf. Definition 8). Let the execution of $a$ yield the actual situation $S^*$. If there is a node $\hat{N}_S[do(a'_j, S)]$ in $\hat{T}_S$ such that $\mathcal{D} \models \varphi(do(a'_j, S)) \equiv \varphi(S^*)$ for all $\varphi$ such that $\varphi$ is a regressed precondition or regressed probability predicate in the annotated search tree $\hat{T}_{do(a'_j,S)}$, then, after calling `redoSit`$(\hat{N}_S[do(a'_j, S)], \hat{T}_{do(a'_j,S)}, S^*)$ the value function $V'$ described by $\hat{T}_{S^*}$ is such that $|V'(S^*) - V^*(S^*)| \leq 4\gamma^{H-1}R_{max}/(1-\gamma)^3$.

*Proof:* Since no preconditions or probabilities have changed, Proposition 2 provides that the updated tree $\hat{T}_{S^*}$ is one of the possible trees that would be created by running the algorithm **A** starting in $S^*$ with sample width $C$ and horizon $H-1$. Further, since $\gamma < 1$, it is obvious from the term manipulation above that $\epsilon' > \epsilon$. Hence, following Theorem 1

of [Kearns *et al.*, 1999], the sampling width $C'$ required to obtain such an error bound is less or equal to $C$. Thus, the error bound obtained by running the algorithm in $S^*$ with sample width $C$ and horizon $H-1$ is less or equal to $\epsilon'$. □

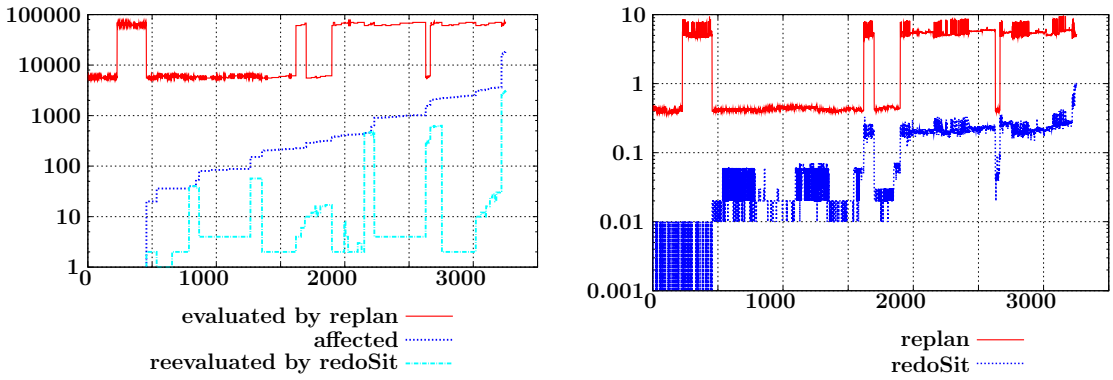
The requirement that none of the preconditions or outcome probabilities have changed is required to ensure that the initial choice of samples is still representative of the probability distributions after the discrepancies. If not, the error bound shown by Kearns *et al.* would not necessarily apply anymore to the patched successor state.

The result is particularly useful when gaging the relevance of exogenous events. If situation $S$ is expected but the actual situation $do(e, S)$ for some exogenous event $e$ is observed, then, if $e$ does not affect any preconditions or action probabilities, `redoSit`$(\hat{N}_S[S], \hat{T}_S, do(e, S))$ resurrects the approximation quality of the current policy for the new situation.

### 6.5.3 Empirical Results

We were interested in determining whether the approach was time-effective—whether the discrepancy-based incremental reevaluation of the search tree could indeed be done more quickly than simply replanning when a discrepancy was detected. The intuition was that most discrepancies only affect a small subset of all fluents and that this effect often does not propagate to relevant values. This intuition is supported practically by the experimental results we present next.

We compared a preliminary implementation of our `redoSit` algorithm to replanning from scratch on different problems in the variant of the metric `TPP` domain described above (where we changed some actions to have stochastic outcomes). In each case, we uniformly spanned the search tree up to a particular horizon, perturbed the state of the world by changing some fluent, and then ran both `redoSit` and replanning from scratch. To maximize objectivity, the perturbations were done systematically by multiplying the value of one of the numeric fluents (driving costs between locations, prices and number of




(a) # of conditions

(b) running time in seconds

Figure 6.3: Replanning vs. `redoSit`. To enhance readability we ordered the test cases by the number of affected conditions in all graphs.

goods on sale on different markets, and number of requested goods) by a factor between 0.5 and 1.5 (step-size 0.1) or by redistributing 0.5 of the probability mass of some action's outcomes. In total we tested on 3256 cases. All experiments were run on an Intel Xeon, 2.6GHz, with 1GB RAM. Figure 6.3(a) shows from top to bottom: the number of relevant conditions as evaluated by replanning, the number of actually affected conditions, and the number of such conditions that are unique. The latter is the number of conditions that actually need to be reevaluated. Values for all other affected conditions, effectively copies of earlier seen ones, can be obtained from a cache of evaluated conditions. The number of unique affected conditions is extremely low. On average this number was 9751 times lower than the number of relevant conditions. This also reflects in the running-time of our preliminary implementation (Figure 6.3(b)), and strongly motivates a patching approach over replanning.

## 6.6 Discussion

We have shown how the abstract monitoring approach proposed in this thesis can be applied to decision theoretic planning in relational MDPs, monitoring the adequacy and

potential optimality of a policy. The approach works by annotating the policy with conditions regressed to the situations where they are relevant. In so doing, the discrepancy between an unplanned-for state and an expected state can be evaluated with respect to the objective of an optimal (or near-optimal) policy. The intuition that this can be significantly more efficient than replanning, should something unexpected happen, is supported by experiments with a preliminary implementation.

We have pointed out two concrete applications. The first is in solving stochastic-shortest path problems optimally given a monotonic heuristic estimate of the value function, where our approach can provide for a sufficient condition for continued optimality of the policy. In the second, near-optimal policies for large, infinite horizon MDPs were considered. Here, our approach may be used to resurrect a value function approximation for which we can show an error bound, when otherwise replanning is the only option.

In future work we intend to perform further theoretical analysis of our approach, and to run more experiments, in particular in continuous domains.

While so far, we have focused on the question of whether a given policy remains optimal, or nearly so, it would be interesting to consider replanning as well. As demonstrated in the previous chapter for the case of deterministic planning, it is reasonable to expect that the computed annotations can be used to speed-up replanning as well. In the case of the current chapter, this would amount to further extending the updated search tree, until the termination criterion is once again met (e.g., a goal is reached with probability one, or a desired approximation error has been achieved).

In particular with respect to the latter, the commonalities and differences of our approach and other symbolic techniques for solving MDPs (cf. Section 8.2), should be investigated further.

Other methods for creating robust policies under time constraints include the work by Dean *et al.* [1995]. The authors consider finite horizon MDPs with an explicit goal area and assume there is a method for creating some path with positive probability from

the initial state to the goal. They propose to then, time permitting, extend this path to an *envelope* of states by successively including additional states that are on the fringe of the current envelope, i.e., possible successors of certain actions when executed in a state inside the envelope. This way, successively more contingencies are added to the policy. This approach differs from the one we presented in this chapter, as it relies on knowledge about an accurate model of possible contingencies and their probabilities and thus does not increase robustness against completely unforeseen courses of events.

Also Real-Time Dynamic Programming (RTDP) [Barto *et al.*, 1995] addresses the problem of incrementally creating a policy for the accessible region of state space given an initial state. RTDP works by interleaving value function estimate improvement and execution. The action executed is always the greedy one according to the current estimate. Interleaved with execution, trial-based look-ahead search from the current state is performed and the values of visited states backed-up accordingly. As with Dean et al.'s approach, also RTDP relies on an accurate model of the possible contingencies.

The most characteristic distinction between our algorithm and these is our use of a relational representation together with a combination of forward expansion and regression reasoning. This constitutes a middle-ground between regression based approaches that compute complete policies (see Page 155), and the non-relational forward search based approaches above.

# Chapter 7

# Generating and Executing Plans with Procedural Control

So far we have considered planning problems with conventional final state goals, or state-based rewards only. In practice, however, it is often desirable to be able to enforce a particular temporally extended behavior of the agent. Such specifications could regard safety regulations, preferred ways of achieving goals, industrial best practices, prescribed roles in a multi-agent setting, etc. While there are languages for specifying such constraints, and algorithms for generating plans that comply with them (e.g., [Levesque *et al.*, 1997; De Giacomo *et al.*, 2000; Bacchus and Kabanza, 1998]), it is not obvious how the continued satisfaction of these constraints can be monitored during execution. We address this problem in this chapter, by providing a compilation of planning problems with such control information into classical planning problems without, and which are hence in the right input format for the approaches presented earlier. Different from the previous chapters, this chapter does not contribute to any of the bold faced steps of the framework of Figure 1.1 (p. 4), but provides the means for pre-processing the input to the planning problem, making all previously developed approaches applicable to a larger set of problems.

## 7.1 Introduction

ConGolog [De Giacomo *et al.*, 2000] is a logical programming language for specifying high-level agent control, that is defined in the situation calculus. It extends the agent programming language Golog [Levesque *et al.*, 1997] by concurrent program execution. Golog's Algol-inspired programming constructs allow a user to program an agent's behavior while leaving parts of the program under-constrained, or "open", through the use of non-deterministic constructs. These under-constrained regions of the program are later filled in by a planner. Such integration of planning and programming has proved useful in a variety of diverse applications including soccer playing robots [Ferrein *et al.*, 2004], museum tour-guide robots [Burgard *et al.*, 1999], and Web service composition [McIlraith and Son, 2002].

By way of illustration, consider a simple delivery problem in which we have an (infinite capacity) truck and the task is to deliver packages from point A to point B. A classical planning problem would simply specify the initial state and the goal state. Using ConGolog, we can provide the following program that constrains the space of possible plans, while still leaving some work to the planner:

> *If not at point A, drive the truck to point A; while there are packages at point A, pick a package and load it onto the truck; drive to point B; while there are packages on the truck, pick a package and unload it from the truck.*

A basic action theory of the situation calculus induces a tree of possible action sequences or situations. A ConGolog program further constrains the tree to those that adhere to the program. However, in order to reason about the satisfaction of these constraints, a system requires special-purpose machinery – it needs to interpret the ConGolog program. This for instance applies to planning, but also to other problems that involve reasoning about feasible trajectories, including the problem of monitoring the continued validity of a plan. As we have already argued on Page 29, also the execution of programs

– or rather the plans resulting from the interpretation of programs – requires monitoring, since also the satisfaction of the hard constraints that are implicitly represented in the program need to be verified during execution, when unexpected world states are reached. This is particularly important if the program was used to enforce system behavior that conforms to given safety regulations.

### 7.1.1 Contributions

In this chapter we propose an algorithm for *compiling* ConGolog programs into basic action theories of the situation calculus whose tree of executable situations corresponds exactly to the one described by the program. We prove the correctness of the compilation and show that its output is of size as most quadratic in the size of the original program.

The compiled theory allows us to reason about the executions of programs using regression. Given an action sequence, we can "regress a program" over this sequence, producing a necessary and sufficient condition for the sequence to be a legal execution of the program. This allows us to once again apply the abstract monitoring approach we propose in this thesis, in order to monitor the continued satisfaction of the procedural hard constraints during planning and plan execution in highly-dynamic environments. We discuss this further in Section 7.5.

The compilation is significant for a number of additional practical and theoretical reasons. From a practical perspective, the compilation provides the mathematical foundation for compiling ConGolog control knowledge into the Planning Domain Definition Language (PDDL) [McDermott, 1998], a *de facto* standard planning problem specification language. This in turn enables state-of-the-art planners to exploit powerful control knowledge without the need for special-purpose machinery within their planners. We have recently shown how this can be done for a subset of the language without concurrency and procedures [Baier *et al.*, 2007]. The experimental results showed that state-of-the-art planners can gain significant speed-ups from that. The current compilation of

ConGolog (including concurrency and procedures) can be seen as an extension of this work – though some restrictions apply when compiling into PDDL.

ConGolog has been used for a variety of purposes, all of which can now benefit from this newly built connection to modern planners. For instance, Hierarchical Task Networks (HTN) have been translated to ConGolog [Gabaldon, 2002]. In combination, this translation and our compiler provide the means for compiling HTN control knowledge into a classical planning problem. We anticipate this contribution to be of significant interest to the planning community.

From a theoretical perspective, the compilation eliminates the need for ConGolog's tedious reification of programs, as well as the second-order axioms necessitated by its transition semantics. This facilitates proving properties of programs (e.g., reachability, invariants, termination). Further, since programs themselves can now be regressed, some proofs can be reduced to first-order theorem proving through the use of regression.

In this chapter we focus on the high-level idea of the compilation. The actual pseudo-code can be found in Appendix C, and experimental evidence in support of our basic approach can be found in [Baier *et al.*, 2007].

## 7.2 Background

### 7.2.1 Golog and ConGolog

#### Golog

Golog is a programming language defined in the situation calculus. It allows a user to specify programs whose set of legal executions specifies a sub-tree of the tree of situations of a basic action theory. From a planning point of view, it can be used to provide an effective way of pruning the search by specifying the skeleton of a plan. Golog has an Algol-inspired syntax extended with flexible non-deterministic constructs. Its constructs

are shown below.

| | |
|---|---|
| $a$ | primitive action |
| $\phi?$ | test condition $\phi$ |
| $(\delta_1; \delta_2)$ | sequence |
| **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ | conditional |
| **while** $\phi$ **do** $\delta'$ | loops |
| $(\delta_1 \vert \delta_2)$ | non-deterministic choice |
| $\pi v.\delta$ | non-deterministic choice of argument |
| $\delta^*$ | non-deterministic iteration |
| $\{P_1(\vec{t}_1, \delta_1); \ldots; P_n(\vec{t}_n, \delta_n); \delta\}$ | procedures |

The semantics of a Golog program $\delta$ is defined in terms of macro expansion into formulae of the situation calculus. $Do(\delta, s, s')$ is understood to denote a formula expressing that executing $\delta$ in situation $s$ is possible and may result in a situation $s'$. This is defined inductively over the program constructs. For instance for a primitive action $a$: $Do(a, s, s') \stackrel{\text{def}}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$, where $a[s]$ denotes the action $a$ with all its arguments instantiated in situation $s$, and for non-determinism: $Do(\delta_1 | \delta_2, s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$. While deterministic constructs enforce the occurrence of particular actions, non-deterministic constructs define "open parts" that are completed using planning. In particular, the non-deterministic choice of argument $\pi v.\delta$ introduces a *program variable* $v$ that may occur in $\delta$. In this chapter, we restrict program variables to only appear as action parameters or in the place of objects in conditions. For instance **while** $(\exists b).OnTable(b)$ **do** $\pi v.\ OnTable(v)?; Remove(v)$ could be a program that removes all blocks, one-by-one from a table.

### ConGolog

ConGolog adds concurrency to Golog, allowing the following additional constructs:

| | |
|---|---|
| $(\delta_1 \parallel \delta_2)$ | concurrent execution |

$(\delta_1 \rangle\!\rangle \delta_2)$ prioritized concurrency

$\delta^{\|}$ concurrent iteration

Concurrency is defined as action interleaving. For example, the program $(a \parallel (b; c))$ admits three executions: *abc*, *bac*, and *bca*.

ConGolog introduced a so-called *transition semantics* for programs. The semantics of a program $\delta$ is given through two predicates $Trans(\delta, s, \delta', s')$ and $Final(\delta, s)$. The former states that in situation $s$ program $\delta$ can perform a step, resulting in a remaining program $\delta'$ and new situation $s'$. The latter states that the program $\delta$ can legally terminate in $s$. De Giacomo *et al.* [2000] provide the complete axioms for the semantics; we show some of them below.

For a primitive action we have

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$$

and $Final(a, s) \equiv false$. One important role of *Final* is with sequences:

$$\begin{aligned} &Trans(\delta_1; \delta_2, s, \delta', s') \equiv \\ &\quad (\exists\gamma).\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \\ &\quad \vee\ Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s'). \end{aligned}$$

For concurrency constructs we have:

$$\begin{aligned} &Trans(\delta_1 \,\|\, \delta_2, s, \delta', s') \equiv \\ &\quad (\exists\gamma).\delta' = (\gamma \,\|\, \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \\ &\quad \vee\ \delta' = (\delta_1 \,\|\, \gamma) \wedge Trans(\delta_2, s, \gamma, s') \\ &Trans(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv \\ &\quad (\exists\gamma).\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \delta' = (\delta_1 \rangle\!\rangle \gamma) \\ &\quad \wedge\ Trans(\delta_2, s, \gamma, s') \wedge (\nexists\zeta, s'').Trans(\delta_1, s, \zeta, s'') \\ &Trans(\delta^{\|}, s, \delta', s') \equiv \\ &\quad (\exists\gamma).\delta' = (\gamma \parallel \delta^{\|}) \wedge Trans(\delta, s, \gamma, s') \end{aligned}$$

The first two programs are only "final" when both subprograms are, while the third can be terminated at will:

$$Final(\delta_1 \,\|\, \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

$$Final(\delta_1 \,\rangle\!\rangle\, \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

$$Final(\delta^{\|}, s) \equiv true$$

A transition semantics facilitates the interleaving of program interpretation (planning) and execution, and reasoning about sensing actions. The downside of this semantics is its requirement to reify programs: programs are represented as terms, in order to quantify over them. The other shortcoming is the requirement of an additional second-order axiom for defining the transitive closure of *Trans*, denoted $Trans^*$. This axiom is needed to define a new $Do_2$ predicate that defines the situations that result from executing a (ConGolog) program:

$$Do_2(\delta, s, s') \stackrel{\text{def}}{=} (\exists \delta').\, Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s').$$

We refer to the axioms defining the transition semantics as $\Sigma_{\text{ConGolog}}$. This includes the mentioned second-order axioms and axioms required for reification of programs.

## 7.3 Compiling ConGolog into Basic Action Theories

In this section we describe an algorithm for compiling a given ConGolog program and a given basic action theory into a new basic action theory. For readability, we focus our description on the intuitions behind the algorithm. The actual pseudo code of the algorithm can be found in Appendix C.

Our algorithm accepts as input a basic action theory $\mathcal{D}$ and a ConGolog program $\mathcal{P} = \{P_1(\vec{t_1}, \delta_{P_1}); \ldots; P_n(\vec{t_n}, \delta_{P_n})\}; \delta_{main}$ containing $n$ procedure definitions with formal arguments $\vec{t_i}$ and procedure body $\delta_{P_i}$, and a main program $\delta_{main}$. It outputs a new

basic action theory $\mathcal{D}_\mathcal{P}$ whose tree of executable situations corresponds to the sub-tree of situations in $\mathcal{D}$ that are executions of $\mathcal{P}$ in $\mathcal{D}$.

The intuition behind our compilation is to model the dynamics of a ConGolog program as a Petri net with an infinite stack, and then represent this Petri net and the stack as a basic action theory in the situation calculus. Roughly, a Petri net is a finite state automaton that can be in more than one state at the same time. To reflect that, in Petri net terminology, states are called *places* and active places are marked by *tokens* which move from place to place using transitions. The total number of tokens can change during execution, for instance to model concurrency. To model the dynamics of ConGolog programs, we use a so-called *colored* Petri net, where tokens have unique identifiers. We do not define the Petri net induced by a program formally, but only use it for illustration. Intuitively, places in the Petri net represent the current position in the execution of the program (i.e., a sort of program counter), while (labeled) transitions specify which actions are legal at each stage during the execution. Each token represents one of possibly several concurrently executing threads. Given a program $\mathcal{P}$, our algorithm generates the axioms required to model the underlying Petri net as a basic action theory. To this end, we create (1) special bookkeeping predicates, to represent the Petri net and the stack, and (2) additional actions, to represent some of the transitions in the Petri net.

It is important to note that our algorithm operates only syntactically on the given inputs. In particular, it does not perform any type of reasoning within the provided basic action theory, which makes it easy to show that our algorithm has modest complexity (see below).

The compilation proceeds in six steps.

**Step 1**

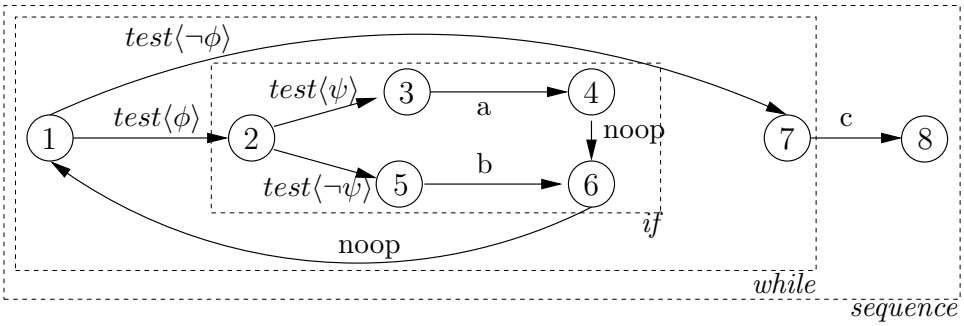
For each procedure $P_j(t_{j_1}, \ldots, t_{j_{k_j}}, \delta_{P_j})$ in $\mathcal{P}$ we compute

$$(\mathtt{ax}_j, \mathtt{i}_j) = \mathbf{comp}(\delta_{P_j}, 0, \{t_{j_1}, \ldots, t_{j_{k_j}}\}, P_j)$$

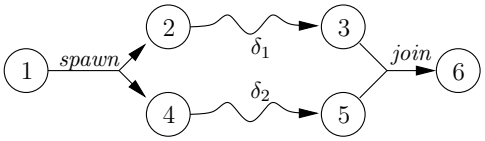
where $\{t_{j_1}, \ldots, t_{j_{k_j}}\}$ are the formal parameters of the procedure, and $\delta_{P_j}$ is the body of $P_j$.[1] The function **comp**, defined in Appendix C.1, takes as input a ConGolog program, an integer used as a program counter, a set of program variables, and a procedure name, used to distinguish different contexts. It outputs a set of sentences `ax`, and an integer `i`, intuitively denoting the value of the program counter after the program terminates. The set of sentences is later processed further to generate the axioms of $\mathcal{D}_\mathcal{P}$, but before we get to this, we first consider the function **comp** in more detail.

**comp** is defined recursively over the structure of programs. Starting from an initial place labeled $(0, \mathit{main})$, **comp** incrementally constructs the Petri net, generating new network places as it recurses over the structure of the program. Assume **comp** is currently at a place labeled with $(i, p)$, where $i$ is the program counter and $p$ a procedure name, and that it encounters a primitive action $\alpha$ in the program. Then, it adds a new place to the Petri net labeled with $(i + 1, p)$ and a transition from the current place to this new place, labeled with $\alpha$. **comp** generates and returns several sentences which will later be included as axioms of $\mathcal{D}_\mathcal{P}$. First, it generates a sentence about the preconditions of $\alpha$. In the described case it generates $\mathit{Poss}(\alpha(\mathit{th}), s) \leftarrow \mathit{Thread}(\mathit{th}, s) \wedge \mathit{state}(\mathit{th}, s) = (i, p)$ which states that we can execute $\alpha$ in thread $\mathit{th}$ if $\mathit{th}$ denotes an active thread and its token is in $(i, p)$. (Note that we give an extra argument to each action, denoting the thread it is being performed in.) It further generates an appropriate effect, stating that when $\alpha$ is performed in $(i, p)$, the token moves to $(i + 1, p)$. The sentence generated in this case is $\mathit{state}(\mathit{th}, \mathit{do}(\alpha(\mathit{th}), s)) = (i + 1, p) \leftarrow \mathit{state}(\mathit{th}, s) = (i, p)$.

[1] For simplicity of presentation we assume that procedures do not contain additional procedure definitions.




(a) Petri net for **while** $\phi$ **do** (**if** $\psi$ **then** $a$ **else** $b$); $c$.




(b) Petri net for $\delta_1 \parallel \delta_2$

Figure 7.1: Two example Petri nets.

**Example 1.** Consider the program of Figure 7.1(a), where special *test* actions are used to transition to a sub-net conditioned on a formula, and *noop* allows unconditional transitions.[2] To keep the presentation simple, we only show the sentences produced by the algorithm for the transitions from state $1 \rightarrow 2$ and $7 \rightarrow 8$.

For transition $1 \rightarrow 2$, if $\phi$ does not mention program variables, the algorithm generates the following sentences:

$$Poss(test(th, 1, 2, main), s) \leftarrow (Thread(th, s) \wedge \phi(s) \wedge state(th, s) = (1, main)), \tag{7.1}$$

$$state(th, do(test(th, 1, 2, main), s)) = (2, main). \tag{7.2}$$

And for the transition $7 \rightarrow 8$ we get:

$$Poss(c(th), s) \leftarrow (Thread(th, s) \wedge state(th, s) = (7, main)), \tag{7.3}$$

$$state(th, do(c(th), s)) = (8, main) \leftarrow state(th, s) = (7, main). \tag{7.4}$$

In the remaining steps of the compilation (see below), the successor state axiom for the

[2]Names used for test actions in this example are simplified for clarity. Refer to the pseudo-code for more details.

*state* fluent is formed and precondition axioms are put into normal form. If in $\mathcal{D}$ the precondition axiom for $c$ was $Poss(c, s) \equiv \Pi_c(s)$, then the new precondition axiom in $\mathcal{D}_\mathcal{P}$ is $Poss(c(th), s) \equiv \Pi_c(s) \wedge \varphi$, where $\varphi$ stands for the right-hand side of Equation 7.3. §

So far, the Petri net is equivalent to a simple automaton, since we have only been concerned with a single token. This changes when one considers concurrency. Concurrency is modeled using threads, where each thread is represented by an identifiable token in the net. For instance, the basic concurrency construct $\delta_1 \parallel \delta_2$ puts the current token in the initial state of the sub-Petri net recursively generated for $\delta_1$, and creates a new token which it puts into the initial state of $\delta_2$. These tokens are joined back together when both programs have finished executing (Figure 7.1(b)).

The greatest challenges we faced while devising **comp**, were caused by the interaction of various advanced programming constructs, in particular program variables, procedures, and iterative concurrency. We elaborate briefly on some of these challenges.

**Procedure calls** are realized using two new actions *call* and *return*. The former moves the token of the current thread to the initial place of the called procedure, while *return* returns it to the next state of the current program, once the token has reached the final state of that procedure. Since the compilation of the procedures themselves needs to be independent from the context from which they are called, we do not know the return state during compile time, but need to store it during run-time instead. Since procedures can be recursive, we require a stack, containing all (recursive) return states. The stack is realized using two functional fluents $stack(th, v, s)$ and $sp(th, s)$, where the former denotes the content of the stack entries, and the latter is a stack-pointer, always pointing to the next free position on top of the stack.

**Concurrency** is realized by using explicit thread names. Each action is given an additional parameter *th*, denoting the thread it is executed in. This is necessary since

there may be situations where two threads intend to execute the same action next. Once that action executes, we need the thread name to disambiguate which thread actually proceeded. Thread names are also required for other purposes, like program variables, described below. The active threads are denoted by the relative fluent $\mathit{Thread}(th, s)$, and initially only one thread, [0], is active. A new thread is created by the *spawn* action, which also sets up some new data structures (fluents) for the new thread, for instance its own procedure call stack. Two threads are joined back by the action *join*.

For thread names, we use lists of numbers. The main thread is [0], and its direct children are called $[N, 0]$ where N is the number of the child. The $k$-th child of the $n$-th child of the main thread is called $[k, n, 0]$. This is more complicated than increasing a single thread counter, which would have been an alternative, but has the advantage that thread names can be reused after threads terminate. With numbers, for instance, an infinitely running program with concurrency would require infinite numbers. This would also more severely limit the ability to compile into PDDL.

Prioritized concurrency is governed by a new fluent $\mathit{Prio}(th_1, th_2, s)$ which indicates that thread $th_1$ has priority over thread $th_2$. A thread can only proceed when no prioritized thread can perform an action.

**Program variables** as created by $\pi$ constructs, are realized using the functional fluent $\mathit{map}(x, s)$, to denote their value. The parameter $x$ is a tuple $(th, y, v)$ where $th$ denotes the thread this variable was created in, $y$ the stack position, and $v$ is the name as mentioned in the program (e.g., $\pi\, v.\delta$). Thread names are required to disambiguate in cases like $(\pi\, v.\delta)^{\|}$ where in each thread a new variable of the same name is created. Similarly stack positions are required when program variables are used in recursive procedures.

To compile the main procedure we call

$$(\mathtt{ax}_{main}, \mathtt{i}_{main}) = \mathbf{comp}(\delta_{main}, \mathtt{0}, \emptyset, main),$$

which yields the final program counter $\mathtt{i}_{main}$, which corresponds to a particular "final" place of the Petri net. This will be used as a goal: if there is a token in $(\mathtt{i}_{main}, main)$, the program has executed successfully. This roughly corresponds to the *Final* predicate in ConGolog.

**Step 2**

Thus far we have generated program-specific sentences, describing the dynamics of the Petri net. There is also a number of program-independent sentences that we require, which intuitively state the default dynamics of the involved bookkeeping actions (see Appendix C.2 for details). We denote these as $\mathtt{ax}_{\text{common}}$ and define the set $\mathtt{AX}$ as $\mathtt{ax}_{main} \cup \bigcup_j \mathtt{ax}_j \cup \mathtt{ax}_{\text{common}}$ .

The remaining steps of the compilation aggregate the sentences in $\mathtt{AX}$ to produce $\mathcal{D}_{\mathcal{P}}$, producing all the precondition axioms, successor state axioms, initial state axioms, and unique names axioms.

**Step 3**

Recall that procedure calls require two new actions *call* and *return*. The effect axioms for both are domain independent and thus in $\mathtt{ax}_{\text{common}}$, and the precondition axioms for *call* are generated by **comp**. In Step 3 we need to create the precondition axioms for *return*, which is possible in all final states, i.e., for each procedure $P_j$ compiled in Step 1, we enable *return* when $state(th, s) = (\mathtt{i}_j, P_j)$.

**Step 4**

For each place of each Petri net, all conditions under which any action can execute in this place and context are recorded. We generate axioms for a new fluent $CanTrans(th, s)$,

which indicates whether in situation $s$ a given thread $th$ can perform an action. This definition is only required in conjunction with concurrency, and can be skipped if this language feature is not used.

**Step 5**

For each primitive action $\alpha$ (including bookkeeping actions), Step 5 removes all sentences $Poss(\alpha, s) \leftarrow \phi$ from `AX` and combines them into a new precondition axiom for $\alpha$, by:

1. disjoining all $\phi$'s,
2. conjoining the resulting formula with any preexisting preconditions for $\alpha$, and
3. conjoining the result with the following additional condition that governs priority among threads and allows forced execution of a selected thread:

   $$(\nexists t).\mathit{Thread}(t) \wedge t \neq th \wedge (\mathit{Forced}(t) \wedge \neg \mathit{Parent}(t, th) \vee \mathit{Prio}(t, th) \wedge \mathit{CanTrans}(t))$$

   where the new relational fluent $\mathit{Parent}(th_1, th_2)$ expresses that thread $th_2$ was spawn from thread $th_1$, directly or indirectly.

The latter is used to enable prioritized concurrency, explicitly prohibiting threads from executing for which there is a thread with higher priority that can execute its next action. This condition is also used to ensure so-called *synchronized* while's and if's. Roughly, the latter means that testing the conditions of these constructs is not a transition by itself, but needs to be immediately followed by a transition on its body, or otherwise one needs to backtrack to a place before the test.

**Step 6**

Since all the *Poss* sentences have been removed, `AX` now only contains sentences describing effects of actions. On these, Step 6 applies Reiter's solution to the frame problem, to produce successor state axioms (see Section 2.1.2).

The result is a set of precondition and successor state axioms, describing the dynamics of all procedures' Petri nets. We also add the axiom $state([0], S_0) = (0, main)$, stating that initially the main thread, denoted [0], is in the initial place of the Petri net of the *main* procedure.

While our compilation makes several second-order axioms, specific to ConGolog's transition semantics, unnecessary, it does require second-order to define natural numbers and lists. The former is used to address the elements of the stack, the later to give names to threads. We assume standard definitions for these. These can be avoided when both recursion, and the number of concurrent threads is bound by a constant. This restriction is also required for further compilation to PDDL (see below).

Let $\mathcal{D}_\mathcal{P}$ be the new basic action theory resulting from compiling $\mathcal{P}$ into the given action theory $\mathcal{D}$. We can show the following theorems which state that the compilation is both correct and succinct. All the proofs can be found in Appendix C.

**Theorem 9.** Let $S'$ be any ground situation term of $\mathcal{D}$. Then there is a ground situation term $S'_\mathcal{P}$ in $\mathcal{D}_\mathcal{P}$ such that $S' = filter(S'_\mathcal{P}, \mathcal{D})$ and

$$\mathcal{D}_\mathcal{P} \models executable(S'_\mathcal{P}) \wedge state([0], S'_\mathcal{P}) = (\mathtt{i}_{main}, main)$$

iff $\mathcal{D} \models Do_2(\mathcal{P}, S_0, S')$.

Here $filter(S'_\mathcal{P}, \mathcal{D})$ is a function that removes from the situation term $S'_\mathcal{P}$ any actions not defined in $\mathcal{D}$, and also removes the additional thread argument from the remaining actions. This removes all bookkeeping actions from $S'_\mathcal{P}$, in order to compare the sequence of contained domain actions with $S'$.

For the next theorem we define the size of a program as the number of program constructs it contains plus the number of logical connectives mentioned in conditions. Similarly, the size of an axiom is measured by the number of logical connectives it contains.

**Theorem 10.** If the size of $\mathcal{P}$ is $n$ and $\mathcal{D}$ contains $m$ axioms each of size $\leq k$, then $\mathcal{D}_\mathcal{P}$ contains $O(n) + m$ axioms each of size $O(k + n)$.

**Theorem 11.** If the size of $\mathcal{P}$ is $n$, then the time required to compute the compilation is $O(n^2)$.

Intuitively, recursive procedure calls, while–loops, concurrency and other seemingly problematic constructs do not incur a significant increase in the size of the output, because of the syntactic nature of the compilation and the careful use of bookkeeping fluents and actions to model the desired behaviors. Similarly, the requirement for second-order logic to define loops is cast into the induction axiom included in the foundational axioms of the situation calculus, through the use of bookkeeping fluents and actions.

## 7.4 Analysis

### 7.4.1 Theoretical Merits

To prove properties of a ConGolog program $\mathcal{P}$, we now have two alternatives. We can reason using the original transition semantics of ConGolog, represented as a fixed set of axioms $\Sigma_{\text{ConGolog}}$, or we can use the new basic action theory $\mathcal{D}_\mathcal{P}$ resulting from applying our compilation, extended with natural numbers and lists. At first glance, using $\Sigma_{\text{ConGolog}}$ may look simpler since the axioms in $\Sigma_{\text{ConGolog}}$ are independent of the program. However, we argue that reasoning itself is actually simplified when using $\mathcal{D}_\mathcal{P}$.

One advantage of $\mathcal{D}_\mathcal{P}$ is that it defines the dynamics of a program in terms of fluents. For example, any executable situation $S$ for which $\mathcal{D}_\mathcal{P} \models \mathit{state}([0], S) = (\mathtt{i}_{\mathit{main}}, \mathit{main})$, with $\mathtt{i}_{\mathit{main}}$ as defined above, is a legal execution of the program. Regressing the condition $\mathit{state}([0], S) = (\mathtt{i}_{\mathit{main}}, \mathit{main})$ over the actions comprising $S$, together with all involved action preconditions, results in a formula over the initial situation $S_0$. Following Theorem 9 and Reiter's Regression Theorem, this formula is equivalent to the question of

whether the actions comprising $S$ are a legal execution of the program. More generally, using regression we can determine sufficient conditions under which a given sequence of actions (whose parameters do not need to be ground) will satisfy a given formula while executing the program. These queries could not be answered using regression in the transition semantics since neither the semantics of Golog nor ConGolog were in terms of regressable formulae[3]. This provides the theoretical basis for the applicability of the abstract monitoring approach of this thesis, as we discuss in Section 7.5.

Another advantage of reasoning in $\mathcal{D}_{\mathcal{P}}$ is that the compilation eliminates the need for ConGolog's tedious (second-order) reification of programs, as well as the second-order axioms found in $\Sigma_{\text{ConGolog}}$ for defining the *Trans* and the *Trans*$^*$ predicates. As such, proving properties of programs in $\mathcal{D}_{\mathcal{P}}$ is not much different from proving properties in the standard situation calculus. In some cases (e.g., when proving a property of a particular execution trace) we can apply regression. In more general cases (e.g., when proving invariants), we can simply use induction over situations [Reiter, 1993]. In fact, we have proved properties of simple Golog programs by representing $\mathcal{D}_{\mathcal{P}}$ in the higher-order theorem prover PVS [Owre *et al.*, 1992]. In PVS, situations, natural numbers, and lists, can be easily defined as recursive data-types. We found the lack of reification in $\mathcal{D}_{\mathcal{P}}$ together with the limited number of second-order axioms made theorem proving less laborious and more intuitive than previous attempts to prove properties of Golog programs in PVS [Shapiro *et al.*, 2002].

In our translated domain it is particularly simple to prove a property about a specific point during the program's execution. The main reason for this is that in our compiled theories we can refer to points in the program's execution by referring to the states of the Petri net that represent those points. For example, proving a property about the situations that result from executing the program to termination reduces to proving that a certain formula is true for every situation in which we are at the Petri net

[3][Reiter, 2001, p. 62] defines regressable formulae.

place that corresponds to the end of the program. When proving these types of properties using the second-order axioms of the original ConGolog semantics, as was done by [Shapiro *et al.*, 2002], one is forced to effectively simulate an execution of the program by incrementally evaluating the transitive closure of the *Trans* predicate. On the other hand, in case we want to prove a property that holds during the whole execution of a program using our compiled theory, we have to resort to induction over situations. The course of the proof in this case is very similar to the one that would be obtained in the framework of [Shapiro *et al.*, 2002].

To demonstrate the feasibility of proving properties of programs using automated theorem provers, we modeled one of the Golog example programs in the blocks world used by [Liu, 2002]. This program consists of a while loop that non-deterministically moves blocks until there is only one block on the table. The task is to prove that there is a single tower in the final situation. This could be proved automatically by PVS in fractions of a second. [Liu, 2002] also obtained a very simple proof but appealing to a Hoare-style proof system on top of ConGolog's semantics.

### 7.4.2 Practical Merits

#### ConGolog to PDDL

A practical consequence of the compilation is the possibility of further compiling the resulting action theory into other action languages, like PDDL. The advantage of this approach is the possibility of using the fastest state-of-the-art planners to accomplish the planning needed while interpreting ConGolog programs. This is not only of interest to the agent programming community but also for the planning community, since ConGolog can be used to express domain control knowledge.

In previous work we have shown that it is possible to compile Golog programs without procedures into PDDL [Baier *et al.*, 2007], and shown that Golog domain control knowledge can speed up search of standard planning benchmarks. Figure 7.2 shows an

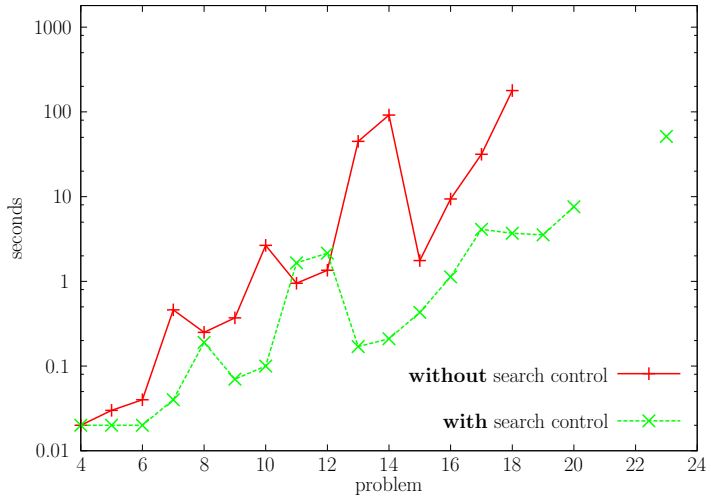



Figure 7.2: Run-time comparison of a heuristic search based planner solving instances of the `storage` domain of the International Planning Competition, with and without Golog search controlled compiled into the PDDL domain definition [Baier *et al.*, 2007].

example of the obtainable speed-up for the `storage` domain of the International Planning Competition.

In the compilation proposed in this paper we are considering the richer variant ConGolog, which allows programs with various forms of concurrency, and we also enabled the use of possibly recursive procedures. Unfortunately, these additions all together cannot be compiled directly into current versions of PDDL. The main reason is that PDDL does not provide the functionality for defining unbounded data structures, which we need, for example, for representing the stack for procedure calls.

Recent versions of PDDL support natural numbers, but these cannot be used as arguments to predicates, since numbers are not considered objects of the domain. The pragmatic reason for this restriction is to avoid the possibility of infinite branching factors [Fox and Long, 2003, p. 68] since actions could take numerical arguments. Since our compilation does not introduce infinite branching factors, we believe that PDDL could be extended accordingly to allow the full expressiveness of ConGolog and HTNs. We hope that our work may convince the planning community that such an extension would lead to a significant increase in the expressiveness of PDDL.

It is still possible to translate ConGolog into PDDL if we are willing to either disallow recursion and iterative concurrency or limit the depth of recursion and the number of concurrently executing threads. The second option is probably the most interesting one, since in practical applications in which finite plans are needed, we will not require the power of infinite recursion. The main challenge in this case is to generate a theory in which the stack and the lists which are used to represent thread names are bounded. The following are the main aspects that are needed to translate to PDDL.

1. All fluents that represent counters (e.g., the stack pointer fluent) are now represented by *relational* fluents, an argument of which corresponds to the value of the counter. The value of the counter is represented by a PDDL *object*. We generate finitely many objects for counters, and a static predicate to indicate the successor for each counter object.

2. All other functional fluents (e.g., *map* and *state*) are represented in PDDL as relational fluents. In particular, the relational fluent for *state* contains one argument for each element of the $(\mathtt{i}, \mathtt{c})$ pair.

3. Threads, which in our basic action theories are represented as lists, and which are employed as arguments to actions are represented in PDDL as (bounded) lists of size equal to a parameter $k$. Moreover, actions, instead of having a single thread parameter, are now represented as having $k$ additional parameters, where the $i$-th parameter of the action corresponds to the $i$-th parameter of the thread list. We emulate lists with fewer than $k$ elements by using a special constant *nao* (not-an-object) to represent a position of the list that is not occupied by any object. Finally, effects of the actions *spawn*, and *join*, which modify the current thread, can be straightforwardly modified to use this new representation.

4. The precondition of the *call* and *spawn* actions are modified such that they will not be possible if the capacity of the stack/thread list is already at its maximum.

Our PDDL translation is defined for ConGolog programs, that are assumed to operate over a preexisting PDDL domain and problem specification. Thus, we assume that, instead of receiving a basic action theory as input, the algorithm receives a PDDL domain and problem definition describing preconditions and effects of actions, and the initial and goal state of the planning problem. The steps of the compilation procedure that integrate the basic action theory with the output of the program compilation are trivially modified for the PDDL case. Thus, new bookkeeping actions are added, and existing domain actions receive additional parameters, preconditions and effects as necessary. More details on the general setup of this compilation can be found in [Baier *et al.*, 2007].

**HTN to PDDL**

Hierarchical Task Networks (HTNs) [Erol *et al.*, 1994; Ghallab *et al.*, 2004] are a popular planning formalism used to provide domain control knowledge to a planner by representing planning solutions in a hierarchical fashion. They have broad applications, including classical planning [Nau *et al.*, 2003] and web service composition [Kuter *et al.*, 2004]. The HTN formalism has been in a sense divorced from classical planning since state-of-the-art planners do not handle HTNs. Our approach enables the compilation of HTNs into basic action theories and – when bounding recursion – to PDDL. Compiling HTNs to PDDL is beneficial, as it provides the means of combining their expressiveness with modern planning techniques.

Several HTN variants have been proposed in the literature, and one particular one has been previously translated to ConGolog [Gabaldon, 2002]. Here we consider the HTN formalism described by [Ghallab *et al.*, 2004], using a compelling subset of the language for constraints allowed by the SHOP2 planner [Nau *et al.*, 2003], which obtained a second place in the 2002 International Planning Competition. The translation of this flavor of HTN to ConGolog is almost trivial.

In the variant of HTN planning that we consider, we distinguish three entities, which

are specified by the user: *tasks, operators*, and *methods.* Tasks represent parametrized activities to perform. They can be *primitive* or *compound*. Primitive tasks are realized by operators, actions that can be physically executed in the domain. Compound tasks need to be decomposed using one of possibly several applicable methods. A method $m$ is of the form `(:method` ***head***$(m)$ $p_1(\vec{v})$ $t_1(\vec{v}) \ldots p_n(\vec{v})$ $t_n(\vec{v})$`)` where the head specifies the task with formal arguments $\vec{v}$ to which this method is applicable, $p_i(\vec{v})$ are preconditions and each $t_i(\vec{v})$ is a list of sub-tasks. As in SHOP2, we give an if–then–else semantics to methods: if $p_1(\vec{v})$ holds, then the task is decomposed into the sub-tasks $t_1(\vec{v})$. Otherwise, $p_2(\vec{v})$ is tested and so on. For a method to be applicable to a given task instance, the task's actual parameters have to unify with the method's formal parameters, and at least one $p_i$ has to be satisfied. Each list of sub-tasks $t_i(\vec{v})$ can be a nesting of `:ordered` and `:unordered` lists, stating restrictions on the order in which these tasks can be carried out.

A detailed description of the formal algorithm for translating these HTNs to ConGolog is beyond the scope of this chapter, but roughly the construction proceeds as follows: For each method $m$, we create a new procedure

$$m\big(\vec{v},\, \textbf{if}\ p_1(\vec{v})\ \textbf{then}\ \delta_1\ \textbf{else if} \ldots \textbf{else}\ (p_n(\vec{v})?;\delta_n)\big)$$

where $\delta_i$ is the following program representing sub-task $t_i$: Recursively, if $t_i$ is an `:ordered` set of tasks, then $\delta_i$ is simply the sequence of these tasks. Otherwise, if $t_i$ is an `:unordered` set, then $\delta_i$ is the concurrent execution of all of these. For instance, `(:unordered` $a$ `(:ordered` $b_1$ $b_2$`) (:ordered` $c_1$ $c_2$ $c_2$`))`, would be translated to: $(a \parallel (b_1; b_2) \parallel (c_1; c_2; c_3))$. Since there may be more than one method applicable to a given task, we translate each task into a non-deterministic choice over all of its applicable procedures: $(m_1|m_2|\ldots|m_n)$.

An HTN represented in such a way as a ConGolog program can thus be compiled into a basic action theory just as easily, and by limiting the recursion depth of methods, we can again compile the resulting theory further into PDDL.

**Execution Monitoring**

The ability to regress the procedural hard constraints of programs over execution traces, allows us to again use the abstract execution monitoring idea developed in this thesis for monitoring the execution of programs. We describe this in detail in the next section, to put it in adequate context with the rest of the thesis.

## 7.5 Monitoring the Execution of ConGolog Programs

Recall our abstract monitoring approach, as stated in Chapter 3. The abstract idea was to regress any decision critical entities over the actions of a plan, and annotate the plan with the resulting formula. This formula could then be reevaluated at run-time to verify the continued satisfaction of the decision criterion. Since after compilation, the hard constraints imposed by the program are expressed as simple (bookkeeping-)fluents, we can again apply this technique to now monitor the continued validity of a plan with respect to the program it was produced from. Since these are hard constraints, this is best illustrated using the approach for monitoring plan validity described in Section 3.4, even though nothing prevents its use in conjunction with user preferences and, hence, optimality-monitoring, as described in Chapters 4 and 5.

As a matter of fact, the approach of Section 3.4 can immediately and without further requirements be applied to the planning problems resulting from our compilation. Recall that the goal formula, representing the successful execution of the compiled program is

$$state([0], s) = (\mathtt{i}_{main}, main).$$

Hence, regressing this formula over the steps of a valid plan and annotating this plan with the resulting formula, again provides us with the means of determining the continued validity of the plan at run-time.

We demonstrate this once again using the example considered in the discussion of the approach for monitoring the execution of plans resulting from Golog programs presented

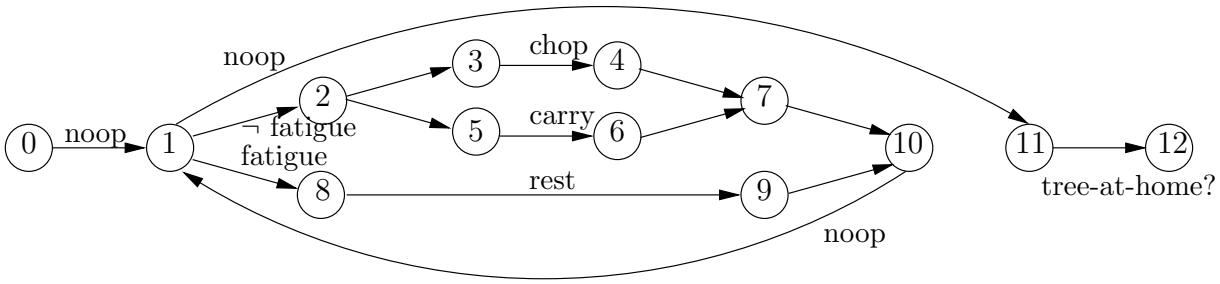



Figure 7.3: Petri-Net for tree chopping program

by [Ferrein *et al.*, 2004], described in Section 3.2.2, page 29. Recall that the task was to chop a tree and carry it home, but resting any time fatigue is felt. This task and hard constraint was represented by the program:

(**if** ¬fatigue **then** (`chop` | `carry-tree`) **else** `rest` **endIf**)* ; tree-at-home?

and given the model assumptions, the Golog interpreter had determined that the sequence of actions `chop, chop, chop, rest, chop, chop, carry-tree` is a legal execution, i.e., solves the task.

Using the compilation approach of this chapter, we can compile the program together with the original basic action theory into a new basic action theory, and use any appropriate planner to find a plan. Since in the resulting theory we use additional bookkeeping actions and fluents to represent the procedural constraints of the program, a plan for the given problem would look like this, where we omit the thread (`[0]`) and context argument (`main`) from all actions for readability, and further enumerate all actions for later reference:

noop(0,1)[1],

test(1,2)[2], noop(2,3)[3], chop[4], noop(4,7)[5], noop(7,10)[6], noop(10,1)[7],

test(1,2)[8], noop(2,3)[9], chop[10], noop(4,7)[11], noop(7,10)[12], noop(10,1)[13],

test(1,2)[14], noop(2,3)[15], chop[16], noop(4,7)[17], noop(7,10)[18], noop(10,1)[19],

test(1,8)[20], rest[21], noop(9,10)[22], noop(10,1)[23],

test(1,2)[24], noop(2,3)[25], chop[26], noop(4,7)[27], noop(7,10)[28], noop(10,1)[29],

test(1,2)[30], noop(2,3)[31], chop[32], noop(4,7)[33], noop(7,10)[34], noop(10,1)[35],

test(1,2)[36], noop(2,5)[37], carry-tree[38], noop(6,7)[39], noop(7,10)[40], noop(10,1)[41],

noop(1,11)[42], rtest(11,12)[43]

This plan, different from the previous one, implicitly contains the hard constraints expressed by the Golog program – in the preconditions of the (bookkeeping-)actions. This information is used by our monitoring approach to generate explicit monitoring formulae, stating the continued validity of the plan.

To provide more detail, assume that the original basic action theory entails the following:

$$chops\_remaining(do(\texttt{chop}, s)) = x \leftarrow chops\_remaining(s) = x + 1$$

$$tree\_down(do(\texttt{chop}, s)) \leftarrow chops\_remaining(s) = 1$$

$$Poss(\texttt{carry-tree}, s) \equiv tree\_down(s)$$

$$exhaustion(do(\texttt{chop}, s)) = x \leftarrow exhaustion(s) = x - 1$$

$$fatigue(do(\texttt{chop}, s)) \leftarrow exhaustion(s) \geq 2$$

$$\neg fatigue(do(\texttt{rest}, s))$$

$$exhaustion(do(\texttt{rest}, s)) = 0$$

Hence, chopping reduces a counter of remaining chops, and when this counter reaches zero, the tree is down and can be carried. Simultaneously, chopping is exhausting, in-

creasing the level of exhaustion by one. Fatigue is felt when the exhaustion reaches a value of three. Resting removes fatigue and resets the level of exhaustion to zero.

Applying the regression technique of Section 3.4, produces the plan annotation required to monitor continued plan validity (cf. Definition 2). The goal is $state([0], s) = (12, main)$, which – for ease of presentation – we abbreviate to $state(s) = 12$: since this program does not utilize procedures or concurrency, the specification of a procedural context or thread name is not necessary in this example. Regressing this goal over the last action and conjoining it with the preconditions of the last action, we get, for instance $G_{43}(s) = (state(s) = 11 \wedge tree\text{-}at\text{-}home(s))$, and after a few more regression steps $G_{38}(s) = (state(s) = 5 \wedge tree\text{-}down)$.

Consider again the same execution discrepancy assumed in Section 3.2.2: the agent feels fatigue already after two chops. This could be represented by a situation term $S^* = do(set(fatigue), S_i)$, where $S_i$ is the expected situation $(do(\texttt{chop}, do(\texttt{chop}, S_0)))$, and the special action $set(\phi)$ makes the fluent $\phi$ true. The condition with which the plan is annotated after the second chop action step (10) is: $G_{11}(s) = (state(s) = 4 \wedge chops\_remaining = 3 \wedge \neg fatigue(s))$. Hence, Algorithm 1 would determine that replanning should be performed instead of continuing with the plan when observing that $\mathcal{D} \not\models G_{11}(S^*)$.

Furthermore, the initial condition is $G_1(s) = (state(s) = 0 \wedge chops\_remaining(s) = 5 \wedge \neg fatigue(s) \wedge exhaustion(s) \not\geq 1)$. Hence, if already before execution commences the initial state of the world unexpectedly changes to a state described by a situation $S_2^*$ for which $\mathcal{D} \models exhaustion(S_2^*) = 1$, then the algorithm immediately calls for replanning, before even beginning to execute the plan, since the plan is already now projected to violate the hard constraints of the program.

While the example of step 11 achieves something very similar to what was achieves by [Ferrein *et al.*, 2004] by the use of explicitly generated "markers", the second example demonstrates one of the benefits of using the regression based technique presented in this

thesis. However, an even greater benefit results from the ability to combine this way of monitoring the continued satisfaction of procedural hard constraints with the monitoring for additional soft constraints – by using the algorithm presented in Chapter 4. Further, also the algorithm of Chapter 5 can now be applied to programs, often allowing us to produce optimal plans that also adhere to procedural hard constraints specified through Golog programs, even under frequent interruptions of the plan generation process through exogenous events.

Last but not least, it is even possible to apply the technique to DTGolog programs, if the approach of Chapter 6 is applied to the compilation result. DTGolog [Boutilier *et al.*, 2000], which stands for decision-theoretic Golog, is a variant of Golog in which actions can have stochastic outcomes (as described in Chapter 6). DTGolog programs are syntactically legal Golog programs as well, and are hence handled gracefully by the existing compilation, and the only required modification is to ensure that action effects are added to the primitive actions describing outcomes, while additional preconditions are added to the stochastic actions themselves (cf. Page 88).

## 7.6 Related Work

There are several pieces of related work. In previous work we provided a compilation of Golog programs without procedures into PDDL [Baier *et al.*, 2007], showing that notable speed-ups can be obtained in planning benchmarks. Our current work significantly extends the aforementioned compilation by showing how ConGolog programs (with procedures and extended with useful features like concurrency) can also be translated into classical planning, under certain restrictions. While our previous work exploited automata in the translation, the added expressivity of ConGolog necessitated the use of Petri nets.

Funge [1998] provided a compilation of Golog programs into Prolog, to make program

interpretation more efficient. His approach is similar to ours in the sense that the output can be viewed as representing a finite-state automaton. However, the output is not a logical theory, the approach cannot handle concurrency, and there are no immediate applications like planning.

There is also related work on the compilation of HTNs into ConGolog and PDDL. As previously noted, Gabaldon [2002] presented a means of translating the general HTN formalism of Erol *et al.* [1994] into ConGolog. We showed how the HTN formalism [Ghallab *et al.*, 2004] with the popular SHOP2 [Nau *et al.*, 2003] language for constraints could be translated into ConGolog and in turn compiled into PDDL. We limited ourselves to SHOP2 constraints because of its practical interest; this syntax also eliminated the need for additional predicates. Nevertheless, we could have just as easily used Gabaldon's more involved translation to ConGolog to compile general HTNs with bounded recursion into PDDL. Of further note, recently Lekavý and Návrat [2007] provided a linear translation of a restricted acyclic subset of HTN into STRIPS. Their translation generates a Turing machine with a finite tape represented in STRIPS.

Finally, there is related work on proving properties of Golog/ConGolog programs. Shapiro *et al.* [2002] used PVS to prove properties of ConGolog programs appealing to a direct representation of the $Trans^*$ second-order axiom, and by reifying programs. As a result it is possible to use induction to prove properties that hold during the execution of programs, but it is not straightforward to prove properties that hold at particular points in the execution (e.g., at the end of the program). As we have demonstrated, in our case proving any of these properties is done as with any property of the situation calculus. Also of note, Liu [2002] introduced a Hoare-style proof system for proving properties of Golog programs (without concurrency). The motivation for this approach was similar to ours: to minimize second-order reasoning. As a consequence, proving properties is facilitated in this formalism, too. Recently, Claßen and Lakemeyer [2008] proposed an interesting algorithm for proving properties of non-terminating Golog programs expressed

in a logic that resembles CTL*. To prove such properties, they construct a *characteristic graph*, which resembles our Petri nets. With our compiled domains and by using known translations of LTL into planning goals (e.g., [Baier and McIlraith, 2006]) we could prove similar properties, but restricted to only finite executions.

## 7.7 Discussion

In this chapter we proposed an algorithm for compiling arbitrary ConGolog programs into basic action theories in the situation calculus. The size of the resulting theory is quadratic in the size of the compiled program, and contains a simpler set of axioms, in the sense that it avoids the need for program reification and reduces the number of second-order axioms. The compilation presents a significant contribution for at least two reasons. First, it provides the mathematical foundations for compiling powerful ConGolog and HTN search control into basic action theories of the situation calculus. These can in turn be translated into other action formalisms including, with minor restrictions, PDDL. Such a translation enables most state-of-the-art planners to exploit powerful domain control knowledge without the need to construct special-purpose machinery within their planner. Second, in eliminating the need for reification, the translated theory facilitates automated proof of program properties in systems such as PVS as well as, in some cases, enabling properties to be proved by regression of ConGolog programs followed by (first-order) theorem proving in the initial situation. In the context of this thesis, the latter is of particular interest, as it once again enables the use of the abstract monitoring approach we propose.

# Chapter 8

# Related Work

In Section 3.2 we have already reviewed a number of existing approaches related to execution monitoring. Our focus there was on approaches that inspired the general abstract approach that provided the overarching technique of which we derived the solutions described in this thesis, and on approaches that, explicitly or implicitly, addressed the problem of state evaluation described in Section 1.2. Before concluding, we now review less directly related work found in the literature, with a focus on breadth rather than depth, in order to position the presented work in an adequate scientific context. Some of these works relate to the other modules listed in the described execution monitoring framework, particularly replanning and state estimation. We also review alternatives to execution monitoring, and proposals that address other issues of practical execution monitoring, not considered in this thesis (meta-reasoning, and learning).

## 8.1 Replanning

Recall our conceptual framework of execution monitoring described in Section 1.2. Once we have estimated the actual state of the world, evaluated it with respect to the plan, and asserted that the current plan has become invalid or sub-optimal, we have to decide how to react to this. In particular we do not want to replan from scratch in the new

state but should try to repair our current plan accordingly. Or not? Nebel and Koehler [Nebel and Koehler, 1995] showed that modifying a given plan for an altered initial and goal state has the same worst case complexity as planning from scratch. This holds even when similarity between the planning tasks is assumed and as little as one atom[1] is removed from or added to the goal while the initial state remains unchanged. The result also holds for the reverse case of a minimally altered initial state, the common situation in execution monitoring after a discrepancy occurred. When *conservative*, that is minimal, modification of the existing plan is required, the case gets even worse. Then plan modification can be even more complex than planning from scratch, that is, there are cases where planning from scratch can be done in time polynomial in the size of the problem definition, while minimally modifying an existing plan is NP-complete.

While this suggests to not even consider repairing a failed plan but to simply replan from scratch, several people (including us) have found plan modification to be more efficient in practice [Kambhampati, 1990; Gerevini and Serina, 2000; Koenig *et al.*, 2002; Hanks and Weld, 1995]. We will elaborate on this further in Section 8.1.1. This raises the question of how significant the presented theoretical worst case results are in practice. The proposed similarity measure based on the removal or addition of unspecified atoms from or to the goal or initial state seems not very informative in this respect as well. In Chapter 5 we have further demonstrated that in order to generate (optimal-) plans in practice when unexpected exogenous events are frequent, not even the absolute replanning time is most significant, but the *average relative* replanning time compared to the "amount of change".

Another reason why Nebel and Koehler's results may not be very relevant to the problem considered in this thesis is the fact that they limit their considerations to STRIPS planning without preferences (soft constraints) over plans. It remains to investigate whether these results also apply to plan modification under planning with preferences.

[1] Nebel and Koehler base their considerations on propositional STRIPS planning.

This raises a much broader issue regarding the available literature on execution monitoring and replanning in particular: In almost all approaches the objective in replanning is to minimize the replanning effort, *not* to maximize the quality of the generated plan. This was also one of the concerns of [Cushing and Kambhampati, 2005]. The authors argue that generally plan optimality is desired and planning time is only a secondary objective, but that current replanning methods do not account for that. The naive solution for achieving optimality is of course continuous planning, i.e., replanning from scratch after every execution step. This strategy was for instance used in [Lazovik *et al.*, 2003], where the authors present a framework for planning and monitoring the execution of web service requests.

Cushing and Kambhampati [2005] also raise concerns about the common limitation to replanning for altered initial and/or goal states. Some changes in the world, in particular those affecting the available operators, cannot be modeled under these assumptions. Imagine for instance a robot who breaks her gripper and cannot lift objects anymore. Cushing and Kambhampati propose to precede replanning with a model-adjustment step to alter the planning operators as necessary to accommodate for this kind of discrepancies. Not doing this implicitly assumes that a failure is never due to a systematic fault and this ignorance can lead to the infinite repetition of such a failure. We will come back to this issue in Section 8.1.3.

### 8.1.1 Plan Repair

Despite the theoretical worst-case results by Nebel and Koehler, many people have shown plan modification more efficient than planning from scratch in practice. The motivation for this work has not always been execution monitoring, it was also explored as a means of more efficient planning, so-called case-based planning or planning from second principles. Instead of planning from first principles when a planning problem arises, the idea is to consult a library of preexisting plans, find one that matches well with the new problem,

and then modify it according to the new requirements. The problem of replanning in execution monitoring is a special case of this where the plan library consists of only one plan, the current but failed plan. We here do not distinguish the presented results by their original motivation and for those approaches rooted in case-based planning, we ignore the methods for plan retrieval from a library.

Kambhampati [1990] describes a plan modification framework based on the hierarchical task network (HTN) planner PRIAR. This paper is of particular interest as it again demonstrates the utility of the rationale, now in particular for the plan modification/repair task. Again the plan is annotated with the rationale, the goal regressed through the remainder of the plan. Kambhampati calls this annotation the *validation structure* of the plan. As the name suggests, the validation structure serves to verify the plan's validity. A *validation* is a 4-tuple $\langle E, n_s, C, n_d \rangle$, where $n_s$ and $n_d$ are leaf nodes of the HTN, i.e., primitive actions, $E$ is an effect of $n_s$ (the source) and $C$ is a precondition of $n_d$ (the destination). Each node $n$ in the HTN is annotated with (i) the schema instance that reduced (expanded) the node, (ii) its *e-conditions* (external effect conditions), the effects of any node below $n$ in the hierarchy that support a validation outside of the $n$-subtree, (iii) its *e-preconditions*, the preconditions of any node in this subtree supported by a node outside of the subtree, and (iv) its *p-conditions* (persistence conditions). The latter, p-conditions, are validations whose source is scheduled before, and whose destination is scheduled after the task of node $n$, thus requiring that the validation's effect is not invalidated by any node within the subtree of $n$. This structure serves both to evaluate whether the plan is still valid and for replanning when it is not. A violation can be the failing of a validation, a missing validation, or an unnecessary validation. Roughly, in the two former cases a new sub-goal node for achieving the missing support is added to the network, in the latter case, the unnecessary validation is removed and this removal propagated, potentially removing any supporting actions which are no longer required. The resulting modified HTN will be handed back to the planner, reducing potentially

remaining new, open sub-goals. Using the blocks world domain, Kambhampati shows that doing plan repair this way can be between 30% and 98% faster than planning from scratch depending on the similarity of the two problem instances. The results also show that plan modification generally seems to pay off more the more complex the problems are, in this case measured by the number of blocks. The setup and generality of the results were, however, questioned by [Nebel and Koehler, 1995] for two reasons:

1. all considered instances belong to a sub-class of planning problems in the blocks world domain which are solvable in polynomial time,

2. all instances are free of "deadlocks", meaning that it is never necessary to put a block temporarily on the table in order to reach the goal.

Based on the SNLP partial-order planner, Hanks and Weld [1995] implemented a plan modification system called SPA. They also annotate the steps of the plan with the "reasons" for adding it, but they utilize this information differently: when the need for plan modification arises, the information can be used to retract earlier additions with all their consequences. The presented replanning algorithm then just performs search in the space of partial plans starting from the current one. The presented empirical comparison to PRIAR draws a mixed picture: the savings rate of PRIAR is higher, but the authors argue that this is because PRIAR is simply slower in generative planning[2], a claim also supported by Nebel and Koehler. This criticism is countered by the theoretical intuition that PRIAR exploits the plans annotation in more detail, which in turn could explain its practical superiority.

Another partial-order planner based approach is the GPG system of Gerevini and Serina [2000], based on planning graphs [Blum and Furst, 1995]. The modification of a plan for altered initial and/or goal sets of atoms is driven by inconsistencies

[2]Both replanning systems are compared to their own generative planner to determine their relative savings.

in the plan. An inconsistency is either an open precondition, an open goal condition, or two parallel actions that are mutually exclusive. The main algorithm (ADJUST-PLAN) processes these inconsistencies one by one starting with those of least time index. A window around the inconsistency is cut out of the plan and replanned, using the set of true fluents at the beginning of the window and the set of preconditions of actions at the end of the window as initial and goal states. If there is no plan for the current window, the window is increased. Replanning for one inconsistency may introduce new inconsistencies later in the plan. These are dealt with as the considered time index proceeds. The replanning algorithm is clearly sound and complete since in the worst case it eventually performs replanning from scratch, namely once the size of the replanning window has been increased to the point where it is spanning the entire plan. Surely, only replanning for the preconditions of the actions immediately to follow the window is not enough, as goal conditions may be affected by the replanning if e.g., an action inside the window that achieved a goal condition is not re-added and not compensated for otherwise. This insight led Gerevini and Serina to what they call the Backward Ω-goal set, which in fact is again nothing more than the regression of the goal over the remainder of the plan. Ideally one would use this set as the sub-goal to plan for when cutting a window instead of just using the preconditions of actions immediately after. In the considered partial-order setup this set can however not be computed when there are still inconsistencies in the remainder of the plan in which case GPG approximates the set. Experimental results on slightly modified `logistics`, `rocket`, and `gripper` example problems show that this technique can again be much faster than planning from scratch, sometimes four orders of magnitude.

An even more efficient and fairly universal approach to plan repair was presented by van der Krogt and de Weerdt [2005]. The idea is to reuse the heuristic used by a competitive planner to guide plan modification. The motivation for this is that replanning is not essentially different from generative planning, but still most available replanning systems

do not use a competitive planner. Roughly, the idea is to non-deterministically remove actions from the plan and then add new actions, where the choice of actions to remove or add is guided by the heuristic. The presented approach is universally applicable with all heuristic search based planners but requires that the heuristic is capable of evaluating arbitrary partial plans. While this is not generally the case, as most heuristic search based planners deploy forward-search and their heuristics are designed accordingly, van der Krogt and de Weerdt propose the following method to overcome this problem: after removing an action from the plan, divide the remainder into pieces, so-called *cuts*, in such a way that no two actions in the same cut were previously connected by a removed action.[3] Then, create a macro-action from each cut and add it to the theory. After that, evaluating the empty plan, or, in heuristic forward-search terminology, evaluating the initial state, provides the required heuristic information for the partial plan that was created from removing actions. An empirical comparison with GPG on problems of slightly modified initial states or goals shows that the presented approach can be anywhere between two to four times faster on some domains and twice as slow on others, while the plan quality, defined as the length, remains comparable. While the idea of lifting the planning heuristic to replanning is certainly an interesting new perspective, the automatic creation of macro actions from plan fragments is not always trivial depending on the action description language, raising questions about the universality of the approach.

One of the results of Nebel and Koehler [1995] was that plan reuse, and therefore also plan repair, can be even less efficient than plan generation in the worst case when minimality of changes, so-called conservative plan modification, is required. Also this concern was addressed empirically: Fox *et al.* [2006] presented a thorough empirical comparison of plan repair and plan generation on problems from the International Planning Competition, using the LPG planner, a local search heuristic planner similar to the earlier used

[3]This is always the case in totally-ordered plans, but in the exemplified partial-order setting (using the VHPOP planner) plans can be graphs as actions can be performed concurrently and then this requirement makes sense.

GPG system. Fox et al. define the *distance* between two plans as the cardinality of actions appearing in either plan but not their intersection and speak of greater *plan stability* when one replanning strategy produces a new plan of a smaller distance to the old plan than another replanning strategy. To maintain high plan stability, they extend LPG's heuristic by a term penalizing the addition or removal of actions increasing this distance. Unsurprisingly the modified system achieves greater plan stability on replanning tasks than planning from scratch, while also being faster in most cases. However, Fox et al. use a different notion of conservative modification than the one underlying the results of Nebel and Koehler. Nebel and Koehler distinguished several ways of modifying a plan and were only able to show that conservative modification can be more complex than planning from scratch for modifications where not only the cardinality of the actions the two plans have in common is maximal, but also their order is preserved. This restriction is not present in the work by Fox et al..[4]

In contrast Sapena and Onaindia [2002] adopt a strategy where actions can only be deleted at the front of a plan. The objective in this work is to find a plan suffix that is executable from a state which is reached after executing a minimal number of actions in the current (actual) state.[5] The presented experimental results show that this approach is faster than planning from scratch on problems with minor changes to the initial state while slower on problems with major changes.

In the robotics community search techniques based on the $A^*$ algorithm dominate approaches used to address the navigation problem. This is the problem of navigating a robot through a dynamic environment without colliding with any objects. Due to the dynamics it is often the case that the presence or position of obstacles change[6] while a

[4]The modification strategy of Fox et al. corresponds to the MODMIX strategy of Nebel and Koehler for which they were not able to show above worst case results (cf. the footnote on page 9 of [Nebel and Koehler, 1995]).

[5]This is subsumed by the MODDEL strategy of Nebel and Koehler.

[6]more specifically the agent's belief about obstacles changes

robot is executing a navigation plan and then the robot needs to replan its trajectory. This is a special case of the general replanning problem we are concerned with where only the applicability of actions in the search tree change, or, in search terminology, edges and states are removed from or added to the search tree or the costs assigned to edges change. Despite these limitations the more advanced research results along these lines can serve as inspiration for the more general replanning problem. These techniques all guarantee optimality[7] of the plan modification result and also other aspects have been addressed and now include, for instance, an anytime algorithm for this sort of "replanning" [Ferguson *et al.*, 2005]. Koenig *et al.* [2002] describe how the techniques can be lifted to symbolic replanning (in a system called SHERPA), unfortunately without lifting the limitation to cases where only the applicability of actions (in the search tree!) have changed. While this is certainly too limited from our perspective, the guaranteed optimality of the resulting plan still makes this interesting for us. It is important though to understand the complexity of the limitation. If none of the edges in the search tree is affected, the algorithm will do nothing and claim optimality of the current plan. However it is easy to construct examples where this fails, that is, cases where optimality is claimed but replanning from scratch would find a better plan. One is in settings with conditional effects: a discrepancy may not affect the preconditions or costs of any action in the plan, yet what the plan produces is not in accordance with what was planned for. Another way to confuse this approach is when the heuristic value depends on the available operators and their costs, as opposed to being a simple mapping from states (or state features) to numbers. In fact the heuristic functions most commonly used satisfy this criterion as they span a relaxed forward search graph to estimate a distance from the current state to the goal. During heuristic search planning, parts of the search space are pruned when, roughly, the heuristic function states that no plan of a better quality than $x$ can be found in this part of the search space and there are candidates of quality better than

[7] with respect to all possible plans in the current state

$x$. But this information may change when the costs of operators in these pruned parts change, potentially making earlier pruned parts now more attractive. It is thus not enough to limit oneself to the edges in the originally spanned search tree, but the impact of discrepancies on the heuristic function has to be considered as well.

We have formally addressed this issue in Chapter 4 by taking the conditions relevant in computing the heuristic function into account. We have shown that regressing these conditions, among others, enables us to derive sufficient conditions for the continued optimality of the current plan which can be evaluated in the actual state encountered during plan execution. The regression further enables us to determine the impact of a discrepancy on these conditions, avoiding a complete reevaluation of these conditions when only a subset of these conditions is affected. While our approach and SHERPA share the requirement to retain the search tree and/or the open list, our approach does not suffer from the limitation to changes that affect the costs or applicability of actions only, but works for arbitrary changes to the state. As we have shown, action costs can be easily represented by fluents, enabling our approach to handle changes to action costs as well.

### 8.1.2 Backtracking

As an alternative to modifying the remainder of the current plan, some have considered the possibility of on-line backtracking to previous points in the plan from where an existing alternative plan could be executed. This makes particular sense in conjunction with conditional planning. When a condition that decides between possible sub-plans is evaluated in a wrong belief about the actual state of the world, it may be beneficial to backtrack to this point of plan execution when later noticing the mistake, so that the correct sub-plan can be followed instead.

In [Golden *et al.*, 1996] the motivation for backtracking is that under time constraints, on-line systems may start executing a plan prefix while the plan has not been worked out

completely yet. Backtracking to a previous choice point then becomes necessary when it turns out that the executed plan candidate does not reach the goal. Unfortunately, this paper, whose focus is the description of the XII planning system, only outlines the benefits and problems with backtracking. The work by Eiter *et al.* [2004] is more elaborate. The authors propose the off-line generation of backtracking libraries that can be used as patch-plans upon execution failure to lead the system back to a diagnosed point of failure from where an alternative plan to reach the goal may be found. The key contribution of this paper is that of formulating the problem of finding pairs of action sequences and reverse plans as a conformant planning problem. From there, the authors show several complexity results by reduction from evaluation of Quantified Boolean Formulae (QBFs). In particular they show that determining whether a given action sequence has a reverse action or a reverse plan is $\Sigma_2^p$ hard or $\Sigma_3^p$ hard respectively (for the considered propositional action representation framework). In this work, a reverse action (resp. reverse plan) for an action sequence $AS$ is, roughly, any action (resp. sequence of actions) such that for any two states $S, S'$ for which executing $AS$ in $S$ produces $S'$ it is the case that the reverse action (resp. plan) executed in $S'$ always leads to $S$ if it is executable in $S'$. This paper is only of theoretical interest: computing a reverse plan for every possible action sub-sequence of a plan prior to executing it does not seem practical considering the demonstrated complexity.

Soutchanski ([Soutchanski, 2003a], [Soutchanski, 2003b, p.122]) has proposed an extended recovery predicate involving backtracking. Unlike the recovery predicate of De Giacomo *et al.* [1998], this predicate is not required to find a repair (patch) plan with which the current remaining program can be prefixed to make it executable again but instead also considers backtracking to an earlier program state (recorded in a so-called program state history) from where alternative execution branches exist. The backtracking is realized through planning on-line. Again we note that Soutchanski's focus is on formalizing the problem in the situation calculus. In particular, the time efficiency of the

applied planning algorithms used to implement the specified predicates do not compare to other state of the art planning techniques.

### 8.1.3 Learning

In all approaches described above the premise is that planning from scratch in the new, unexpected situation would produce a plan that is valid and optimal and which, in fact, is taken as measure for any replanning algorithm in terms of quality and speed. This premise also applies to the approaches proposed in this thesis. But what if the discrepancy that occurred is due to a systematic error and will thus repeat itself? This is for instance the case when the agent applies an incorrect model of its own actions during planning. Consider the following example of a soccer robot equipped with a kicking device[8]: The user provided the robot with a model describing that kicking will make the ball travel in a straight line until it hits an obstacle. Unfortunately, during the game a fuse blows, causing the kicking device to fail entirely. Assume the robot has intercepted the ball and is in a good position to score a goal by either kicking or pushing the ball into the goal. Since kicking is usually faster this is the preferred option, as it has a higher probability of success. Hence the robot triggers a kick action, but nothing happens, because of the hardware defect. The robot realizes that something went wrong when observing that the ball is still right in front of it, as this observation is inconsistent with the model. But planning anew in the new situation will not do any good since the best plan will still be to kick instead of pushing the ball – according to the erroneous model. None of the approaches we have described so far would ever get out of this loop and the robot would miss its chance of scoring. What is missing is a model adjustment step before replanning to account for modeling faults.

In this section we review replanning approaches where this problem has been addressed. We will elaborate on the obvious relation of this problem to reinforcement

[8]We have indeed encountered such a scenario in our RoboCup experience.

learning in the next section where we discuss other related fields of research.

McNeill *et al.* [2003] consider the problems arising in the execution of agent plans in a multi-agent setting due to faulty ontologies. Their approach prescribes that before executing a plan, the plan is "deconstructed" in order to annotate it with the assumptions made during planning. These annotations are then used when a discrepancy arises, to pinpoint possible causes of the fault. Intuitively these annotations state why the agent thought the plan-elements would work, that is, why it thought each precondition was satisfied. If the action then fails at execution, one can conjecture what went wrong in the past, i.e., which past action failed to produce its required effect (compare this to the diagnosis task and in particular the work by [Iwan, 2000], described in Section 8.3). The authors make a strong assumption namely that the reason why an action fails is given to the agent already as a condition: if an action $A$ fails, the system is informed that this is because a – possibly previously unknown – precondition $\phi$ of $A$ was not satisfied. The authors propose to then correct the systems ontology. This may involve either changing facts in the theory, corresponding to changing the belief about the state of the world, or modifying the signature of the ontology itself. Unfortunately the paper does not suggest ways of deciding what has to actually be changed and how this change can be done automatically.

Less original but more principled and detailed is the proposal of Bjäreland [1999]. This paper begins by formalizing the problem in the situation calculus and distinguishes between two sources for discrepancies *exogenous actions* (EA) and *violation of ontological assumptions* (VOA). The paper makes one critical assumption: the system is always able to tell whether an action has been executed completely or not by reading internal sensors. This is used to determine whether an EA or a VOA has caused a discrepancy: if no action has been executed but a discrepancy occurs it is assumed to be due to an EA, otherwise it is due to a VOA (combinations are not considered). When an EA occurs there is no need to adjust the model of the dynamics, instead only the knowledge about the current

situation is modified. This is done by replacing the axioms describing the initial situation $S_0$ with axioms describing the values of all ground fluents in the observed new state of the world. This is only possible in the face of complete knowledge about the initial situation. If the discrepancy is deemed to be due to a VOA, four cases are distinguished, two where the truth value of a fluent unexpectedly changed (positively or negatively), and two where it unexpectedly did not change (positively or negatively). In these cases the suggestion is to extend the successor state axioms, describing how fluents change in response to the execution of actions in the domain, according to the action that was performed and the discrepancy that was observed. This is done in a straightforward way: the conditions under which a fluent changes when the action in question is executed are either restricted by adding conjuncts or relaxed by adding disjuncts to the existing conditions. This does not seem to be a very favorable approach from the machine learning perspective as it does not perform any sort of generalization. While this may not be so critical with toy domains which can be modeled propositionally, in real-world systems the fluents of the model often involve many real-valued numbers, e.g., a position, and restricting learning to instances of these numbers will not be of any help in improving the model, as it is unlikely that the exact same state will be visited repeatedly. The assumption of perfect actuators, that is, assuming that the system always knows whether an action has been executed completely, is problematic, too. Some actions have indirect effects that only become observable sometime after the actual action was performed, but these effects would here be classified as exogenous and the system would thus never learn about their cause.

Wang [1994] is concerned with learning planning operators by observing the actions of an expert agent and "practicing" the newly acquired operators in the real-world to refine the model.[9] Observations consist of the state prior and the state after a named

[9]In fact, the author does not propose to practice in the real-world but a simulation thereof. This seems unreasonable though, because any simulation requires a model of the world itself. Thus if it is possible to implement and run such a simulator, the model learning problem has already been solved

action is executed. The system is learning in a specific-to-general manner by generalizing preconditions and effects. A number of assumptions are made: operators and states are deterministic, sensors are noise-free, the state is fully observable, preconditions are conjunctions of literals, and actions do not have conditional effects. In order to refine learned operators, the author proposes to solve practice problems in the environment to obtain more observations. While the paper does not elaborate on how to choose these practice problems, it does address the problem of planning using potentially over-constrained plan operators. When during planning the agent is uncertain about a conjectured but not yet verified precondition, the corresponding action may be considered applicable even when the conjectured precondition is not satisfied, in order to test the conjecture. A plan repair strategy that plans for open preconditions handles the potentially resulting execution failures.

Pasula *et al.* [2004] have addressed the same problem but in the presence of uncertainty about action outcomes. Given a set of (pre-state, action, post-state) examples, $(s, a, s')$, they greedily search for a best set of operators where the quality measure to maximize is the likelihood of the data given the operators, minus a term penalizing complex operator sets as determined through the number of preconditions and outcomes of all operators. The actual search is divided into three functions of which *LearnRules* is the main function. It initializes the search by creating operators for each tuple $(s, a)$ and then performs the search by specializing or generalizing these operators. Each time it creates a new operator it calls the second function *InduceOutcomes*. This function adds the specification of the outcomes (conjunctions of literals) to the operator. It again applies search to maximize the score (likelihood of data minus complexity) by merging compatible outcomes and removing redundant ones. Finally, the *LearnParameters* function adjusts the probabilities of these outcomes using a gradient method for optimization

and the simulator model could be adopted by the agent. Fortunately, for the purpose of our review it is irrelevant whether the practice happens in the real world or a simulation thereof.

with respect to the score. While no theoretical properties like convergence guarantees or complexity are analyzed, the authors compare the approach to a learning method for Dynamic Bayesian Networks (DBNs) showing that the presented approach learns the true distribution of outcomes better than DBNs for all considered training set sizes. They also demonstrate that relational operators, i.e., operators with variable arguments, are learned faster than purely propositional (ground) ones.

Pasula et al.'s approach can be classified as learning from specific-to-general. Alternatively, one can learn from general-to-specific as demonstrated, for the purpose of learning planning operators under partial observability, by Amir [2004] (also see [Amir, 2005]). There the approach is to start out with the set of all possible transition systems and filter this set by a given sequence of action-observation tuples, only keeping those transition systems consistent with the observations. Dealing with the explicit set of all possible transition systems (called the *transition belief state*) is not feasible as it is doubly exponential in the number of domain features and the number of actions. Instead, Amir represents the transition belief more compactly as a formula of propositional logic. The actual learning of plan operators is then defined as the progression of this *transition belief formula* through the actions in the given sequence and the filtering by conjunction with the made observations. This approach is, however, not likely to be applicable to the problem of model adjustment in the context of replanning. In model adjustment one starts with a specific transition system, the one used in planning, and it is unclear how one could generate a larger set of possible transition systems from that in order to make the described filtering approach applicable.

## 8.2 Contingency Planning

An alternative to the planning and execution monitoring approach we presented, is to pre-plan for all possible contingencies before beginning execution.

A desire for reactivity was the driving force for so called *universal plans*, proposed by Schoppers [1987]. The idea is simple: starting from the goal, the planner performs backward-chaining search until in each branch either a contradiction is produced or no open sub-goal (precondition of some action in the tree) has support, i.e., there is no action whose effects satisfy any of these open sub-goals. This produces a decision tree that dictates which action to perform in which state in order to reach the goal. The approach is indeed universal in the sense that for any state for which there exists a plan of getting to the goal, this plan can readily be read off the tree. This way, whatever goes wrong during execution of a plan, the system instantly knows how to react if it is possible to reach the goal from the new state.

Unfortunately, the price for this is high and makes the approach intractable: in the worst case, the time and space complexity is linear in the size of the domain, which, if not infinite, is exponential in the number of domain features. This approach strongly resembles the computation of policies in Markov Decision Processes, with a factored state space: In decision theory, one is concerned with optimally choosing actions in systems of stochastic state transitions in order to maximize a given utility function. The de-facto standard for representing these systems are Markov Decision Processes (MDPs) (see, e.g., [Puterman, 1994; Boutilier *et al.*, 1999]), when the state is fully observable, and Partially-Observable Markov Decision Processes (POMDPs), when the state is only partially observable. Solving an MDP (resp. POMDP) amounts to generating a *policy*, a control rule mapping states (resp. belief states) to actions in a way that maximizes the expected accumulated reward received from the states visited when following this rule. Policies are universal. Since they map every state to an action, an agent executing the policy always knows what to do next and this choice will be optimal and so no execution monitoring for dealing with run-time discrepancies is required, except for state estimation in the case of POMDPs. But the price for this is high: as with Schopper's Universal Plans the enumeration of all states makes the approach generally infeasible for

large or infinite domains.

The forward decision-tree search planner underlying our discussion in Chapter 6 is one way of avoiding this problem, by limiting the policy computation to (a subset of) the states reachable from a known initial state. In Section 6.6 we further reviewed a number of related methods for creating robust policies under time constraints.

Strongly related to our approach is the work on solving first-order MDPs by Boutilier *et al.* [2001] and more recently Sanner and Boutilier [2005]. This work proposes first-order decision-theoretic regression (FODTR) to solve First-Order MDPs exactly for all states, as opposed to a particular initial state. The approach works, roughly, by repeatedly regressing the value function and rewards, both represented as first-order formulae of a particular form, over stochastic actions, providing an improved value function for abstract states, where the abstraction is induced by the regression. There are certain practical limitations to this approach, in particular it does not generally allow for continuous domains. In real-world systems it further seems beneficial to follow a forward search approach to focus on the reachable subset of state space. As such our approach of Chapter 6 explores a middle-ground between this and plain decision-tree forward search.

When there remains uncertainty about the model applied in planning, there is a trade-off to be made. Either one exploits the current model, that is, tries to maximize the reward by behaving optimally according to it, or one can explore the environment in order to improve the model and benefit from this information gain in the future. This problem has been formalized and addressed in decision theory, reinforcement learning, and adaptive control (see for example [Duff, 2002, Chapter 2], for a survey). While the approaches have their appeal because of their rigorous mathematical foundation, they are limited in their expressiveness by using a transition function that maps ground states and ground actions to new states. For instance, a system performing action `pickup(a)` 1000 times does not learn anything about `pickup(b)`. This contrasts with the approaches presented in Section 8.1.3.

## 8.3 State Estimation

In this section we review selected previous work relevant to the problem of state estimation in the context of execution monitoring. We will first briefly review early work on static model-based diagnosis and then relate this to the problem of state estimation and the diagnosis of dynamical systems.

Model-based diagnosis was first described by Ray Reiter ([Reiter, 1987]). In Reiter's approach the system description, $SD$, is a finite set of first-order sentences and in this system a set of components, $COMP = \{c_1, \ldots, c_n\}$, exists, each of which may either perform abnormally ($Ab(c_i)$) or nominally ($\neg Ab(c_i)$). Given $SD$ and $COMP$, an observation $OBS$, represented as another finite set of first-order sentences, *conflicts* with the assumption that all components work correctly if $SD \cup \{\neg Ab(c_1), \ldots, \neg Ab(c_n)\} \cup OBS$ is inconsistent. The problem of diagnosis is then to find a subset $\Delta$ of the components such that assuming these components abnormal and the rest to work nominally, consistency of the union with $SD$ and $OBS$ is reestablished. The subset $\Delta$ is called a *diagnosis*. Generally there are several possible diagnoses in which case one is generally preferred over the others. In Reiter's approach this preference is defined through minimality, that is, a diagnosis $\Delta$ is preferred over another diagnosis $\Delta'$ if and only if $\Delta \subset \Delta'$. The preference for minimal diagnosis also makes sense from a probability point of view. Assuming that correct behavior of a component is more likely than its failure and that component failures happen independent of each other, minimal diagnoses are also most likely diagnoses.

[Witteveen *et al.*, 2005] have used this approach to diagnose abnormal events during the execution of agent plans. The system description models the dynamics of the domain, in particular the effects of the agent's actions, and the components are the agent's actions themselves. In this setup the independence of component failures is not given anymore, as the failure of one action can cause others to fail to. The authors accommodate for this fact by introducing the notion of *pareto minimal causal diagnosis*. Roughly, the original

diagnoses are reduced to their causes when a subset of components in the diagnosis causes the failure of other components in the set. Minimality is then determined based on the reduced sets.

Reiter later extended his original work with de Kleer and Mackworth to allow for fault models and exoneration axioms, which violate the assumption implicitly made in the original work that also every super-set of a diagnosis is a diagnosis itself [de Kleer *et al.*, 1992]. The main approach presented in this paper was based on the notion of prime implicants and the diagnoses defined from that were called *kernel diagnoses*.

Another way of defining the most probable diagnosis is the introduction of explicit numeric failure probabilities. The generation of candidate diagnoses can then be focused to the most likely ones [de Kleer, 1991].

The task of state estimation is arguably similar in nature to the above described problem of diagnosis: given some observation that gives rise to the suspicion that the actual current state is not the one we expected, we would like to identify the actual state as best we can. As such the above described diagnosis is a special case of state estimation where the incompleteness of knowledge is limited to the abnormality of the components. It is also limited as it does not provide for dynamics in the system, that is, it is only possible to talk about what holds or does not hold *now* but not what may have happened in the past to reach the current state.

McIlraith addressed this misfit in [McIlraith, 1997]. She combined above work with the situation calculus to model action dynamics and introduced the notion of explanatory diagnosis. Given a basic action theory in the situation calculus $\Sigma$, a history of actions $HIST$ that is known to have happened since the initial state $S_0$, and an observation $OBS$, an *explanatory diagnosis* is a sequence of actions $E = \alpha_1; \ldots; \alpha_n$ such that $\Sigma \models Poss(HIST; E, S_0) \wedge OBS(do(E, do(HIST, S_0)))$. That is, an explanatory diagnosis is a sequence of actions that may have happened following $HIST$ and explains the observations. These actions are assumed to be *exogenous*, that is, not under the control

of the monitored agent. McIlraith shows that the problem of finding an explanatory diagnosis coincides with the problem of planning: The system dynamics are described by the action theory $\Sigma$, the initial state is the state resulting from executing $HIST$ in the initial state $S_0$, $do(HIST, S_0)$, and the goal is described by the observations $OBS$.

McIlraith's work was extended by [Iwan, 2000; Iwan and Lakemeyer, 2003]. Iwan argued that in order to explain the observations, it is sometimes not enough to conjecture the occurrence of exogenous actions after the given history of actions $HIST$ but that in order to explain the observations one has to assume the occurrence of exogenous actions in between the given action history and/or that some actions in this sequence were not performed as expected. Iwan also addressed the problem of characterizing and computing the most preferred diagnosis using explicit probabilities for the occurrence of events, much like [de Kleer, 1991] proposed for the static case. The computation is based on best-first forward search.

One shortcoming of these approaches is their limited applicability to real-world systems involving continuous evolutions of real valued features, like for instance the three-dimensional position coordinates of a helicopter in operation. Consequently, the majority of deployed robotic systems described in the literature approach the problem of state estimation differently. The main difference is the representation of the dynamics: instead of sets of logical sentences for representing states and some kind of effect axioms for describing the effects of actions in the domain, these systems describe the state by the numeric values of certain properties and control values, and use algebraic or differential equations to describe the impacts these properties have on each other depending on the current mode of operation. This way not only continuous values and continuous time becomes manageable, but also continuous probability distributions can be modeled to capture the uncertainty of the domain.

This approach was used in [McIlraith, 2000] and [de Freitas *et al.*, 2004]. In both these papers the system to be diagnosed was modeled as a hybrid system, that is, using a

state representation that has both a discrete and a continuous part. The continuous part generally describes the dynamics, whereas the discrete part describes the operational modes. The latter induce different dynamics in the continuous part, for instance the direction of travel of a robot. Faults are defined through fault modes in the discrete part and cause the dynamics of the continuous part to change. Observations, on the other hand are generally only made in the continuous part, except for deliberate control mode changes. The task then is generally to infer if and when a faulty mode has materialized and reasoning is generally based on the belief state, that is, a probabilistic distribution over possible system states.

In [McIlraith, 2000], McIlraith casts the problem of diagnosing a hybrid system as a Bayesian model tracking and selection problem. To efficiently track multiple models simultaneously, she proposes the use of particle filters. One major problem with the use of particle filters for diagnosis is that they focus on the most likely models, that is the nominal behavior, while fault modes are unlikely and can therefore slip the attention of the – approximate – filter. McIlraith overcomes this problem by biasing the samples towards the results of a separate, qualitative diagnosis as it is described in [McIlraith *et al.*, 2000]. The paper makes a single-fault assumption.

Particle filters now are a very common approach to approximate the distribution of the state variables. These can also be integrated with exact methods like for instance Kalman filters into so called Rao-Blackwellised particle filter (see, e.g., [de Freitas *et al.*, 2004]). [Verma *et al.*, 2002] combine particle filters with Partially Observable Markov Decision Processes (POMDP) for controlling a system: a policy for the POMDP is computed off-line while particle filters are used on-line to track the belief state. Particle filters can also be used for approximate inference in Dynamic Bayesian Networks (see, e.g., [Russell and Norvig, 2003], pp. 565–568).

Depending on the way the system is modeled and on the available observations, diagnosis may not be required or can be trivial. This is for instance the case when the state can

be sensed completely, or when it can be sensed partially and the observations coincide exactly with the predictions of the model. Since then there is no discrepancy, there is no reason to believe that the actual state is any different from the predicted one. In any case, the situation-dependent need and requirements for diagnosis should be guided by its purpose, that is, in the case of execution monitoring, it should be determined with respect to the following steps, state evaluation and replanning. If, for instance, state evaluation is able to specify a sub-set of states in all of which the current plan should be continued, then there is no need to disambiguate between two candidate diagnoses when both candidates belong to this sub-set. This is in fact achieved, for instance, by the explicit annotation of a sufficient and necessary condition for the continued plan validity, as described in Chapter 3.

[Boutilier, 2000] described a decision-theoretic model of monitoring the preconditions of actions in a plan during execution to determine whether or not the current plan should be continued, and proposed heuristic methods to make this otherwise intractable problem tractable for more than just very short plans. This was motivated by three drawbacks with existing approaches[10]: (i) they generally ignore the monitoring cost, (ii) they do not account for monitoring errors (noisy sensors), (iii) they ignore the fact that a previously false precondition of an action later in the plan may be reestablished at the time when this action is to be executed (e.g., hearing about a traffic jam on a route that won't be reached for several hours). To address these concerns Boutilier modeled the decision of continuing or abandoning the plan as a POMDP. The state space of this POMDP is the set of vectors of truth values for all preconditions in the plan. At every stage of plan execution the POMDP can choose for each precondition in the remainder of the plan whether to monitor it or not, and after monitoring is done, whether to continue executing the current plan or to abandon it in favor of adopting the best alternative at this point.

[10]including approaches that replan whenever an unexpected state is reached

The approach requires the availability of certain information, part of which may be difficult to obtain in practice:

- For any point in the plan the value of the best alternative plan at that point has to be known, since it will be used as the value for abandoning the current plan. This is not generally known as common planners do not provide this information since that would incur greater planning time costs. Many, in particular currently popular heuristic search based planners, would provide an upper bound on this value however.

- The probability of possible failures has to be known. Although these probabilities certainly exist, estimating them may be difficult. This assumption also implies that the agent is aware of all possible faults and this again is not generally the case in the real world.

- A monitoring (e.g., sensor) model has to provide the likelihood for a particular sensor reading for the case that a particular precondition has failed, and for the case that it has not failed. This is to allow for monitoring errors (e.g., noisy sensors).

The general POMDP defined this way is too complex to be tractably solvable. Instead Boutilier proposed decomposition and approximation techniques. These make the plan monitoring problem solvable in a reasonable amount of time, even for long plans involving hundreds of steps, while compromising only little on quality as shown by experiments.

Boutilier assumed away many complicating factors to keep the presentation simple. It is clear that the approach generalizes to cases where these assumptions do not hold, but it is unclear how that would affect the complexity. For instance are all preconditions assumed to be established prior to plan execution, i.e., the system does not have to reason about which actions are establishing preconditions of later ones and adjust the monitoring decisions accordingly. Also are conditional effects left out of the picture this way, but it is generally not enough to just monitor the preconditions of actions,

but also the conditions under which certain desired or required effects are produced are relevant. Another limitation is the fact that alternative plans are not monitored. This is problematic because if during execution an alternative becomes better than the current plan, we should adopt it. Also if the best alternative decreases in value, this should affect our decisions as it may no longer be advisable to adopt the current alternative plan just yet when some future precondition of the current plan is expected to fail. Overall, Boutilier addresses the question "whether" and "when" to monitor relevant conditions, but he does not cover the question "which" conditions are relevant. By assumption the set of relevant conditions is already given. The regression based approach we have presented in this thesis, may be used to ameliorate this issue and identify what needs to be monitored, including ramifications due to conditional effects.

## 8.4 External Monitoring

There are a number of approaches of what might be best described as "external monitoring", where the monitoring and the executing entity are independent, meaning that the monitor watches the agents behavior by means of certain sensors, but does not have access to the agents internal state. In particular the monitor is not informed of the actions being executed but can only try to infer this information from observing the actions' effects. Although this was proposed in the literature to enable one agent to monitor other agents, this work may still be of interest even in frameworks like ours where we do not separate these two entities, as these approaches also provide insight into the diagnostic problem of deciding "what happened". But in contrast to the approaches in Section 8.3, the approaches presented here also evaluate the found diagnoses.

The Autominder system [Pollack *et al.*, 2003] is a cognitive orthodic helping people with memory impairment by issuing personalized reminders to help the client accomplish her daily agenda. The system observes the client's actions to infer whether a reminder is

necessary or not. The authors of the system argue that since sensory input is noisy and the action in the client's plan (agenda) can have complex temporal constraints, reasoning under uncertainty and reasoning with quantitative temporal relationships between events has to be integrated for this task. For this purpose they introduce Quantitative Temporal Dynamic Bayesian Networks (QTDBNs; [Colbry *et al.*, 2002]), an integration of Time Nets and Dynamic Bayesian Networks (DBN). The Time Net component models the relationship of the time intervals client actions occur in, representing the probability that an event occurs in a certain time interval, while the DBN reasons about actions, sensors, and domain properties within any such time interval. The DBN is updated with the arrival of new information, i.e., sensory input, while the Time Net is updated only as the actual time crosses a time interval boundary. At these time changes two interface functions pass information from the DBN to the Time Net and back. Experimental results show that for sufficiently small tasks, QTDBNs achieve their purpose of monitoring the execution of another agent's plan – here the client's agenda –, but the required Bayesian reasoning, which is known to be NP-hard, causes a time complexity exponential in the number of modeled actions.

To monitor the execution of a multi-agent system, [Dix *et al.*, 2003] investigate an approach that could be described as meta-planning. Again unaware of the internal states of the participating agents in the system, the authors propose to create a set of *intended plans* off-line in a meta-theory that reasons about messages passed between the individual agents partially revealing the executed actions. On-line, these intended plans are filtered based on the messages actually passed between agents. That is, when a new message is observed, the set of intended plans is reduced to those plans compatible with this observation. The system raises an alert to the user if the set becomes empty, to indicate that according to the meta-theory there is no way the monitored system is going to achieve its objectives, or when no message has arrived for a specified time, indicating that the system is stuck. Hence, instead of proving that the monitored system performed

nominally, the approach here is to try to prove the opposite and assume nominal operation as long as there is at least one possible way of reaching the goal, i.e., at least one compatible intended plan. This approach demonstrates that monitoring is possible even when the applied models in the execution system and the monitoring system are different, which contrasts to the approach in Autominder, where the client's plan is known precisely and taken into account for monitoring. Instead, the monitoring system here merely bases its decisions on the objectives of the agents being monitored and a rough indication on which actions are being performed, perceivable from the passed messages. It also allows monitored agents to change their plans, as long as there remains a way of reaching the goal. Hence the goal and not the plan is used to determine failure.

## 8.5 Meta-reasoning

In the face of limited computational resources, a rational agent interleaving planning and execution in a dynamic world should also be aware that deliberation itself impacts the state of the world. This is because the world evolves while the agent deliberates. In many applications it may thus be sometimes beneficial to commit to a seemingly sub-optimal plan quickly, because determining the optimal plan may cost[11] more than the potential gain, namely when time directly or indirectly affects the preferences of the agent. Reasoning about the agent's own reasoning process and capabilities is called *meta-reasoning* ([Russell and Wefald, 1991]). In the context of execution monitoring we may be faced with questions of meta-reasoning when monitoring plan optimality: It may be that the current plan, while still valid, has been found to be sub-optimal. Then it may still be optimal (in the meta sense) to continue its execution, namely when replanning will cost more than the potential gain. Another aspect of meta-reasoning concerns the evaluation step itself: if the evaluation incurs costs, e.g., by interrupting the exe-

[11] We use the terms "cost" and "gain" loosely here and understand them to stand for any negative, resp. positive, impact on the agents preference criteria.

cution, it may be beneficial to omit evaluation and any subsequent replanning entirely. [Russell and Wefald, 1991] suggest two main applications for meta-reasoning: (a) enabling the agent to decide on-line which computations to perform and which not, but also (b) analyzing the rationality of a system design. The latter may be used in future work to study approaches like the one of Chapter 5, in terms of their overall optimality.

## 8.6 Control Theory

The high-level objective of control theory – "control a system such that it behaves in a particular way" – is very similar to ours. The main difference between control theory and AI planning and execution monitoring approaches are the applied mathematics (cf., e.g., [Dean and Wellman, 1991]). Typically in control theory states are represented through the values of continuous variables, and "control laws" map the current state and current time to control values, much like policies do in decision theoretic planning. While the dynamics of the controlled system can generally be characterized by algebraic, often differential, equations, the use of an explicit model is uncommon. Often the *error*, $e(t)$, the difference between the current state and the desired state at time $t$, is used directly to control the system. The very popular proportional-integral-derivative (PID) controller, for example, defines the control value to take in terms of a linear combination of the error itself, $e(t)$, the accumulated error, $\int_t e(t)$, and the error's gradient, $\frac{de(t)}{dt}$. Apart from the difference in applied techniques, research in control theory is also generally concerned with other questions such as the controllability, stability, or diagnosability of a system. Similarities exist in so-called optimal control which is concerned with optimizing the systems behavior with respect to some "cost index", the counterpart to preferences in AI planning. Finally, adaptive control techniques address the problem of refining the controller automatically, by adjusting the control parameters during operation as necessary.

# Chapter 9

# Conclusion

## 9.1 Summary

When executing plans in dynamic environments, discrepancies between the expected and actual state of the world can arise for a variety of reasons, including imprecise models, noisy sensors, or exogenous events not under the control of the agent. When such circumstances cannot be anticipated and accounted for during planning, they bring into question whether discrepancies are relevant, and whether they render the current plan invalid or sub-optimal. While there are several approaches for monitoring validity, no approaches exist for monitoring optimality. Instead it is common practice to replan when a discrepancy occurs or to ignore the discrepancy, accepting potentially sub-optimal behavior. Time-consuming replanning is impractical in highly dynamic domains and many discrepancies are irrelevant and thus replanning unnecessary, but to maintain optimality one has to determine which these are.

In practice, this problem may not only occur during execution, but during planning as well. Even while no plan has been found yet and before execution begins, the state of the world may change in unexpected ways. This again raises the question of how to react, as the planning effort up to this point may or may not be invalidated by the discrepancy.

This question is not well studied in the literature.

## 9.2 Contributions

This thesis contributes to these problems in several ways. Using the formal notion of goal regression, we were able to summarize a number of existing approaches for monitoring plan validity during execution. From this we abstracted a general abstract approach for monitoring more general objectives, including plan optimality with respect to given user preferences. This approach is based on annotating the plan with conditions for the continued satisfaction of the objectives, which are checked during execution. We then showed the broad applicability and benefit of this approach by developing five concrete applications of this abstract approach. A central intuition behind our approach is that many discrepancies are completely irrelevant, while others only affect small parts of the plan or search tree. Exploiting this structure allowed us to gain significant computational speed-ups in several practically interesting cases.

**Monitoring plan validity.** The provision of the formal characterization of the existing approaches for plan validity as goal regression (Section 3.4) enables its exploitation with other planners, such as very effective heuristic forward search planners. It further allowed us to formally prove the correctness of the approach.

**Monitoring plan optimality during execution.** We showed how the abstract approach can be beneficially extended to the significantly more difficult problem of monitoring plan optimality (Chapter 4). Since optimality is relative rather than absolute, this required us to regress over plan alternatives as well, and in order to measure the relative quality of alternatives, the evaluation function had to be regressed, too. We derived a sufficient condition for optimality, given an admissible heuristic, and proved its correctness. From this we derived plan annotations and an algorithm that can be used during plan execution to verify the continued optimality

of the plan. We proved that the algorithm makes sound decisions regarding this optimality. Our empirical results show that a significant speed-up can be gained by our approach compared to replanning from scratch in order to make this decision.

**Generating optimal plans in highly dynamic environments.** In practice, in highly-dynamic environments it is necessary to monitor the state of the world during planning itself as well. The question of how to react when the state of the world unexpectedly changes during planning has been largely ignored in the literature. In Chapter 5 we made three contributions regarding this problem:

1. We showed that the abstract monitoring approach we propose can be applied to devise a planner that is able to monitor the environment for relevant changes, while planning. The basic idea is to use regression to reason about all relevant entities, including accumulated action costs and heuristic function, and to annotate the search tree with this information. Then, when discrepancies occur, this information can be used to pinpoint possibly affected search nodes, whose formulae can then be reevaluated as necessary to update the search tree according to the events. We proved that this update is correct, in the sense that when planning is continued from the updated data structures, the produced plan is optimal with respect to the new initial state of the world.
2. We introduced a new criterion for evaluating plan adaptation approaches—the relative replanning time compared to the amount of change—and argued for its practical significance.
3. Finally, we presented empirical results that demonstrate that our approach is able to produce optimal plans in cases where repeated replanning would fail, due to the high frequency of expected state changes.

**Monitoring policy execution in stochastic domains.** Since in practice certain actions may be known to have stochastic outcomes, it is often advisable to deploy some

form of conditional planning, where planning is performed for several possible evolutions of the world. In Chapter 6 we demonstrated that also for decision-theoretic planning using forward search from a known initial state, the proposed abstract monitoring approach can be used to (partially) compensate for unpredictable state changes. We again developed the annotation and recovery algorithm required to update an existing search tree when discrepancies occur. We proved that the annotation does not significantly increase the memory required to store the search tree. We also demonstrated the use of the approach in two concrete MDP applications: stochastic shortest-path problems, and sampling-based forward search solutions for MDPs with large or infinite state spaces. Finally, experimental results demonstrate that the proposed update of the search tree can indeed be done significantly faster than the alternative of re-spanning the search tree through replanning from scratch.

**Generating and executing plans with procedural control.** In practice, the user may desire to control the behavior of the agent in a procedural way. This has proved useful in several practical applications and ConGolog has been demonstrated to be a well suited language to express such control. However, special purpose machinery is required to reason about ConGolog programs, both in planning and execution monitoring. In Chapter 7 we proposed a compilation approach that takes a basic action theory of the situation calculus and a ConGolog program, and produces a new basic action theory whose tree of situations describes exactly the subset of executable situations of the original theory which are permitted by the program, which we formally proved. The compilation result is at most quadratic in the size of the program and can be computed in quadratic time. Since the resulting theory is an ordinary basic action theory, a number of reasoning tasks that can be performed on these can now be done over programs as well. One such task is plan generation, but also the proposed abstract monitoring approach benefits from this. In particular, the approach can now be used to monitor the execution of plans resulting from the

interpretation of ConGolog programs, in order to verify the continued satisfaction of the hard constraints described by the program.

A key feature of the monitoring approach proposed in this thesis, is its ability to handle *any* perturbation to the fluents describing the state of the world, rather than being limited to some pre-defined set of contingencies, and its running time is independent of the cardinality of possible events. This makes it attractive for real-world applications, where the set of all possible contingencies is hard to determine or simply too big to plan for—in continuous domains, e.g., there may be infinitely many such contingencies.

Relevant properties of a situation, as provided in the annotation, can further serve to focus on-line sensing when faced with limited sensing resources: the agent knows which features to sense and which can simply be ignored as they do not influence its objective.

The requirements for the applicability of our method are easy to fulfill: The used action language has to be expressive enough to represent the user's preferences and regression has to be defined. However, in order to be used with heuristic search, the heuristic function needs to be represented as a regressable formula (cf. [Reiter, 2001, Definition 4.5.1,p.62] for a definition of regressable formulae in the situation calculus). This does not seem immediately possible for some heuristics that have recently been popular in the planning community, since these heuristics are often defined algorithmically rather than by a closed form formula. Finding and applying successful heuristics that are compatible with our approach is a topic of future work. One such heuristic might be pattern databases (see, e.g., [Edelkamp, 2001]). Heuristics which are themselves based on some form of regression (e.g., [McDermott, 1999; Bonet and Geffner, 2001] and [Refanidis and Vlahavas, 2001]) may also be more amenable to our approach.

In practice, the memory requirements of our approach may be a bottleneck, when the regressed formulae cannot be represented compactly (cf. Section 2.2). At least for the majority of the domains of the International Planning Competition, however, this does not seem to be the case, since these domains are described in STRIPS, which, as dis-

cussed, guarantees succinct regression results. This also applies to a possible comparison to incremental heuristic search methods, which are generally only applicable to simple planning domain descriptions and search problems. In such problems, we conjecture the memory requirements of our approach to not be problematic from a practical point of view either.

While in our exposition we focus on optimal plan that are generated using a monotone heuristic in conjunction with $A^*$ search, our approach can equally be used in conjunction with non-monotone heuristics (e.g., weighted $A^*$ [Pohl, 1970]), which are often found to speed up planning at the cost of optimality. The resulting plans are guaranteed to be no more sub-optimal than a certain bound which depends on the degree of inadmissibility of the heuristic. In this context, the presented monitoring approach can be used to verify the continued satisfaction of this bound on sub-optimality, given unexpected changes in the environment.

## 9.3 Future Work

We have already pointed out a number of possible directions for future work in each chapter. We here list more general directions, in particular those that are shared by all the pieces, and those that apply to the thesis as a whole.

Since the thesis is concerned with issues that arise in practice, and aims at contributing to their solution, these solutions should be evaluated on applications in highly dynamic real-world domains, too. Such domains could, for instance, be the mentioned robotic soccer domain (RoboCup) or the high-level control of Unmanned Aerial Vehicles. We believe success in RoboCup in particular, may strongly benefit from our approach, as we anticipate it will drastically improve successful generation and execution of plans, i.e., goal scoring. Also the focusing of sensing activities bears potential in this domain, an aspect we want to investigate further.

An optimized version of our implementation may be required for some of these applications. All current implementations are in Prolog and immediately amenable for integration with existing on-line Golog interpreters (e.g., Readylog [Fritz, 2003]). However, since most current state-of-the-art planners deploy progression to solve the projection problem, our use of regression may be an obstacle for easy integration of our methods with those planners. There may however be ways of exploiting regression also to improve planning itself (for one possibly interesting such direction see [Fritz, 2008]), in which case the issue would lose significance. In this context it would also be interesting to study popular, existing heuristics in terms of their regressability—which is a prerequisite shall our method be used with heuristic search. Also related work in decision-theory (e.g., [Boutilier *et al.*, 2001], see Section 8.2) may benefit from such an investigation. As mentioned previously, in this regard heuristics based on pattern databases (e.g., [Edelkamp, 2001]) could prove to be a good starting point.

Even after contributing possible solutions to a number of problems regarding the use of planning technology in practical applications, there remain a number of open questions. For some of these, an adequate solution may again be derived from the techniques presented in this thesis. One such problem is that of how to decide whether an action has executed completely and successfully. Since many actions in the real-world have durations which are generally very difficult to predict, it is necessary to device a monitor that determines the completion of actions. In practice, this decision is often based on the materialization of expected post-conditions (effects). However, as we argued in this thesis, unexpected events may jeopardize these effects, invalidating this approach of determining action completion. Indirectly, we already made a first immediate contribution to this problem in Chapter 3, which is suggestive of a method for deciding action completion, based on the satisfaction of the annotated condition. However, also then a question would remain: how long should the monitor wait? When has an action ultimately failed?

This problem becomes even harder when monitoring plan optimality, and the lack of a closed form necessary condition for continued plan optimality in our approach limits its use.

Some researchers argue that in practice it is often the so-called wall-clock time that needs to be minimized, i.e., the absolute time that it takes to both generate and execute a plan successfully. When optimality is defined in terms of the temporal length of the plan, our method may again have merits in this context, as it allows one to determine bounds on the degree of sub-optimality of an executing plan when unexpected changes happen. If an estimate exists on how long it would take to find a better plan, this information can be used to make a decision as to whether to continue with the sub-optimal plan, or whether replanning would reduce wall-clock time to completion.

Another problem, not addressed in this thesis, but of great practical relevance, is that of changing goals, including the arrival of new goals, during both planning and execution. While our approach is able to handle changes in the state of the world, which can also be used to model changes in the feasibility and cost of actions, it is not obvious how this might be extended to handle changing goal specifications as well. However, if instead of forward search, backward search was deployed, then a counterpart of our approach in this setting—based on progression instead of regression—may again be beneficial.

Finally, as we have pointed out earlier in the context of related replanning approaches, our approach does not improve its model of the dynamics of the world when unexpected changes happen, in order to explain and avoid these changes in the future. This provides another interesting direction of future research and we have discussed some promising approaches that could be built on in Section 8.1.3.

Our discussion on Chapter 5 shows that the notion of optimality is not trivial to define in domains where unexpected changes may occur and real-time constraints exist. This links to the explore-exploit trade-off investigated in decision-theory, and the problem of meta-

reasoning. The difficulty of arriving at a satisfactory definition of optimality seems to stem from the fact that this is technology dependent: depending on the abilities of the (re-)planner, the optimal strategy may be a different one. We hope that our considerations regarding replanning-convergence in that chapter may add to this discussion, but believe that further theoretical study of the problem is necessary.

We conjecture that the ideas behind the presented approach may be beneficially applied to planning under initial state uncertainty, in particular when such uncertainty ranges over continuous domains. Since our approach relies on the identification of what is relevant in the assumed current state of the world for plan success, it is conceivable that this information can be used as well, to build a sampling based approach for planning under incomplete knowledge about the initial state: After planning for one possible initial state, our approach may be used to generalize from this state to a larger set. If an appropriate sampling strategy can be found, this may help in covering large parts of the initial belief state more quickly.

Further, as pointed out earlier and above, the relevance information may benefit other decisions as well. In particular the decision of a situated agent on what to sense, can be greatly alleviated. This may be extended to the more general problem of deciding sensor placement, or which sensing results to transmit in a sensor network, given a particular monitoring task. The latter may be interesting, since such networks generally need to economize on the energy expended in order to transmit sensing results back to the station. Hence it is desirable to limit sensing to only relevant features.

Finally, a number of related application domains may require similar monitoring techniques as the ones presented in this thesis. For instance, the execution of business processes or work-flows is prone to execution discrepancies. This again suggests a monitoring approach that is able to determine the relevance of these discrepancies. In this context, the monitoring of ConGolog programs discussed in Chapter 7 may again prove

useful, as ConGolog has recently been suggested to be particularly well suited for the modeling, planning, and execution of business processes [de Leoni *et al.*, 2007].

# Bibliography

[Ambros-Ingerson and Steel, 1988] J.A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI)*, pages 83–88, 1988.

[Amir, 2004] E. Amir. Learning partially observable action models. In *Proceedings of The 4th International Cognitive Robotics Workshop, at ECAI-2004, Valencia, Spain*, 2004.

[Amir, 2005] E. Amir. Learning partially observable deterministic action models. In *19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 1433–1439, 2005.

[Bacchus and Kabanza, 1998] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

[Baier and McIlraith, 2006] Jorge A. Baier and Sheila A. McIlraith. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pages 788–795, Boston, MA, 2006.

[Baier *et al.*, 2007] Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 26–33, Providence, Rhode Island, USA, September 22 - 26 2007.

[Barto *et al.*, 1995] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. In *Artificial Intelligence, Special Volume on Computational Research on Interaction and Agency, 72(1)*, pages 81–138, 1995.

[Beetz and McDermott, 1994] M. Beetz and D. McDermott. Improving robot plans during their execution. In K. Hammond, editor, *Second International Conference on AI Planning Systems*, pages 3–12, Morgan Kaufmann, 1994.

[Beetz and McDermott, 1996] M. Beetz and D. McDermott. Local planning of ongoing activities. In Brian Drabble, editor, *Proceedings of the Third International Conference on AI Planning Systems*, pages 19–26, Morgan Kaufmann, 1996.

[Bertsekas, 1995] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

[Bjäreland, 1999] M. Bjäreland. Recovering from modeling faults in GOLOG. In *IJCAI'99 Workshop: Scheduling and Planning Meet Real-Time Monitoring in a Dynamic and Uncertain World, Stockholm, Sweden, August 1999*, 1999.

[Bjäreland, 2001] Marcus Bjäreland. *Model-Based Execution Monitoring*. PhD thesis, Linköping University, Schweden, 2001.

[Blum and Furst, 1995] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, Montréal, Québec, Canada, August 20–25 1995. Morgan Kaufmann, 2 Volumes.

[Bonet and Geffner, 2001] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[Boutilier *et al.*, 1999] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Boutilier *et al.*, 2000] Craig Boutilier, Ray Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 355–362, Menlo Park, CA, July 30–3 2000. AAAI Press.

[Boutilier *et al.*, 2001] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence 2001*, pages 690–700, 2001.

[Boutilier, 2000] Craig Boutilier. Approximately optimal monitoring of plan preconditions. In *Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence (UAI'00)*, pages 54–62, Stanford University, Stanford, California, USA, June 30–July 3 2000. Morgan Kaufmann.

[Burgard *et al.*, 1999] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2):3–55, 1999.

[Claßen and Lakemeyer, 2008] Jens Claßen and Gerhard Lakemeyer. A logic for nonterminating Golog programs. In *Proceedings of the 11th International Conference on Knowledge Representation and Reasoning (KR)*, Sydney, Australia, 2008.

[Colbry *et al.*, 2002] D. Colbry, B. Peintner, and M. E. Pollack. Execution monitoring with quantitative temporal bayesian networks. In *Proc. of the 6th International Conference on AI Planning and Scheduling*, pages 194–203, 2002.

[Cushing and Kambhampati, 2005] William Cushing and Subbarao Kambhampati. Replanning: A new perspective. In *Poster Program, ICAPS 2005*, 2005.

[Darwiche and Marquis, 2002] A. Darwiche and P. Marquis. A knowledge compilation map. In *Journal of AI Research*, 2002.

[de Freitas *et al.*, 2004] Nando de Freitas, Richard Dearden, Frank Hutter, Ruben Morales-Menendez, Jim Mutch, and David Poole. Diagnosis by a waiter and a mars explorer. Invited paper for Proceedings of the IEEE, Special Issue on Sequential State Estimation, 92(3):455-468, 2004.

[De Giacomo *et al.*, 1998] Giuseppe De Giacomo, Ray Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 453–465, 1998.

[De Giacomo *et al.*, 2000] Giuseppe De Giacomo, Yves Lespérance, and Hector Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.

[de Kleer *et al.*, 1992] J. de Kleer, A.K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[de Kleer, 1991] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 842–848. AAAI Press / The MIT Press, ISBN 0-262-51059-6, Volume 2, July 14–19 1991.

[de Leoni *et al.*, 2007] Massimiliano de Leoni, Massimo Mecella, and Giuseppe De Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. In *Proceedings of Business Process Management*, pages 182–197, 2007.

[Dean and Wellman, 1991] Thomas Dean and Michael Wellman. *Planning and Control*. Los Altos, Calif.: M. Kaufmann Publishers, 1991.

[Dean *et al.*, 1995] Thomas Dean, Leslie Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. In *Artificial Intelligence, Volume 76, Number 1-2, Pages 35-74*, 1995.

[Dearden and Boutilier, 1994] Richard Dearden and Craig Boutilier. Integrating planning and execution in stochastic domains. In *Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning*, pages 55–61, Stanford, CA, 1994.

[Dix *et al.*, 2003] Jürgen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang. Monitoring agents using declarative planning. In *Proceedings 26th German Conference on Artificial Intelligence (KI 2003)*, pages 646–660, University of Hamburg, Germany, September 15–18 2003. LNCS/LNAI 2821.

[Doyle *et al.*, 1986] Richard J. Doyle, David Atkinson, and Rajkumar Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI'86)*, pages 81–88, Philadelphia, PA, USA, August 11–15 1986. Morgan Kaufmann, Two Volumes, Volume 1: Science.

[Duff, 2002] Michael Duff. *Optimal Learning: Computational Procedures for Bayes-adaptive Markov decision processes.* PhD thesis, University of Massachusetts Amherst, January 2002.

[Earl and Firby, 1997] Charles Earl and Jim Firby. Combined execution and monitoring for control of autonomous agents. In *Proceedings of the First International Conference on Autonomous Agents*, pages 88–95, Marina del Rey CA, USA, February 1997.

[Edelkamp, 2001] S. Edelkamp. Planning with pattern databases. In *Proceedings of the $6^{th}$ European Conference on Planning*, pages 13–24, 2001.

[Eiter *et al.*, 2004] Thomas Eiter, Esra Erdem, and Wolfgang Faber. Plan reversals for recovery in execution monitoring. In *Proceedings of The 10th International Workshop on Non-Monotonic Reasoning (NMR 2004)*, Whistler, Canada, June 6–8 2004.

[Erol *et al.*, 1994] Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, 1994.

[Ferguson *et al.*, 2005] Dave Ferguson, Maxim Likhachev, Geoff Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pages 262–271, Monterey, California, USA, June 5–10 2005.

[Ferrein *et al.*, 2004] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic Golog for unpredictable domains. In *Proceedings of 27th German Conference on AI (KI)*, pages 322–336, Ulm, Germany, September 20-24 2004. Also appeared in Proceedings of The 4th International Cognitive Robotics Workshop, at ECAI-2004, Valencia, Spain.

[Fichtner *et al.*, 2003] Matthias Fichtner, Axel Grossmann, and Michael Thielscher. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57(2–4):371–392, 2003.

[Fikes and Nilsson, 1971] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1971*, pages 608–620, 1971.

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[Fox *et al.*, 2006] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *Proceedings of The International Conference on Automated Planning & Scheduling, Lake District, U.K., June 6–10*, 2006.

[Fritz, 2003] Christian Fritz. Integrating decision-theoretic planning and programming for robot control in highly dynamic domains. Master's thesis, RWTH Aachen, Germany, November 2003.

[Fritz, 2008] Christian Fritz. Finding state similarities for faster planning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI)*, pages 925–930, Chicago, Illinois, USA, July 13–17 2008.

[Funge, 1998] John Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, Toronto, Canada, 1998.

[Gabaldon, 2002] Alfredo Gabaldon. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 2002.

[Gerevini and Serina, 2000] Alfonso Gerevini and Ivan Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, pages 112–121, Breckenridge, CO, USA, April 14–17 2000.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[Golden *et al.*, 1996] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. Technical Report 96-01, University of Washington, February 1996.

[Hanks and Weld, 1995] Steve Hanks and Daniel S. Weld. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research (JAIR)*, 2:319–360, 1995.

[Hart *et al.*, 1968] P.E. Hart, N. Nilsson, and B. Raphael. A formal basis for the determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[Hoffman and Nebel, 2001] J. Hoffman and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Iwan and Lakemeyer, 2003] G. Iwan and G. Lakemeyer. What observations really tell us. In *KI 2003: Advances in Artificial Intelligence*, pages 194 – 208, 2003.

[Iwan, 2000] Gero Iwan. Explaining what went wrong in dynamic domains. In *Proceedings of the 2nd International Cognitive Robotics Workshop*, 2000.

[Kambhampati, 1990] Subbarao Kambhampati. A theory of plan modification. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI'90)*, pages 176–182, Boston, Massachusetts, July 29–August 3 1990. AAAI Press / The MIT Press, ISBN 0-262-51057-X, 2 Volumes.

[Kearns *et al.*, 1999] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1324–1231, 1999.

[Koenig *et al.*, 2002] Sven Koenig, David Furcy, and Colin Bauer. Heuristic search-based replanning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS'02)*, pages 294–301, Toulouse, France, April 23–27 2002.

[Kuter *et al.*, 2004] Ugur Kuter, Evren Sirin, Dana S. Nau, Bijan Parsia, and James A. Hendler. Information gathering during planning for web service composition. In *Proceedings of the 3rd International Semantic Web Conferece (ISWC)*, pages 335–349, 2004.

[Lazovik *et al.*, 2003] Alexander Lazovik, Marco Aiello, and Mike Papazoglou. Planning and monitoring the execution of web service requests. Technical Report DIT-03-049, Informatica e Telecomunicazioni, University of Trento, 2003.

[Lekavý and Návrat, 2007] Marián Lekavý and Pavol Návrat. Expressivity of STRIPS-like and HTN-like planning. In *Proceedings of Agent and Multi-Agent Systems: Technologies and Applications, First KES International Symposium (KES-AMSTA)*, pages 121–130, Wroclaw, Poland, May 31–June 1 2007.

[Lespérance *et al.*, 2000] Y. Lespérance, H. Levesque, F. Lin, and R. Scherl. Ability and knowing how in the situation calculus. *Studia Logica*, 66(1):165–186, October 2000.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[Levesque *et al.*, 1998] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linkoping Electronic Articles in Computer and Information Science*, 3(018), 1998.

[Lin, 1999] Fangzhen Lin. *Logical Foundations for Cognitive Agents: Contributions in Honor of Ray Reiter*, chapter Search algorithms in the situation calculus, pages 213–233. Springer, Berlin, 1999.

[Liu, 2002] Yongmei Liu. A hoare-style proof system for robot programs. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, pages 74–79, 2002.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[McDermott, 1998] Drew V. McDermott. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[McDermott, 1999] Drew V. McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

[McIlraith and Son, 2002] S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR)*, pages 482–493, Toulouse, France, April 22-25 2002.

[McIlraith *et al.*, 2000] S. McIlraith, G. Biswas, D. Clancy, and V. Gupta. Hybrid systems diagnosis. In *Proceedings of Hybrid Systems: Computation and Control*, pages 282–295, 2000.

[McIlraith, 1997] Sheila A. McIlraith. Explanatory diagnosis: Conjecturing actions to explain obsevations. In *Proceedings of the Eighth International Workshop on Principles of Diagnosis (DX'97)*, pages 69–78, 1997.

[McIlraith, 2000] S. McIlraith. Diagnosing hybrid systems: A bayesian model selection approach. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis (DX'00)*, pages 140–146, June 2000.

[McNeill *et al.*, 2003] Fiona McNeill, Alan Bundy, and Marco Schorlemmer. Dynamic ontology refinement. In *Proceedings of ICAPS'03 Workshop on Plan Execution*, Trento, Italy, June 2003.

[Musliner *et al.*, 1991] D. Musliner, E. Durfee, and K. Shin. Execution monitoring and recovery planning with time. In *Proceedings of the IEEE Seventh Conference on Artificial Intelligence Applications*, pages 385–388, 1991.

[Myers, 1998] K. L. Myers. Towards a framework for continuous planning and execution. In *Proceedings of the AAAI Fall Symposium on Distributed Continual Planning*, 1998.

[Nau *et al.*, 2003] Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[Nebel and Koehler, 1995] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence (Special Issue on Planning and Scheduling)*, 76(1-2):427–454, 1995.

[Nebel *et al.*, 1997] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24-26*, pages 338–350, 1997.

[Owre *et al.*, 1992] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga Springs, NY, 1992.

[Palacios and Geffner, 2006] Héctor Palacios and Hector Geffner. Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, 2006.

[Palacios and Geffner, 2007] Héctor Palacios and Hector Geffner. From conformant into classical planning: Efficient translations that may be complete too. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, pages 264–271, 2007.

[Pasula *et al.*, 2004] Hanna Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning probabilistic relational planning rules. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 73–82, Whistler, British Columbia, Canada, June 3–7 2004.

[Pednault, 1989] Edwin P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Pohl, 1970] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236, 1970.

[Pollack *et al.*, 2003] Martha E. Pollack, Laura E. Brown, Dirk Colbry, Colleen E. McCarthy, Cheryl Orosz, Bart Peintner, Sailesh Ramakrishnan, and Ioannis Tsamardinos. Autominder: an intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*, 44(3-4):273–282, 2003.

[Puterman, 1994] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, New York, 1994.

[Refanidis and Vlahavas, 2001] I. Refanidis and I. Vlahavas. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, 15:115–161, 2001.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Reiter, 1991] Ray Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Reiter, 1993] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

[Reiter, 2001] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, 2001.

[Rintanen, 2008] Jussi Rintanen. Regression for classical and nondeterministic planning. In *Proceedings of the 18th European Conference on Artificial Intelligence*, pages 568–571, 2008.

[Russell and Norvig, 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1-3):361–395, 1991.

[Sanner and Boutilier, 2005] Scott Sanner and Craig Boutilier. Approximate linear programming for first-order MDPs. In *Proc. UAI05*, pages 509–517, 2005.

[Sapena and Onaindia, 2002] Oscar Sapena and Eva Onaindia. Execution, monitoring and replanning in dynamic environments. In *AIPS-02 Workshop on On-line Planning and Scheduling*, 2002.

[Schoppers, 1987] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In John McDermott, editor, *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy, 1987. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Shapiro *et al.*, 2002] Steven Shapiro, Yves Lespérance, and Hector J. Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 19–26, Bologna, Italy, 2002.

[Son and Baral, 2001] Tran Cao Son and Chitta Baral. Formalizing sensing actions a transition function based approach. *Artif. Intell.*, 125(1-2):19–91, 2001.

[Son and Tu, 2006] Tran Cao Son and Phan Huy Tu. On the completeness of approximation based reasoning and planning in action theories with incomplete information. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 481–491, 2006.

[Soutchanski, 2003a] Mikhail Soutchanski. High-level robot programming and program execution. In *Proceedings of the Workshop on Plan Execution, 10th June 2003, held at ICAPS'03, Trento, Italy*, 2003.

[Soutchanski, 2003b] Mikhail Soutchanski. *High-Level Robot Programming in Dynamic and Incompletely Known Environments*. PhD thesis, University of Toronto, 2003.

[Thielscher, 1998] Michael Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2:179–192, 1998.

[Thielscher, 2005] Michael Thielscher. Flux: A logic programming method for reasoning agents. *TPLP*, 5(4-5):533–565, 2005.

[van der Krogt and de Weerdt, 2005] R.P.J. van der Krogt and M.M. de Weerdt. Plan repair as an extension of planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 161–170, Monterey, California, USA, June 5–10 2005.

[van Ditmarsch *et al.*, 2007] Hans P. van Ditmarsch, Andreas Herzig, and Tiago De Lima. Optimal regression for reasoning about knowledge and actions. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1070–1075, 2007.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental AI*, 7(1), 1995.

[Veloso *et al.*, 1998] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-based monitoring for continuous planning in dynamic environments. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, pages 171–179, Pittsburgh, PA, USA, June 1998.

[Verma *et al.*, 2002] V. Verma, J. Fernandez, and R. Simmons. Probabilistic models for monitoring and fault diagnosis. In Raja Chatila, editor, *The Second IARP and IEEE/RAS Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, October 2002.

[Waldinger, 1977] R. Waldinger. Achieving several goals simultaneously. In E. W. Elcock and D. Michie, editors, *Machine Intelligence*, volume 8, pages 94–136. Wiley, 1977.

[Wang, 1994] Xuemei Wang. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS'94)*, pages 335–340, University of Chicago, Chicago, Illinois, June 13–15 1994.

[Wilkins, 1985] David E Wilkins. Recovering from execution errors in SIPE. *Computational Intelligence*, 1:33–45, 1985.

[Wilkins, 1988] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1988.

[Witteveen *et al.*, 2005] C. Witteveen, N. Roos, R.P.J. van der Krogt, and M.M. de Weerdt. Diagnosis of single and multi-agent plans. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS-05)*, pages 805–812, Utrecht, The Netherlands, July 25–29 2005.

# Appendix A

# Description of Domains Used in Experiments

The example domains we used to evaluate our algorithms were taken from the set of domains used in the International Planning Competition, and the problems we tested on were variations of problems used in previous competitions. Both, the domain and problem definitions were manually converted to situation calculus representations, taken as input by our planners.

As explained in the text, we required the used heuristic functions to be expressed in terms of regressable formulae. We therefore opted to hand-code simple domain specific heuristics.

## A.1 TPP

In the TPP domain an agent travels to various markets to purchase goods and brings them back to a depot. The goal is to fill the number of requested goods. At different markets, goods are sold at different prices and are available in different quantities.

The available actions are: "drive(destination)", for driving to a given destination (a market, or the depot); "buyallneeded(good, market)", for purchasing the remaining

requested quantity of a good at the said market (which needs to be the market of the current location); “buyall( good, market)”, for buying the complete quantity of a good that is available at a market; and, as explained, “finish”.

The actions have their intuitive preconditions and effects, and purchasing goods incurs the cost of quantity times price. In addition, driving between different locations incurs different predefined costs. An optimal plan is one that fills the requests while minimizing costs.

As heuristic, we simply used the costs of driving from the current location back to the depot.

The perturbations to the initial state affected the drive costs, the prices or available quantities of goods on the various markets, or the number of requested goods. Again, these perturbations where done systematically, by multiplying these values by a factor between 0.5 and 1.5.

## A.2 Open Stacks

In the open stacks domain, the task is to fill orders consisting of a list of requested products, which need to be produced. After producing a product, an arbitrary number of instances of that product is available for inclusion in orders, but there cannot be more than a certain number of open orders at a time, each of which occupies one of a limited number of available stacks. Furthermore, the production of a product cannot start before all orders requiring that product have been started.

Orders can be started, and shipped when ready. Likewise the production of a product can be started and ended. The time required for the production of various products varies. An optimal plan is one that ships all requested orders, while minimizing the overall time.

The two heuristics used in this domain, denoted ’A’ and ’B’ in the text, were the zero-heuristic (’A’) and the maximum of the make-times of all remaining products (’B’).

Our perturbations in this domain regarded the production time of one or more products (simultaneously), the status of orders (not-started/started) and productions (not-making/making/made), and the composition of orders (which products to include). The latter is particularly interesting, as it emulates a certain degree of goal-changing discrepancies. Such perturbations are also easy to imagine in practice, say at an Internet retailer's order fulfillment center, where clients may request changes to existing orders while these are already being processed.

## A.3 Zenotravel

The Zenotravel domain is a simple logistics domain, where there are various cities, persons, and planes, and the task is to use the plans to fly the persons to their desired destination cities.

The available actions are the boarding and debarking of a person to/from a plan, flying a plane to a given city, "zooming" a plane to a given city, which means flying with higher fuel consumption, and refueling a plane (and again "finish").

The distances between cities vary and the required fuel, naturally, increases with distance. Further, all actions incur a fixed time cost. An optimal plan is one that takes all persons to their respective destination cities, while minimizing a linear combination of time cost and fuel cost.

The two, hand-coded, heuristics we used were defined as the fuel cost factor times the sum of the distances of a sub-set of all persons to their respective target locations. This number was multiplied by a factor of 1.0 for the less-informed heuristic, and by the slow-flying fuel burning rate for the most informed, more accurate heuristic. The considered persons, their origins, and their destinations were chosen as to ensure that these heuristics would be monotonic.

Our perturbations in this domain affected the distances between cities, the fuel level

and fuel capacity of planes, the rate at which fuel is burned, the time cost constant and the fuel cost factor.

# Appendix B

# Proof of Theorem 5

**Definition 9.** Given a situation $S$, a goal $G$, a cost formula $Cost(a, c)$, an admissible heuristic $Heur(h)$, and a list $L$ of tuples $(g, h, \vec{\alpha})$, we call $L$ *a consistent open list w.r.t.* $S, G,$ *Cost, and Heur* if w.r.t. $S$

(i) $L$ contains no infeasible action sequences;

(ii) any feasible action sequence is either itself contained in $L$, or a prefix or extension of it is, however, for any feasible sequence that satisfies $G$, either itself or a prefix is contained (but not an extension);

(iii) every element $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n]) \in L$ is such that $h$ is indeed the heuristic value for $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ according to $S$, i.e. $\mathcal{D} \models Heur(h, do([\alpha_1, \alpha_2, \ldots, \alpha_n], S))$; and

(iv) the accumulated costs $g$ are such that

$$\mathcal{D} \models (\exists c_1, \ldots, c_n).Cost(\alpha_1, c_1, S) \wedge Cost(\alpha_2, c_2, do(\alpha_1, S)) \wedge$$
$$\cdots \wedge Cost(\alpha_n, c_n, do([\alpha_1, \alpha_2, \ldots, \alpha_{n-1}], S)) \wedge g = c_1 + \cdots + c_n.$$

**Proposition 3.** Any open list output by $A^*$, and hence `RegBasA`$*$, is consistent w.r.t. to the given initial situation, goal, cost-, and heuristic function.

**Lemma 1.** In the sequence of invocations of Theorem 5, $O_2$ is a consistent open list w.r.t. $S_2, G, Cost$, and $Heur$.

*Proof:* We prove each item of Definition 9 in turn.

*Proof of (i):* Let $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n]) \in O_2$ and assume to the contrary that $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ is infeasible in $S_2$, i.e. there is an $1 \leq i \leq n$ such that $\mathcal{D} \not\models Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_2))$. It must be the case that $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ was either element of $O_1$, or it was introduced by `Recover`. We lead both cases to a contradiction.

In the former case the preconditions of $\alpha_i$ have different truth values in $S_1$ and $S_2$, since they must have been true in $S_1$, or else the sequence wouldn't have been included in $O_1$. Hence, there must be at least one fluent $F$ mentioned in $Poss(\alpha_i, s)$ for which we have that $\mathcal{D} \models F(S_1) \not\equiv F(S_2)$. Then, however, by definition of `Recover` (Lines 2, 3), $([\alpha_1, \ldots, \alpha_i], P)$ was included in $\Delta'$. Since $\mathcal{D} \models Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_1))$ at that point, following the definition of `RegBasA`$*$ (Lines 11, 12, 13), $T([\alpha_1, \ldots, \alpha_i]).p = true$ and $\mathcal{D} \not\models Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_2))$ (due to $T([\alpha_1, \ldots, \alpha_i]).P(s) = \mathcal{R}[Poss(\alpha_i), [\alpha_1, \ldots, \alpha_{i-1}]]$ (definition of `RegBasA`$*$), and the Regression Theorem [Reiter, 2001]), and the fact that $[\alpha_1, \ldots, \alpha_{i-1}]$ is a prefix of $[\alpha_1, \ldots, \alpha_n]$, $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n])$ would be removed from the open list, concluding the contradiction for this case.

In the latter case, i.e. `Recover` introduced this element, we get a contradiction just as easily. The only place where recover inserts new elements into $O$ is in Line 19, i.e. in the body of an **elseif** statement with condition $T(\vec{\alpha}).p = false \wedge \mathcal{D} \models T(\vec{\alpha}).P(S_2)$. This condition is violated by the assumption that $\mathcal{D} \not\models Poss(\alpha_i, do([\alpha_1, \ldots, \alpha_{i-1}], S_2))$ (again, due to the Regression Theorem and the definition of `RegBasA`$*$ stating that $T([\alpha_1, \ldots, \alpha_i]).P(s) = \mathcal{R}[Poss(\alpha_i), [\alpha_1, \ldots, \alpha_{i-1}])$. Hence, Line 19 is never reached for this sequence, and thus `Recover` cannot have inserted this element. Hence, no infeasible action sequence is contained in $O_2$.

*Proof of (ii):* Assume again to the contrary that there is an action sequence

$[\alpha_1, \alpha_2, \ldots, \alpha_n]$ which is feasible in $S_2$, but neither itself, nor a prefix or extension of it is included in $O_2$. This sequence (like any other) is either feasible in $S_1$, or infeasible in $S_1$. We lead both cases to a contradiction.

By Proposition 3, $O_1$ is consistent. Hence, `Recover` must have removed an appropriate sequence. However, as already seen above, `Recover` only removes sequences that have a prefix whose last action is not feasible in the corresponding situation in $S_2$, and hence the entire sequence $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ wouldn't be feasible in $S_2$, a contradiction.

Otherwise, if $[\alpha_1, \alpha_2, .., \alpha_n]$ is not feasible in $S_1$, there is a minimal $1 \leq i \leq n$ such that $\mathcal{D} \not\models \mathit{Poss}(\alpha_i, \mathit{do}([\alpha_1, .., \alpha_{i-1}], S_1))$, and we have $T([\alpha_1, .., \alpha_i]).p = \mathit{false}$, by Line 20 of `RegBasA`$*$. However, by assumption, $\mathcal{D} \models \mathit{Poss}(\alpha_i, \mathit{do}([\alpha_1, .., \alpha_{i-1}], S_2))$ and thus also $\mathcal{D} \models T([\alpha_1, .., \alpha_i]).P(S_2)$, by definition of `RegBasA`$*$ and the Regression Theorem. Hence, the condition on Line 11 is satisfied for the sequence $[\alpha_1, .., \alpha_i]$, as there must be a fluent mentioned in $\mathcal{R}[\mathit{Poss}(\alpha_i), [\alpha_1, .., \alpha_{i-1}]]$ with opposite truth values in $S_1$ and $S_2$, so that this sequence is included in $\Delta'$. Following the condition on Line 11 the action sequence $[\alpha_1, .., \alpha_i]$ is added to the open list, concluding the second contradiction.

Now let's turn to the second part of (ii). Assume there was an element $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n]) \in O_2$ such that there exists a minimal index $1 \leq i < n$ with $\mathcal{D} \models G(\mathit{do}([\alpha_1, \ldots, \alpha_i], S_2))$. Since we assume that the goal can only be achieved through the action *finish*, and when this action can execute it will always produce the goal, we know that $\alpha_i = \mathit{finish}$. Then however, the element $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n])$ cannot be in $O_1$ since otherwise also $\mathcal{D} \models G(\mathit{do}([\alpha_1, \ldots, \alpha_i], S_1))$, because `RegBasA`$*$ does not introduce infeasible action sequences into the open list, and since this sequence would satisfy the goal, it would not have been expanded further (cf. Proposition 3). Also, the sequence cannot have been introduced by `Recover`, since `Recover` only introduces sequences for whose last action the preconditions differ between $S_1$ and $S_2$ (but $i < n$). This concludes the contradiction for this part.

*Proof of (iii):* There are two possible cases: (a) Either $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n])$ was

added by `Recover`, or (b) it was added by the first `RegBasA`$*$. Case (a) leads easily to a contradiction, since due to Lines 17, 19 of `Recover` only elements with correct values according to $S_2$ are added to the open list. Also, this element cannot be changed in further iterations of `Recover` as it cannot be member of $\Delta'$. In case (b) we have that $\mathcal{D} \models \mathit{Heur}(h, \mathit{do}([\alpha_1, \alpha_2, \ldots, \alpha_n], S_1)) \not\equiv \mathit{Heur}(h, \mathit{do}([\alpha_1, \alpha_2, \ldots, \alpha_n], S_2))$ and hence there must be a fluent mentioned in $\mathcal{R}[\mathit{Heur}(h), [\alpha_1, \alpha_2, \ldots, \alpha_n]]$ on whose truth value $S_1$ and $S_2$ disagree. Therefore, the sequence $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ is included in $\Delta'$ in `Recover` and Lines 28–29 executed for it. Since this element is part of the open list, the **elseif**-condition holds and its new value according to $S_2$ is determined. This value is written back into the entry of the open list 29 (and also into the tree annotation 29). This concludes the contradiction of this case, that the $h$ value for any element of the open list $O_2$ is wrong w.r.t. $S_2$.

*Proof of (iv):* There are again the two possible cases: (a) $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n])$ was added by `Recover`, or (b) it was added by the first `RegBasA`$*$.

In case (a) the new value for $g$ is computed by *getGval* on Line 18 of `Recover`. This value is either accurate according to $S_2$, namely when all annotated costs for actions in the sequence are already with respect to $S_2$, in which case the contradiction is immediate, or some of them are still with respect to $S_1$ and are going to be fixed subsequently. In the later case, there must be actions in the sequence for which the costs according to $S_1$ and $S_2$ disagree. In each of these cases there are disagreeing fluents mentioned in the corresponding regressed cost formulae that trigger the treatment in Line 20, which will adjust $g$ accordingly (also cf. case (b)). Hence, this cases contradicts the assumption that the value is incorrect with respect to $S_2$.

In case (b) there again must be a fluent $F$ mentioned in $\mathcal{R}[\mathit{Cost}(\alpha_i, c), [\alpha_1, \ldots, \alpha_{i-1}]]$ for at least one $1 \leq i \leq n$ such that $\mathcal{D} \models F(S_1) \not\equiv F(S_2)$, or else the accumulated cost values for $S_1$ and $S_2$ would be the same. Hence, Lines 20–26 are executed for all such $i$'s. In each case, the correct costs for $\alpha_i$ according to $S_2$ are computed and the offset from the

previous value (w.r.t. $S_2$) is added to any element of the open list which has $[\alpha_1, \ldots, \alpha_i]$ as a prefix (a sequence has trivially itself as a prefix), and in particular $[\alpha_1, \alpha_2, \ldots, \alpha_n]$. Hence, after the **foreach** loop terminates, there are no more nodes along the branch to $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ which show a *Cost* value that is not according to $S_2$, and value $g$ reflects their sum. This completes the last contradiction. □

*Proof of Theorem 5*: Let us first consider the case where there is a plan for $S_3$, i.e. the two open lists $O_3$ and $O'$ aren't empty. Let the first element in $O'$ be $(g', 0, \vec{\alpha}')$, i.e. $\vec{\alpha}'$ is an optimal plan for $G$, for the initial state $S_2$ and $g'$ is its overall cost. The heuristic value for this element is, of course, 0, since the action sequence is a plan.

We need to show that the first element of $O_3$, let's call it $(g_3, 0, \vec{\alpha}_3)$, is exactly the same. Since we assume that any open list output by `RegBasA*` or `Recover` is sorted according to *Value*, it suffices to show that there is a member in $O_3$ whose action sequence is $\vec{\alpha}'$, no other element $(g'_3, h'_3, \vec{\alpha}'_3) \in O_3$ is such that $g'_3 + h'_3 < g_3$ (we assume that tie breaking is done the same way every time an open list is sorted, and omit these details here), and that $g_3 = g'$. Again, since it is output by `RegBasA*`, the heuristic value can only be zero.

All this follows from Lemma 1, and the completeness and optimality of $A^*$ and hence `RegBasA*`, based on the admissibility of the heuristic function.

Now to the case where $O'$ is empty, i.e. no feasible action sequence to reach the goal. We show that also $O_3$ is empty. Assume to the contrary that there is an element $(g, h, [\alpha_1, \alpha_2, \ldots, \alpha_n]) \in O_3$. This sequence satisfies the goal (ignoring preconditions), or else `RegBasA*` would not have returned it. But then $O_2$ must have already contained an element whose action sequence was a prefix $[\alpha_1, \alpha_2, \ldots, \alpha_i]$ of $[\alpha_1, \alpha_2, \ldots, \alpha_n]$, $i \leq n$, since `RegBasA*` itself is assumed correct and never introduces infeasible action sequences into the open list. The contradiction now follows again from Lemma 1 (Case (i)). □

# Appendix C

# Definitions and Proofs of Chapter 7

## C.1 Definition of comp

We here provide the pseudo-code for the **comp** function of Step 1 of our compilation. It takes four inputs: a program $\delta$, an integer `i` used as a program counter, a set `e` of program variables (introduced using the $\pi$-construct, see below), and the name of the procedure this program belongs to, `c`. It outputs a set of first-order sentences and a new integer. We will further process the sentences in the subsequent steps of the compilation, eventually producing the axioms of the new theory. The integer represents the value of the program counter at the end of the program. The definition of **comp** is given on pages 198 and 199. The following glossary is to provide some intuition about the used bookkeeping fluents and actions. For parsimony we omit situation arguments:

**Fluents:**

*Thread*$(th)$*:* Thread $th$ exists/is active.

*state*$(th) = y$*:* Thread $th$ is in state $y$, where $y = (\mathtt{i}, \mathtt{c})$ for some integer `i` (program counter), and some procedure name `c` ("context").

*stack*$(th, p) = y$*:* A stack for storing procedure call return addresses: $p$ is the stack posi-

tions, and $y$ the content.

*sp*$(th) = y$*:* The stack pointer for thread *th*, pointing to the top of the stack.

*map*$(th, p, v) = y$*:* The program variable $v$ has value $y$ in thread *th* on stack position $p$. Note that thread names and stack positions are required, since the same program variable name ($v$) may be used simultaneous in different threads, and in recursive procedures.

*childp*$(th) = y$*:* For modeling concurrency: the number of children thread *th* has.

*Parent*$(th_1, th_2)$*:* Thread $th_1$ is an ancestor of thread $th_2$.

*Forced*$(th)$*:* Thread *th* and its descendants, have exclusive execution rights.

*Prio*$(th_1, th_2)$*:* Thread $th_1$ has priority over thread $th_2$. Note that it may still temporarily be the case that *Forced*$(th_2)$ is true, in which case $th_2$ is still forced to execute before $th_1$.

*final*$(th)$*:* Thread *th* has executed completely.

*blockeds*$(th, p, \mathtt{i}, \mathtt{c})$*:* Thread *th* on stack position $p$ may not move to state $(\mathtt{i}, \mathtt{c})$, for instance, because that branch has been tried before and a backtrack action (see below) has been executed.

*blocked*$(th, p, \mathtt{i}, \mathtt{c}, x)$*:* Similarly, a $\pi$ construct that transitions into $(\mathtt{i}, \mathtt{c})$ in thread *th* on stack position $p$ may not chose value $x$.

*backtp* $= y$*:* A pointer to positions in a backtracking stack. The stack itself contains so-called "shadowed" versions of all relevant bookkeeping fluents $X$, named $s\_X$, see below.

**Actions:**

*test*$(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$*:* Under a certain condition, compiled into the preconditions of this action, thread *th* may move from state $\mathtt{i}$ to $\mathtt{i}'$, in context $\mathtt{c}$.

*rtest*$(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$*:* The same as above, but corresponding to an actual transition in the program. Used for the $\phi$? constructs only.

*noop*$(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$*:* Just like *test*, but unconditional.

*rnoop*$(th, \mathtt{i}, \mathtt{c}, \mathtt{i}')$*:* An unconditional transition, just like the previous, but only for cases where no backtracking information needs to be recorded.

*spawn*$(th, \mathtt{c}, \mathtt{i}', \mathtt{i}_1, \mathtt{i}_2)$*:* Creates two new threads (tokens in the Petri net) and sets the first thread to state $\mathtt{i}_1$, the second to $\mathtt{i}_2$, and the current thread to $\mathtt{i}'$.

*join*$(th, \mathtt{c}, \mathtt{i})$*:* Joins the child threads back into their parent.

$\pi(th, v, x, \mathtt{c}, \mathtt{i})$*:* Chose object $x$ for program variable $v$.

*call*$(th, p, \mathtt{c}, \mathtt{i})$*:* Call procedure $p$.

*return*$(th)$*:* Return from a procedure call: look up return address on the stack, and move to the stated program position.

*backtrack*$(th)$*:* Backtrack to the last backtracking point.

*finalize*$(th)$*:* Mark thread *th* as final.

In the algorithm we use the auxiliary functions **test**, **rtest**, **noop**, **rnoop** to create additional transitions in the generated Petri net, which may be conditional (*test*) or unconditional (*noop*). In these algorithms $\phi(s)|_{\mathtt{V}}$ denotes the formula resulting from substituting each occurrence of $v$ by $x_v$ for every pair $(v, x_v) \in \mathtt{V}$.

The following condition, **bindProc**, is required for handling program variables in the positions of procedure calls. The variables $\mathtt{t}_\mathtt{i}$ serve as actual parameters and $\mathtt{x}_\mathtt{i}$ as formal

**Function** $\mathtt{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ – Part 1 of 2

**Output**: a tuple $(\mathtt{ax}, \mathtt{i}')$ with $\mathtt{ax}$ a set of sentences, $\mathtt{i}'$ an integer

**switch** $\delta$ **do**

  **case** *nil*

    **return** $(\emptyset, \mathtt{i})$

  **case** $A(t_1, \ldots, t_n)$ (*where* $A(x_1, \ldots, x_n)$ *is an action*)

    $\mathtt{ax} = \{Poss(A(th, x_1, \ldots, x_n), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c}) \wedge$
    $\bigwedge_{\mathtt{j} \text{ s.t. } t_\mathtt{j} \notin \mathtt{e}} x_\mathtt{j} = t_\mathtt{j} \wedge \bigwedge_{\mathtt{j} \text{ s.t. } t_\mathtt{j} \in \mathtt{e}} map(th, sp(th, s), t_\mathtt{j}, s) = x_\mathtt{i},$
    $state(th, do(A(th, \vec{x}), s)) = (\mathtt{i}{+}1, \mathtt{c}) \leftarrow state(th) = (\mathtt{i}, \mathtt{c})\}$;

    **return** $(\mathtt{ax}, \mathtt{i}{+}1)$

  **case** $(\phi?)$

    **return** $(\mathbf{rtest}(\phi, \mathtt{i}, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c}), \mathtt{i}{+}1)$

  **case** $(\delta_1; \delta_2)$

    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}, \mathtt{e}, \mathtt{c})$;

    $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1, \mathtt{e}, \mathtt{c})$;

    **return** $(\mathtt{ax}_1 \cup \mathtt{ax}_2, \mathtt{i}_2)$

  **case** $(\delta_1 | \delta_2)$

    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;

    $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;

    $\mathtt{ax} = \{\ \mathbf{noop}(\mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), \mathbf{noop}(\mathtt{i}, \mathtt{i}_1{+}1, \mathtt{c}), \mathbf{noop}(\mathtt{i}_1, \mathtt{i}_2{+}1, \mathtt{c}), \mathbf{noop}(\mathtt{i}_2, \mathtt{i}_2{+}1, \mathtt{c})\}$;

    **return** $(\mathtt{ax} \cup \mathtt{ax}_1 \cup \mathtt{ax}_2, \mathtt{i}_2{+}1)$

  **case** $(\textbf{\textit{if}}\ \phi\ \textbf{\textit{then}}\ \delta_1\ \textbf{\textit{else}}\ \delta_2)$

    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;

    $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;

    $\mathtt{ax} = \{\mathbf{test}(\phi, \mathtt{i}, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c}), \mathbf{test}(\neg\phi, \mathtt{i}, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c}), \mathbf{noop}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{c})\}$;

    **return** $(\mathtt{ax}_1 \cup \mathtt{ax}_2 \cup \mathtt{ax}, \mathtt{i}_2)$

  **case** $(\textbf{\textit{while}}\ \phi\ \textbf{\textit{do}}\ \delta')$

    $(\mathtt{ax}, \mathtt{i}_1) = \mathbf{comp}(\delta', \mathtt{i}{+}2, \mathtt{e}, \mathtt{c})$;

    **return** $(\{\mathbf{test}(\neg\phi \vee blockeds(th, sp(th)), \mathtt{i}{+}2, \mathtt{c}), \mathtt{i}{+}1, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c}),$
    $\mathbf{rnoop}(\mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), \mathbf{test}(\phi, \mathtt{i}{+}1, \mathtt{i}{+}2, \mathtt{e}, \mathtt{c}), \mathbf{rnoop}(\mathtt{i}_1, \mathtt{i}{+}1, \mathtt{c})\} \cup \mathtt{ax}, \mathtt{i}_1{+}1)$

  **case** $(\delta'^*)$

    $(\mathtt{ax}, \mathtt{i}_1) = \mathbf{comp}(\delta', \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;

    **return** $(\mathtt{ax} \cup \{\mathbf{noop}(\mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), \mathbf{noop}(\mathtt{i}{+}1, \mathtt{i}_1{+}1, \mathtt{c}), \mathbf{noop}(\mathtt{i}_1, \mathtt{i}{+}1, \mathtt{c})\}, \mathtt{i}_1{+}1)$

  **case** $(\pi(v, \delta))$

    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta, \mathtt{i}{+}1, \mathtt{e} \cup \{v\}, \mathtt{c})$;

    $\mathtt{ax} = \{Poss(pi(th, v, x, \mathtt{c}, \mathtt{i}{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})\}$;

    **return** $(\mathtt{ax} \cup \mathtt{ax}_1, \mathtt{i}_1)$

  **case** $P(t_1, \ldots, t_n)$ (*where* $P(x_1, \ldots, x_n)$ *is a procedure*)

    $\mathtt{ax} = \{Poss(call(th, P, \mathtt{i}{+}1, \mathtt{c}), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})\}$
    $\cup\ \mathbf{bindProc}(\mathtt{e}, [t_1, \ldots, t_n], [x_1, \ldots, x_n], call(th, P, \mathtt{i}{+}1, \mathtt{c}))$;

    **return** $(\mathtt{ax}, \mathtt{i}{+}1)$

  **otherwise** see Part 2 on next page

**Function** $\mathtt{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ – Part 2 of 2

**switch** $\delta$ **do**
  **case** $(\delta_1 \parallel \delta_2)$
    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
    $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;
    $\mathtt{ax} = \{Poss(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s) \leftarrow$
      $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
      $Poss(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_2{+}1, \mathtt{c}) \wedge$
        $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
      $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
      $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
        $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
    **return** $(\mathtt{ax}_1 \cup \mathtt{ax}_2 \cup \mathtt{ax}, \mathtt{i}_2{+}2)$
  **case** $(\delta^{\parallel})$
    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
    $\mathtt{ax} = \{\mathbf{noop}(\mathtt{i}, \mathtt{i}_1{+}1, \mathtt{c})$,
      $Poss(spawn(th, \mathtt{c}, \mathtt{i}_1{+}1, \mathtt{i}, \mathtt{i}{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
      $Poss(join(th, \mathtt{c}, \mathtt{i}_1{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_1{+}1, \mathtt{c}) \wedge$
        $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
      $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
      $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
        $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
    **return** $(\mathtt{ax}_1 \cup \mathtt{ax}, \mathtt{i}_1{+}2)$
  **case** $(\delta_1 \rangle\!\rangle \delta_2)$
    $(\mathtt{ax}_1, \mathtt{i}_1) = \mathbf{comp}(\delta_1, \mathtt{i}{+}1, \mathtt{e}, \mathtt{c})$;
    $(\mathtt{ax}_2, \mathtt{i}_2) = \mathbf{comp}(\delta_2, \mathtt{i}_1{+}1, \mathtt{e}, \mathtt{c})$;
    $\mathtt{ax} = \{Poss(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s) \leftarrow$
      $Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$,
      $Poss(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s) \leftarrow Thread(th, s) \wedge state(th, s) = (\mathtt{i}_2{+}1, \mathtt{c}) \wedge$
        $final([childp(th, s){-}1|th], s) \wedge final([childp(th, s){-}2|th], s)$,
      $state(th, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) = (\mathtt{i}_2{+}2, \mathtt{c})$,
      $Prio(th_1, th_2, do(spawn(th, \mathtt{c}, \mathtt{i}_2{+}1, \mathtt{i}{+}1, \mathtt{i}_1{+}1), s)) \leftarrow$
        $th_1 = [childp(th, s) + 1|th] \wedge th_2 = [childp(th, s) + 2|th]$,
      $\neg Prio(th_1, th_2, do(join(th, \mathtt{c}, \mathtt{i}_2{+}1), s)) \leftarrow$
        $th_1 = [childp(th, s) + 1|th] \wedge th_2 = [childp(th, s) + 2|th]$,
      $Poss(finalize(th), s) \leftarrow Thread(th) \wedge \neg final(th) \wedge$
        $state(th, s) = (\mathtt{i}_1, \mathtt{c}) \vee state(th, s) = (\mathtt{i}_2, \mathtt{c})\}$;
    **return** $(\mathtt{ax}_1 \cup \mathtt{ax}_2 \cup \mathtt{ax}, \mathtt{i}_2{+}2)$

**Function** $\mathtt{test}(\phi, \mathtt{i}_1, \mathtt{i}_2, \mathtt{e}, \mathtt{c})$

$\mathtt{V} = \{(v, x_v) \mid v \in \mathtt{e} \wedge \phi \text{ mentions } v\}$ ; // $x_v$ *a new var.*
**return** $\{$*Forced*$(th, do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s))$,
  $\neg$*Forced*$(th', do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) \leftarrow$ *Thread*$(th', s) \wedge th' \neq th$,
  *state*$(th, do(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})$,
  *Poss*$(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow$ *Thread*$(th, s) \wedge$ *state*$(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge$
    $(\bigwedge_{(v,x_v)\in \mathtt{V}}$ *map*$(th, sp(th, s), v, s) = x_v) \wedge \phi(s)|_{\mathtt{V}} \wedge \neg$*blockeds*$(th, sp(th, s), \mathtt{i}_2, \mathtt{c})\} \cup$
  **shadow**$(test(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), \neg$*Forced*$(Th, s))$

**Function** $\mathtt{rtest}(\phi, \mathtt{i}_1, \mathtt{i}_2, \mathtt{e}, \mathtt{c})$

$\mathtt{V} = \{(v, x_v) \mid v \in \mathtt{e} \wedge \phi \text{ mentions } v\}$ ; // $x_v$ *a new var.*
**return** $\{\neg$*Forced*$(th', do(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) \leftarrow$ *Thread*$(th', s)$,
    *state*$(th, do(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})$,
    *Poss*$(rtest(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow$ *Thread*$(th, s) \wedge$ *state*$(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge$
      $(\bigwedge_{(v,x_v)\in \mathtt{V}}$ *map*$(th, sp(th, s), v, s) = x_v) \wedge \phi(s)|_{\mathtt{V}}\}$

**Function** $\mathtt{noop}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{c})$

**return** $\{$*Poss*$(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow$
      *Thread*$(th, s) \wedge$ *state*$(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge \neg$*blockeds*$(th, sp(th, s), \mathtt{i}_1, \mathtt{c})$,
    *Forced*$(th, do(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s))$,
    *state*$(th, do(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})\}$
    $\cup$ **shadow**$(noop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), true)$

**Function** $\mathtt{rnoop}(\mathtt{i}_1, \mathtt{i}_2, \mathtt{c})$

**return** $\{$*Poss*$(rnoop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s) \leftarrow$ *Thread*$(th, s) \wedge$ *state*$(th, s) = (\mathtt{i}_1, \mathtt{c})$,
    *state*$(th, do(rnoop(th, \mathtt{i}_1, \mathtt{i}_2, \mathtt{c}), s)) = (\mathtt{i}_2, \mathtt{c})\}$

parameters. Also note that we apply *call-by-value* by evaluating all actual parameters which are not program variables before passing them to the procedure.

$$\begin{aligned}&\mathbf{bindProc}(\mathtt{e}, [\mathtt{t_1}, \ldots, \mathtt{t_n}], [\mathtt{x_1}, \ldots, \mathtt{x_n}], \mathtt{a}) \stackrel{\text{def}}{=} \\ &\quad \bigcup_{\mathtt{j}\ \text{s.t.}\ \mathtt{t_j} \in \mathtt{e}} \{map(th, sp(th, s)+1, \mathtt{x_j}, do(\mathtt{a}, s)) = map(th, sp(th, s), \mathtt{t_j}, s)\} \\ &\quad \cup \bigcup_{\mathtt{j}\ \text{s.t.}\ \mathtt{t_j} \notin \mathtt{e}} \{map(th, sp(th, s)+1, \mathtt{x_j}, do(\mathtt{a}, s)) = \mathtt{t_j}(s)\}\end{aligned}$$

The following function (**shadow**) creates axioms which are required for a limited form of backtracking. This is occasionally required to realize synchronized while's and if's. Consider the program (**if** $\phi$ **then** $a$ **else** $b$) $\parallel$ $c$, and imagine that initially $\phi$ is false, but that $c$ makes it true. Then still, the sequence $[c, a]$ is not permitted: testing $\phi$ and executing the next action has to be atomic and may not be interrupted by other threads. In our compilation we realize this through the use of a particular *Forced*$(th, s)$ fluent, stating that only thread $th$ may execute next. Usually this fluent is false, but bookkeeping-actions make it true. This way, after performing the test action for $\phi$, only $a$ may execute next. However, imagine $a$ is not executable. We need to lift the force, and allow other threads to execute, but in order to implement synchronized if's correctly, we need to *backtrack* the thread to a state before the test first. In the example above, this is because after executing $c$, the else-case of the conditional must be executed. Backtracking is realized by keeping a stack of previous configurations of bookkeeping fluents, to which the system can revert to when necessary. The following axioms implement the pushing of backtracking information onto the stack. The *backtrack*$(th)$ action, see below, implements the restoration, i.e. popping from the stack.

**shadow**$(\mathtt{a}, \psi) \stackrel{\text{def}}{=} \{$

$S\_Thread(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Thread(x, s) \wedge \psi,$

$\neg S\_Thread(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Thread(x, s) \wedge \psi,$

$s\_state(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge state(x, s) = v \wedge \psi,$

$s\_stack(b, x, y, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge stack(x, y, s) = v \wedge \psi,$

$s\_sp(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge sp(x, s) = v \wedge \psi,$

$s\_map(b, t, p, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge map(t, p, x, s) = v \wedge \psi,$

$s\_childp(b, x, do(\mathtt{a}, s)) = v \leftarrow b = backtp(s) \wedge childp(x, s) = v \wedge \psi,$

$S\_Prio(b, x, y, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Prio(x, y, s) \wedge \psi,$

$\neg S\_Prio(b, x, y, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Prio(x, y, s) \wedge \psi,$

$S\_Forced(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge Forced(x, s) \wedge \psi,$

$\neg S\_Forced(b, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge \neg Forced(x, s) \wedge \psi,$

$S\_blockeds(b, x, p, y, c, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge blockeds(x, p, y, c, s) \wedge \psi,$

$S\_blocked(b, x, p, y, c, x, do(\mathtt{a}, s)) \leftarrow b = backtp(s) \wedge blocked(x, p, y, c, x, s) \wedge \psi,$

$backtp(do(\mathtt{a}, s)) = v \leftarrow v = backtp(s) + 1 \wedge \psi \quad \}$

## C.2 Program-Independent Axioms

The default dynamics of the involved bookkeeping actions, which are program independent, are described by the following axioms (cf. Step 2 of the compilation described in Section 7.3).

For procedure calls:

$$\begin{aligned}
\mathtt{ax}_{\mathrm{procs}} \stackrel{\mathrm{def}}{=} \{ \quad & sp(th, do(call(th, x_1, x_2, x_3), s)) = y \leftarrow y = sp(th, s)+1, \\
& state(th, do(call(th, P, x_1, x_2), s)) = y \leftarrow y = (0, P), \\
& stack(th, v, do(call(th, x_1, i, \mathtt{c}), s)) = y \leftarrow y = (i, \mathtt{c}) \wedge v = sp(th, s)+1, \\
& Forced(th, do(call(th, x_1, x_2, x_3), s)) = y \leftarrow true, \\
& state(th, do(return(th), s) = y \leftarrow y = stack(th, sp(th, s), s), \\
& sp(th, do(return(th), s)) = y \leftarrow y = sp(th, s)–1 \quad \}
\end{aligned}$$

For concurrency:

$\mathtt{ax}_{\mathrm{conc}} \stackrel{\mathrm{def}}{=} \{$

$Thread(th', do(spawn(th, \mathtt{c}, x_1, x_2, x_3), s)) \leftarrow$

$$th' = [childp(th,s)|th] \vee th' = [childp(th,s){+}1|th],$$

$$childp(th, do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) = y \leftarrow y = childp(th,s){+}2,$$

$$sp(th', do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) = y \leftarrow$$
$$y = 0 \wedge (th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$childp(th', do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) = y \leftarrow$$
$$y = 0 \wedge (th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$Parent(\bar{th}, th', do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) \leftarrow$$
$$(th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]) \wedge (\bar{th} = th \vee Parent(\bar{th}, th)),$$

$$state(th, do(spawn(th,\mathsf{c},i,x_1,x_2),s)) = y \leftarrow y = (i,\mathsf{c}),$$

$$state(th', do(spawn(th,\mathsf{c},x_1,i,x_2),s)) = y \leftarrow y = (i,\mathsf{c}) \wedge th' = [childp(th,s)|th],$$

$$state(th', do(spawn(th,\mathsf{c},x_1,x_2,i),s)) = y \leftarrow y = (i,\mathsf{c}) \wedge th' = [childp(th,s)+1|th],$$

$$map(th', p, v, do(spawn(th,\mathsf{c},x_1,x_2,i),s)) = x \leftarrow$$
$$map(th,p,v,do(spawn(th,\mathsf{c},x_1,x_2,i),s)) = x \wedge th' = [childp(th,s)|th],$$

$$Prio(th', x, do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) \leftarrow$$
$$Prio(th,x) \wedge (th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$Prio(x, th', do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) \leftarrow$$
$$Prio(x,th) \wedge (th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$backtp(th', do(spawn(th,\mathsf{c},x_1,x_2,x_3),s)) = 0 \leftarrow$$
$$(th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$\neg final(th', do(spawn(th,\mathsf{c},x,y,z),s)) \leftarrow$$
$$(th' = [childp(th,s)|th] \vee th' = [childp(th,s)+1|th]),$$

$$\neg Thread(th', do(join(th,x_1,x_2),s)) \leftarrow$$
$$childp(th,s) > 1 \wedge (th' = [childp(th,s){-}1|th] \vee th' = [childp(th,s){-}2|th]),$$

$$childp(th, do(join(th,x_1,x_2),s)) = y \leftarrow childp(th,s) > 1 \wedge y = childp(th,s){-}2,$$

$$Forced(th, do(join(th,x_1,x_2),s)) \leftarrow Forced([childp(th,s)|th],$$

$$final(th, do(finalize(th),s)) \leftarrow true\}$$

For program variables:

$$\begin{aligned}\mathtt{ax}_{\pi} \stackrel{\text{def}}{=} \{ \quad & state(th, do(pi(th, v, x, \mathtt{c}, i), s)) = y \leftarrow y = (i, \mathtt{c}), \\ & Forced(th, do(pi(th, v, x, \mathtt{c}, i), s)) = y \leftarrow true, \\ & map(th, p, v, do(pi(th, v, x, \mathtt{c}, i), s)) = x \leftarrow p = sp(th, s) \quad \}\end{aligned}$$

The following axioms realize the backtracking described earlier, i.e. the restoration of an earlier configuration of the bookkeeping fluents. This is only required when concurrency is used.

$\mathtt{ax}_{backtrack} \stackrel{\text{def}}{=} \{$

$\neg Thread(x, do(backtrack(th), s)) \leftarrow \neg S\_Thread(backtp\text{–}1, x, s),$

$Thread(x, do(backtrack(th), s)) \leftarrow S\_Thread(backtp\text{–}1, x, s),$

$state(x, do(backtrack(th), s)) = v \leftarrow s\_state(backtp\text{–}1, x, s) = v,$

$stack(x, y, do(backtrack(th), s)) = v \leftarrow s\_stack(backtp\text{–}1, x, y, s) = v,$

$sp(x, do(backtrack(th), s)) = v \leftarrow s\_sp(backtp\text{–}1, x, s) = v,$

$map(t, p, x, do(backtrack(th), s)) = v \leftarrow s\_map(backtp\text{–}1, t, p, x, s) = v,$

$childp(x, do(backtrack(th), s)) = v \leftarrow s\_childp(backtp\text{–}1, x) = v,$

$Prio(x, y, do(backtrack(th), s)) \leftarrow S\_Prio(backtp\text{–}1, x, y, s),$

$\neg Prio(x, y, do(backtrack(th), s)) \leftarrow \neg S\_Prio(backtp\text{–}1, x, y, s),$

$Forced(x, do(backtrack(th), s)) \leftarrow S\_Forced(backtp\text{–}1, x, s),$

$\neg Forced(x, do(backtrack(th), s)) \leftarrow \neg S\_Forced(backtp\text{–}1, x, s),$

$blockeds(th, p, i, c, do(backtrack(th), s)) \leftarrow S\_blockeds(backtp\text{–}1, th, p, i, c, s),$

$\neg blockeds(th, p, i, c, do(backtrack(th), s)) \leftarrow \neg S\_blockeds(backtp\text{–}1, th, p, i, c, s),$

$blocked(th, p, i, c, x, do(backtrack(th), s)) \leftarrow S\_blocked(backtp\text{–}1, th, p, i, c, x, s),$

$\neg blocked(th, p, i, c, x, do(backtrack(th), s)) \leftarrow \neg S\_blocked(backtp\text{–}1, th, p, i, c, x, s),$

$backtp(do(backtrack(th), s) = v \leftarrow v = backtp\text{–}1,$

$blockeds(th, p, i, c, do(noop(th, c, i), s)) \leftarrow p = sp(th, s),$

$S\_blockeds(b, th, p, i, c, do(noop(th, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$

$$blockeds(th, p, i, c, do(test(th, i', c, i), s)) \leftarrow p = sp(th, s),$$

$$S\_blockeds(b, th, p, i, c, do(test(th, i', c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$$

$$blocked(th, p, i, c, x, do(pi(th, i', x, c, i), s)) \leftarrow p = sp(th, s),$$

$$S\_blocked(b, th, p, i, c, x, do(pi(th, i', x, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$$

$$blockeds(th, p, i, c, do(join(th, c, i), s)) \leftarrow p = sp(th, s),$$

$$S\_blockeds(b, th, p, i, c, do(join(th, c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp,$$

$$blockeds(th, p, i, c, do(call(th, i', c, i), s)) \leftarrow p = sp(th, s),$$

$$S\_blockeds(b, th, p, i, c, do(call(th, i', c, i), s)) \leftarrow p = sp(th, s) \wedge b = backtp\}$$

$$\cup\, \mathbf{shadow}(do(call(th, i', c, i), s), \neg Forced(th))$$

$$\cup\, \mathbf{shadow}(do(join(th, c, i), s), \neg Forced(th))$$

$$\cup\, \mathbf{shadow}(do(\pi(th, i', x, c, i), true)$$

$$\cup\, \{Poss(backtrack(th), s) \leftarrow backtp(s) > 0 \wedge Thread(th) \wedge$$

$$Forced(th) \wedge \neg CanTrans(th) \wedge (\nexists t).Parent(th, t) \wedge Thread(t) \wedge CanTrans(t)\}$$

Intuitively, above blocking effects ensure that after backtracking not the same pseudo actions are performed, and thus creating a cycle. The unblocking effects of real actions (see below), imply that a bookkeeping action can be repeated after a real action has taken place – and thus, the state of the world may have changed. Backtracking is only possible, when a forced thread cannot execute any other action, and none of its descendant threads can either (cf. Step 4).

The following axioms describe some of the values of above used bookkeeping fluents in the initial situation $S_0$:

$$\begin{aligned} \mathtt{ax}_{S_0} \stackrel{\text{def}}{=} \{ \quad & Thread([0], S_0) \leftarrow true, \\ & state([0], S_0) = (0, \textit{'main'}) \leftarrow true, \\ & sp([0], S_0) = 0 \leftarrow true, \\ & stack([0], 0, S_0) = \textit{'final'} \leftarrow true, \\ & childp([0], S_0) = 0 \leftarrow true \quad \} \end{aligned}$$

We further have the following additional bookkeeping effects for *real actions*, where $\mathcal{A}$

denotes the set of all *domain actions*, i.e. primitive actions of the original theory $\mathcal{D}$ :

$$\begin{aligned}\mathtt{ax}_{real} \stackrel{\text{def}}{=} \{ \quad & \neg blockeds(th, p, i, c, do(\alpha, s)), \\ & \neg blocked(th, p, i, c, x, do(\alpha, s)), \\ & \neg final(th, do(\alpha, s)), \\ & \neg Forced(th, do(\alpha, s)) \quad \}_{\alpha \in \mathcal{A} \cup \{rtest\}}\end{aligned}$$

The union of these sets forms the set of common, program independent axioms:

$$\mathtt{ax}_{\text{common}} \stackrel{\text{def}}{=} \mathtt{ax}_{\text{procs}} \cup \mathtt{ax}_{\text{conc}} \cup \mathtt{ax}_{\pi} \cup \mathtt{ax}_{backtrack} \cup \mathtt{ax}_{S_0} \cup \mathtt{ax}_{real}$$

## C.3 Proof of Theorem 9

### C.3.1 Definitions

We use the following auxiliary definitions for the proofs.

- If $\mathcal{D}$ is a basic action theory, then we denote by $\mathcal{D}_{\delta,\mathtt{i},\mathtt{e},\mathtt{c},\mathtt{i}'}$ the basic action theory resulting from computing $(\mathtt{ax}, \mathtt{i}') = \mathbf{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ and performing steps 2 to 6 of the compilation on $\mathtt{ax}$. When $\mathtt{i}$ is omitted, we mean $\mathtt{i} = 0$. Similarly, if $\mathtt{e}$ is omitted, we assume $\mathtt{e} = \emptyset$.

- As mentioned earlier, in action theories resulting from a compilation, all actions take a thread name as their first argument, i.e. domain actions (defined in the original theory $\mathcal{D}$) receive an additional argument in the compiled theories. In the proof below we sometimes refer to situation terms in compiled theories containing action terms without this additional thread argument. We interpret these – in the context of a compiled theory – as macros, expanding into action terms with a dummy thread $[-1]$ as their first argument (e.g. $A(t_1, \ldots, t_n)$ becomes $A([-1], t_1, \ldots, t_n)$). Even though the implied situation term is not executable in the respective theory (since that dummy thread is never active), it serves our purposes, as the action still has its intended effects on the state of the world, i.e. on all non-bookkeeping fluents. This convention ensures that all situation terms of the original theory $\mathcal{D}$ are also situation terms in any new theory, resulting from the compilation of some program, and represent the same state of the world.

- For two situation terms $S$ and $S' = do(\vec{a}, S)$, we refer to the action sequence $\vec{a}$ by $S' - S$. We denote the concatenation of two action sequences $\vec{a}, \vec{b}$ by $\vec{a} \cdot \vec{b}$.

- We call a situation $S'$ a *continuation* of another situation $S$, if $\Sigma \models S \sqsubseteq S'$. We call a continuation *proper* if $\sigma = S' - S$ is non-empty. We say that a continuation

is *in thread th* to express that all actions in $\sigma$ are executed in thread *th*, i.e. their first argument is *th*.

- For two situations $s, s'$ with $\Sigma \models s \sqsubset s'$, we write $exec(s, s')$ to state that the part of $s'$ that extends $s$ is executable, formally:

$$exec(s, s') \stackrel{\text{def}}{=} (\forall a, s^*).s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*).$$

Similarly, $exec_{th}(s, s')$ denotes that all actions in the part of $s'$ that extends $s$ that are in thread *th* or any of its descendants, are executable. Formally:

$$exec_{th}(s, s') \stackrel{\text{def}}{=}$$
$$(\forall a, th', \vec{x}, s^*).s \sqsubset do(a(th', \vec{x}), s^*) \sqsubseteq s' \wedge Suffix(th, th') \supset Poss(a(th', \vec{x}), s^*).$$

where $Suffix(L_1, L_2)$ holds if $L_1$ and $L_2$ are lists and the former is a suffix of the latter.

- We call domain actions and *rtest* actions *real actions*, and call all other bookkeeping actions *pseudo actions*.

We define a new predicate, $\widehat{Trans}$, which is just like *Trans* but allows the execution of arbitrary actions at any point during program execution. These actions are marked, so that later it can be distinguished whether they are the result of executing program steps or whether they were injected. This is to formalize the notion of executability of a program, conditioned on the execution of other actions in concurrently executing threads.

**Definition 10.** Let $\mathcal{D}$ be any basic action theory whose set of actions is $\mathcal{A}$. Then $\widehat{\mathcal{D}}$ is just like $\mathcal{D}$, except that the set of actions is $\mathcal{A} \cup \widehat{\mathcal{A}}$ where $\widehat{\mathcal{A}} = \{\widehat{a} \mid a \in \mathcal{A}\}$, and the effects and preconditions of $\widehat{a}$ are equal to those of $a$.

We define:

$$\widehat{Trans}(\delta, s, \delta', s') \stackrel{\text{def}}{=} Trans(\delta, s, \delta', s') \vee (\delta' = \delta \wedge (\exists \widehat{a}).s' = do(\widehat{a}, s)).$$

and define $\widehat{Trans}^*$ just like $Trans^*$ but for $\widehat{Trans}$ instead of $Trans$. Further, $pure(s)$ denotes the situation obtained by replacing any marked action in $s$ by its unmarked counterpart.

In order to compare two situation terms $\widehat{S}$ in $\widehat{\mathcal{D}}$ and $S_\delta$ in a compiled theory $\mathcal{D}_\delta$, we define $S_\delta \mathrel{\widehat{=}}_{th} \widehat{S}$ as follows:

$$
\begin{aligned}
do(a(th', \vec{x}), s) \mathrel{\widehat{=}}_{th} do(a(\vec{x}), s') \quad &\text{if} \quad s \mathrel{\widehat{=}}_{th} s' \text{ and } a \in \mathcal{A} \text{ and } (th' = th \text{ or } Suffix(th, th')) \\
do(a(th', \vec{x}), s) \mathrel{\widehat{=}}_{th} do(\widehat{a}(\vec{x}), s') \quad &\text{if} \quad s \mathrel{\widehat{=}}_{th} s' \text{ and } a \in \mathcal{A} \text{ and } th' \neq th \text{ and } \neg Suffix(th, th') \\
do(a(th', \vec{x}), s) \mathrel{\widehat{=}}_{th} s' \quad &\text{if} \quad s \mathrel{\widehat{=}}_{th} s' \text{ and } a \notin \mathcal{A} \\
s \mathrel{\widehat{=}}_{th} s &
\end{aligned}
$$

That is, the actions are matched in order, where each domain action executed in $th$ or its descendants matches an action in $\mathcal{D}$, each domain action executed in any other thread matches an inserted action, and all non-domain actions are ignored.

### C.3.2 Lemmata

We use the following auxiliary lemma.

**Lemma 2.** All precondition axioms in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ are conjunctions where one of the conjuncts is either of the form $state(th, s) = (\mathtt{i}_1, \mathtt{c}_1) \vee \cdots \vee state(th, s) = (\mathtt{i}_n, \mathtt{c}_n)$, or $false$, and $\mathtt{i} \leq \mathtt{i}_j < \mathtt{i}'$ for all $1 \leq j \leq n$, and another conjunct is $Thread(th, s)$.
Further, let $s$ be any situation in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models Thread(th, s) \wedge state(th, s) = (\mathtt{i}, \mathtt{c})$. Then there is no continuation $s'$ of $s$ and an integer $\mathtt{i}'' > \mathtt{i}'$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(s, s') \wedge state(th, s') = (\mathtt{i}'', \mathtt{c})$.
*Proof:* The theorem follows by construction of the compilation. □

Intuitively, the first part of the lemma states that all action preconditions in the compiled theory explicitly enumerate all *state*'s in which the action can execute, these states are within the parameters input and output by the compilation algorithm, and

that only active threads can execute. The second part says that executable actions do not result in states whose number is beyond the final state, defined by the compilation. As a consequence of this lemma we have that if in a situation $s$ a thread $th$ is in a state $\mathtt{i}^*$ of some context $\mathtt{c}$ (i.e. $state(th, s) = (\mathtt{i}^*, \mathtt{c})$), then the only actions that might be executable in $s$ in $th$ are those listed in $\mathtt{ax}$ of $(\mathtt{ax}, \mathtt{i}') = \mathbf{comp}(\delta, \mathtt{i}, \emptyset, \mathtt{c})$ for some $\mathtt{i} \leq \mathtt{i}^*$ and some $\mathtt{i}' > \mathtt{i}^*$.

The following is going to be our main lemma. The theorem is going to follow as a special case of it. The lemma indeed states something stronger than the actual theorem, namely, intuitively, that the compiled theory and the original semantics admit the same set of future situations, irrespective of whether these situations denote complete program execution or can be extended into one. This is needed in order to make the induction work.

**Lemma 3.** Let $\delta$ be any ConGolog program, and $S$ any ground situation term in $\mathcal{D}$. Then:

"$\Rightarrow$"
1. for any thread name $th$, integer $\mathtt{i}$, and situation $S_\delta$ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $filter(S_\delta, \mathcal{D}) = S$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models Thread(th, S_\delta) \wedge state(th, S_\delta) = (\mathtt{i}, \mathtt{c})$, if there exists a continuation $S'_\delta$ of $S_\delta$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$ then there is a program $\delta'$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta, S, \delta', \widehat{S}')$; and
2. if further $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then also $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$.

"$\Leftarrow$"
1. if there is a program $\delta'$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta, S, \delta', \widehat{S}')$ then, for any thread name $th$, integer $\mathtt{i}$, and situation $S_\delta$ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $filter(S_\delta, \mathcal{D}) = S$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models Thread(th, S_\delta) \wedge state(th, S_\delta) = (\mathtt{i}, \mathtt{c})$, there exists a continuation $S'_\delta$ of $S_\delta$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$; and
2. if further $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$, then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$.

Note that as a special case of this, if $\widehat{S}'$ does not mention any marked actions, then also $S'_\delta$ does not mention any actions not in thread *th* or its descendants, and vice versa.

*Proof of Lemma 3 for programs without the $\pi$ construct:*
The proof proceeds by induction over the structure of program $\delta$. We refer to the definition of *Trans* and *Final* of [De Giacomo *et al.*, 2000].

The induction has several base cases:

$\delta = nil$**:**

$\Rightarrow$:

By definition of **comp**, we have that $\mathbf{comp}(\delta, \mathtt{i}, \emptyset, \mathtt{c}) = (\emptyset, \mathtt{i})$. Since $\mathtt{ax}$ is empty, so is the disjunction of $\phi$'s (preconditions) in Step 5 of the compilation (cf. Lemma 2). Since an empty disjunction is equivalent to *false*, no action is ever possible. Hence, the new action theory $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ has no executable situations which are proper continuations of $S_\delta$ and which mention domain actions in *th*. Since $\mathcal{D} \models (\forall s)\mathit{Final}(nil, s)$, the second implication trivially holds for $S = \mathit{filter}(S_\delta)$.

$\Leftarrow$:

By definition, $\mathcal{D} \models (\forall s\ \not\exists \delta', s')\mathit{Trans}(nil, s, \delta', s')$. Hence, $\widehat{S}'$ cannot contain any unmarked actions. Any situation $S'_\delta$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$, hence does not contain any actions in *th*. Since by construction the only domain actions that can change the state of thread *th* are those in *th* itself, we get that any such $S'_\delta$ satisfies:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta) \wedge \mathit{state}(th, S'_\delta) = (\mathtt{i}, \mathtt{c})$$

and hence the thesis for this case.

$\delta = A(t_1, \ldots, t_n)$ **where $A$ is a primitive domain action:**

$\Rightarrow$:

By construction of **comp** for this case and due to Lemma 2, the only potentially executable action in $S_\delta$ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ in thread *th* is $A(th, x_1, \ldots, x_n)$, where, by definition of **comp**, $x_i = t_i(S_\delta)$ (recall that $\mathtt{e} = \emptyset$). It is possible, only if $A$'s original preconditions are satisfied as well. By definition of **comp** this action causes $state(th, do(A(th, x_1, \ldots, x_n), S_\delta)) = (\mathtt{i}', \mathtt{c})$, and hence does not admit any further actions in thread *th*, due to Lemma 2. By definition of *Trans* it follows that $\mathcal{D} \models Trans(A(t_1, \ldots, t_n), S, nil, do(A(t_1(S), \ldots, t_n(S)), S))$, and obviously $do(A(th, x_1, \ldots, x_n), S_\delta) \widehat{=}_{th} do(A(t_1(S), \ldots, t_n(S)), S)$ from the above, and the assumption that $filter(S_\delta, \mathcal{D}) = S$.

Further, by definition, $\mathcal{D} \models Final(nil, do(A(t_1(S), \ldots, t_n(S)), S))$, hence the thesis for this case ($\widehat{S}' = do(A(t_1(S), \ldots, t_n(S)), S)$).

$\Leftarrow$:

By definition of *Trans*, $\widehat{S}' = do(A(t_1(S), \ldots, t_n(S)), S)$ is the only continuation of $S$ such that there exists a program $\delta'$ such that $\mathcal{D} \models Trans(A(t_1, \ldots, t_n), S, \delta', \widehat{S}')$, and hence the only continuation in $\widehat{\mathcal{D}}$ mentioning only unmarked actions such that $\widehat{\mathcal{D}} \models \widehat{Trans}(A(t_1, \ldots, t_n), S, \delta', \widehat{S}')$. Hence, any continuation $S'_\delta$ such that $S'_\delta \widehat{=}_{th} \widehat{S}'$ containing $A(th, t_1(S), \ldots, t_n(S))$ as its only action in *th* and not containing any pseudo-actions, satisfies the condition. It hence follows, as before:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta) \wedge state(th, S'_\delta) = (\mathtt{i}', \mathtt{c}).$$

$\delta = \phi?$**:**

$\Rightarrow$:

By definition of **comp** and due to Lemma 2, the only potentially possible action in situation $S_\delta$ in thread *th*, is $rtest(th, \mathtt{i}, \mathtt{i}', \mathtt{c})$, and after that no more actions are possible in *th*. Also note that $\mathtt{i}' = \mathtt{i}+1$. Hence, we can assume that $S'_\delta$ only contains this action in *th*. Let $\sigma, \sigma'$ be such that $S'_\delta = do(\sigma', do(rtest(th, \mathtt{i}, \mathtt{i}', \mathtt{c}), do(\sigma, S_\delta)))$,

i.e. the sequences of actions (in other threads) before and after this action. The action is, by construction, only possible if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \phi(do(\sigma, S_\delta))$ (recall, $\mathtt{e} = \emptyset$). Since $\phi$ cannot mention any of the bookkeeping fluents introduced by the compilation, we have $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \phi(do(\sigma, S_\delta))$ iff $\mathcal{D} \models \phi(\mathit{filter}(do(\sigma, S_\delta), \mathcal{D}))$. Let $\widehat{S}'$ be any continuation of $S$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$. By definition of $\mathrel{\widehat{=}}_{th}$ we have that there is a situation $\widehat{S}''$ such that $\Sigma \models S \sqsubset \widehat{S}'' \sqsubset \widehat{S}'$ and which satisfies $do(\sigma, S) \mathrel{\widehat{=}}_{th} \widehat{S}''$. It follows by definition of *Trans* that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}(\phi, \widehat{S}'', \mathit{nil}, \widehat{S}'')$, and hence $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\phi, S, \mathit{nil}, \widehat{S}')$

As before, $\mathcal{D} \models (\forall s).\mathit{Final}(\mathit{nil}, s)$ and hence the thesis.

$\Leftarrow$:

By definition of *Trans*, $\widehat{S}' = S$ is the only (improper) continuation of $S$ such that there exists a program $\delta'$ such that $\mathcal{D} \models \mathit{Trans}(\phi, S, \delta', \widehat{S}')$, and hence the only continuation in $\widehat{\mathcal{D}}$ mentioning only unmarked actions such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\phi, S, \delta', \widehat{S}')$. Hence, any situation $S'_\delta$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ containing no actions in *th* and not containing any pseudo-actions, satisfies the condition. It hence follows, as before:

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta) \wedge \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$$

The induction steps regard all other programming constructs and are as follows:

$(\delta_1; \delta_2)$**:**

$\Rightarrow$:

There are two cases to distinguish: (a) $S'_\delta$ is such that the integer $\mathtt{i}^*$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'_\delta) = (\mathtt{i}^*, \mathtt{c})$ is $\leq \mathtt{i}_1$, where $\mathtt{i}_1$ is as defined by **comp** for this case, or (b) $\mathtt{i}^* \geq \mathtt{i}_1$. In case (a) the thesis follows immediately by induction hypothesis. We hence only need to consider case (b).

By Lemma 2 and definition of **comp** there is a prefix $\sigma_1$ of $S'_\delta - S_\delta$, such that

$$\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, do(\sigma_1, S_\delta)) \wedge \mathit{state}(th, do(\sigma_1, S_\delta)) = (\mathtt{i}_1, \mathtt{c})$$

where $\mathtt{i}_1$ is the integer defined in **comp** for this case. By induction hypothesis, there hence is a continuation $\widehat{S}'_1$ of $S$ in $\widehat{\mathcal{D}}$ and a program $\delta'$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta', \widehat{S}'_1) \wedge \mathit{Final}(\delta', \widehat{S}'_1)$.

Furthermore, since $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, do(\sigma_1, S_\delta)) = (\mathtt{i}_1, \mathtt{c})$, induction hypothesis also applies to $do(\sigma_1, S_\delta)$ and $\delta_2$: Since $S'_\delta$ is a continuation of $do(\sigma_1, S_\delta)$ which, by the initial assumption, satisfies $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(do(\sigma, S_\delta), S'_\delta)$ it follows that there also exists a ground continuation $\widehat{S}'_2$ of $\mathit{filter}(do(\sigma_1, S_\delta), \mathcal{D})$ in $\widehat{\mathcal{D}}$ such that there is $\delta'$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_2, \mathit{filter}(do(\sigma_1, S_\delta), \mathcal{D}), \delta', \widehat{S}'_2)$. Hence $\widehat{S}' = do(S'_2 - \mathit{filter}(do(\sigma_1, S_\delta), \mathcal{D}), do(\widehat{S}'_1 - S, S))$ satisfies, by definition of *Trans* for sequences: $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta', \widehat{S}')$.

Further by induction hypothesis, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$ then also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$. Hence the thesis.

$\Leftarrow$:

In analogy to above, we can again distinguish the two cases: (a) there is no situation $\widehat{S}''$ in $\widehat{\mathcal{D}}$ such that $\Sigma \models S \sqsubset \widehat{S}'' \sqsubset \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \mathit{nil}; \delta_2, \widehat{S}'')$, or (b) there is. In case (a) the thesis again follows immediately by induction hypothesis. In the following we show case (b).

By definition of $\widehat{\mathit{Trans}}^*$ there is a prefix $\sigma$ of $\widehat{S}' - S$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1; \delta_2, S, \mathit{nil}; \delta_2, do(\sigma, S))$. Hence, by induction hypothesis, there is a continuation $S'_{\delta_1}$ for any $S_\delta$, where the latter is as described in the lemma, such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} do(\sigma, S)$ and

$$\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S_\delta, S'_{\delta_1}) \wedge \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$$

for $\mathtt{i}_1$ as defined by **comp**.

We can once again apply induction hypothesis on the second sub-program, using the situation $S'_{\delta_1}$, since it satisfies the constraints. The situation $\widehat{S}'$ is a continuation of $\mathit{pure}(do(\sigma, S))$ in $\widehat{\mathcal{D}}$ that satisfies $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_2, \mathit{pure}(do(\sigma, S)), \delta', \widehat{S}')$.

Hence, there exists a continuation $S'_\delta$ of $S'_{\delta_1}$ such that $S'_\delta \widehat{=}_{th} \widehat{S'}$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1,\mathtt{c},\mathtt{i}'} \models exec_{th}(S'_{\delta_1}, S'_\delta)$. It follows by definition of $exec_{th}$ that also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$.

Further by this second application of induction hypothesis, we get that if $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S'})$, then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$.

$(\delta_1 \parallel \delta_2)$**:**

$\Rightarrow$:

By definition of **comp** and Lemma 2, the first action in $\sigma = S'_\delta - S_\delta$ in thread $th$ can only be $spawn(th, \mathtt{c}, \mathtt{i}+1, \mathtt{i}_1+1)$ where $\mathtt{i}_1$ is an integer defined in the compilation. By $\mathtt{ax}_{common}$ the reached situation $S_1 = do(spawn(th, \mathtt{c}, \mathtt{i}_2+1, \mathtt{i}+1, \mathtt{i}_1+1), S_\delta)$ is such that two new threads exist which we here denote $th_1, th_2$, one in state $\mathtt{i}+1$ and another in $\mathtt{i}_1+1$ of the same context.

We consider both cases of the lemma separately.

1. By definition of $exec_{th}$ and Lemma 2 it follows that $S'_\delta$ satisfies both $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models exec_{th_1}(S_1, S'_\delta)$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models exec_{th_2}(S_1, S'_\delta)$, and also $filter(S_1, \mathcal{D}) = S$. Hence, by induction hypothesis, there are programs $\delta'_1, \delta'_2$ and respective continuations $\widehat{S}'_1$ and $\widehat{S}'_2$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_\delta \widehat{=}_{th_1} \widehat{S}'_1$, $S'_\delta \widehat{=}_{th_2} \widehat{S}'_2$, $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta'_1, \widehat{S}'_1)$, and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta'_2, \widehat{S}'_2)$.

   It is easy to show from the definition of $\widehat{=}_{th}$ that all unmarked actions of $\widehat{S}'_1$ then also appear marked in $\widehat{S}'_2$ in the same order, and vice versa, and that all other marked actions are shared. Hence, $pure(\widehat{S}'_1) = pure(\widehat{S}'_2)$. Construct $\widehat{S'}$ as follows: take $\widehat{S}'_1$ and unmark all actions that appear unmarked in $\widehat{S}'_2$. It is easy to see from the definition of $\widehat{=}_{th}$ that $S'_\delta \widehat{=}_{th} \widehat{S'}$. Similarly, from the definition of $\widehat{Trans}$ it follows that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1 \parallel \delta_2, S, \delta'_1 \parallel \delta'_2, \widehat{S'})$.

2. If further $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then by definition of **comp** the last action of $S'_\delta$ in thread $th$ can only be $join(th, \mathtt{c}, \mathtt{i}_2+2)$, which in turn

can only occur after executing *finalize*($th_1$) and *finalize*($th_2$). These actions are only possible in a situation $s$ where $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th_1, s) = (\mathtt{i}_1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models state(th_2, s) = (\mathtt{i}_2, \mathtt{c})$ respectively. Hence, by induction hypothesis also $\widehat{\mathcal{D}} \models Final(\delta_1', \widehat{S}_1')$ and $\widehat{\mathcal{D}} \models Final(\delta_1', \widehat{S}_1')$. Hence, by definition of *Trans*, $\widehat{\mathcal{D}} \models Final(\delta_1' \parallel \delta_2', \widehat{S}')$.

$\Leftarrow$:

We again consider the two parts:

1. By the definition of $\widehat{Trans}^*$ the situation $\widehat{S}'$ can be transformed into two situations $\widehat{S}_1'$ and $\widehat{S}_2'$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta_1', \widehat{S}_1')$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta_2', \widehat{S}_2')$ for some $\delta_1', \delta_2'$, by marking the actions executed in the respectively other subprogram.

   We hence get by induction hypothesis that for any situation $S_\delta^*$ which satisfies $filter(S_\delta^*, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models Thread(th_1) \wedge state(th_1, S_\delta^*) = (\mathtt{i}+1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models Thread(th_2) \wedge state(th_2, S_\delta^*) = (\mathtt{i}_1+1, \mathtt{c})$, for any $\mathtt{i}, \mathtt{i}_1$, that there exist continuations $S_{\delta_1}', S_{\delta_2}'$ of $S_\delta^*$ such that $S_{\delta_1}' \mathrel{\widehat{=}}_{th} \widehat{S}_1'$ and $S_{\delta_2}' \mathrel{\widehat{=}}_{th} \widehat{S}_2'$, and $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models exec_{th_1}(S_\delta^*, S_{\delta_1}')$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models exec_{th_2}(S_\delta^*, S_{\delta_2}')$.

   It is easy to show that $S_1 = do(spawn(th, \mathtt{c}, \mathtt{i}_2+1, \mathtt{i}+1, \mathtt{i}_1+1), S_\delta)$ satisfies the above conditions for any $S_\delta$ satisfying the conditions in the lemma. By definition of **comp**, the action $spawn(th, \mathtt{c}, \mathtt{i}_2+1, \mathtt{i}+1, \mathtt{i}_1+1)$ is executable in $S_\delta$ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, following the assumptions about this situation.

   Construct a new situation $S_\delta''$ as follows: Iterate concurrently through $S_1'$ and $S_2'$ (remember they are essentially the same, ignoring the marking): if the next unmarked action is in $S_1'$ then move all actions $S_{\delta_1}'$ up to and including this action to $S_\delta''$. Otherwise, do the same using $S_{\delta_2}'$. Since the domain actions are the same and have not changed compared to $S_{\delta_1}'$ and $S_{\delta_2}'$ it follows that $S_\delta' = do(S_\delta'' - S_\delta, do(spawn(th, \mathtt{c}, \mathtt{i}_2+1, \mathtt{i}+1, \mathtt{i}_1+1), S_\delta))$ satisfies: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models$

$exec_{th_1}(S_\delta, S'_\delta)$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th_2}(S_\delta, S'_\delta)$, and hence, by definition of $exec_{th}$ also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$. Further, by construction, $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ (note that *spawn* is not in $\mathcal{A}$).

2. If further $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, then, by definition of *Final* for concurrency, also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1, \widehat{S}'_1)$ and $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_2, \widehat{S}'_2)$. It hence follows by induction hypothesis that also $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$ and $\mathcal{D}_{\delta_2,\mathtt{i}_1+1,\mathtt{c},\mathtt{i}_2} \models \mathit{state}(th, S'_{\delta_2}) = (\mathtt{i}_2, \mathtt{c})$, with $\mathtt{i}_1, \mathtt{i}_2$ as defined in the compilation.

   Let $S''_\delta$ be as constructed above, but if after iterating over $S'_1/S'_2$ a sequence of (pseudo-)actions $\sigma_1$ remains in $S'_{\delta_1}$ in $th_1$ and/or $\sigma_2$ in $S'_{\delta_2}$ in $th_2$, then create $S'''_\delta = \mathit{do}(\sigma_1 \cdot [\mathit{finalize}(th_1), \mathit{backtrack}(th_1)] \cdot \sigma_2 \cdot [\mathit{finalize}(th_2), \mathit{join}(th, \mathtt{c}, \mathtt{i}_2 + 2)], S''_\delta)$. This situation is such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'''_\delta)$ by the above, definition of **comp** for concurrency, and $\mathtt{ax}_{backtrack}$, and it satisfies: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'''_\delta) = (\mathtt{i}', \mathtt{c})$.

$(\delta_1|\delta_2)$**:**

$\Rightarrow$:

By Lemma 2 and definition of **comp**, the first action in thread $th$ in $\sigma = S'_\delta - S_\delta$ is either $\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c})$ or $\mathit{noop}(th, \mathtt{i}, \mathtt{i}_1+1, \mathtt{c})$. We here only show the thesis for the first case, as the second follows analogously. The resulting situation $S_1$ satisfies $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models \mathit{state}(th, S_1) = (\mathtt{i}+1, \mathtt{c})$.

1. First consider the case where $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Then by Lemma 2 and definition of **comp**, $S'_\delta$ is a continuation of $S_1$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_1, S'_\delta)$ and hence, by induction hypothesis, then there is a program $\delta'_1$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta'_1, \widehat{S}')$. Hence, by definition of *Trans* also $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1|\delta_2, S, \delta'_1, \widehat{S}')$.

2. Otherwise, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, then the last action in thread $th$ in $S'_\delta$ can only be $\mathit{noop}(th, \mathtt{i}_1, \mathtt{i}_2+1, \mathtt{c})$, by definition of **comp** and Lemma

2. This action is only possible if for $S'_{\delta_1}$ such that $S'_\delta = do(noop(th, \mathtt{i}_1, \mathtt{i}_2 + 1, \mathtt{c}), S'_{\delta_1})$ we have that $\mathcal{D}_{\delta_1,\mathtt{i},\mathtt{c},\mathtt{i}_1} \models state(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Hence, by induction hypothesis and definition of $\widehat{=}_{th}$, $\widehat{\mathcal{D}} \models Final(\delta'_1, \widehat{S}')$.

$\Leftarrow$:

1. By definition of *Trans* either $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta', \widehat{S}')$ or $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_2, S, \delta', \widehat{S}')$. Without loss of generality we here assume the first case.

   Then, by induction hypothesis, for any $S_{\delta_1}$ such that $filter(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models Thread(th, S_{\delta_1}) \wedge state(th, S_{\delta_1}) = (\mathtt{i}+1, \mathtt{c})$ there is a continuation $S'_{\delta_1}$ such that $S'_{\delta_1} \widehat{=}_{th} \widehat{S}'$ and $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models exec_{th}(S_{\delta_1}, S'_{\delta_1})$. Let $\sigma = S'_{\delta_1} - S_{\delta_1}$.

   The situation $S^*_{\delta_1} = do(noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta)$ satisfies above conditions for any situation $S_\delta$ which is as described in the lemma. Since $noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c})$ is executable in $S_\delta$, by definition of **comp** we get: $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, do(\sigma, S^*_\delta))$.

2. if $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$, then, by induction hypothesis, $\mathcal{D}_{\delta_1,\mathtt{i}+1,\mathtt{c},\mathtt{i}_1} \models state(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Hence, $S'_\delta = do(noop(th, \mathtt{i}_1, \mathtt{i}_2 + 1, \mathtt{c}), do(\sigma, S^*_\delta))$ is such that both $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, and obviously still $S'_\delta \widehat{=}_{th} \widehat{S}'$.

(**while** $\phi$ **do** $\delta_1$)**:**

$\Rightarrow$:

By Lemma 2 and definition of **comp** the first action of $\sigma = S'_\delta - S_\delta$ in thread *th* can only be $noop(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c})$. Thereafter, if (a) $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models \phi(do(noop(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c}), S_\delta))$ or $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models blockeds(th, sp(th, S_\delta), \mathtt{i}+2, \mathtt{c}, S_\delta)$ then the next action in *th* is $test(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})$. Otherwise, (b) it is $test(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})$.

We show the thesis for this case by induction over the cycles of the loop (we refer to this as the inner induction). There are two base cases: case (a) above, and case (b) with zero cycles – intuitively the situation "ends in $\delta_1$". For the inner induction

step we assume (b) and that one cycle less remains. Such in induction is possible, since we assume finite situation terms.

**(a)** In this case we immediately get that $\mathcal{D} \models \neg\phi(S)$ and hence $\mathcal{D} \models \mathit{Final}(\textbf{while } \phi \textbf{ do } \delta_1, \widehat{S'})$ for any $\widehat{S'}$ not mentioning any unmarked actions (this also implies, trivially $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta, \widehat{S'})$). Since, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ does not admit any further actions in *th* and the above are not domain actions, we have $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S'}$ for any such situation.

**(b) zero cycles left:** Let $S^*_\delta = \mathit{do}(\mathit{test}(th, \mathtt{i}{+}1, \mathtt{i}{+}2, \mathtt{c}), \mathit{do}(\mathit{noop}(th, \mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), S_\delta))$. Then $\mathit{filter}(S^*_\delta, \mathcal{D}) = S$, since *test* and *noop* are not domain actions, and, by construction, $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{Thread}(th, S^*_\delta) \wedge \mathit{state}(th, S^*_\delta) = (\mathtt{i}+2, \mathtt{c})$. Let $\sigma' = S'_\delta - S^*_\delta$. Then, $S'_\delta$ is a continuation of $S^*_\delta$ such that $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S^*_\delta, S'_\delta)$. Hence, by induction hypothesis, there is a program $\delta'_1$ and a continuation $\widehat{S'_1}$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S'_1}$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta'_1, \widehat{S'_1})$. Hence, by definition of *Trans* and the above assumption that $\mathcal{D} \models \phi(S)$, it follows that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta'_1; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S'_1})$.

By assumption, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \not\models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$, hence, the second implication holds trivially in this case.

**(b) induction step:** In the inner induction case, there is a prefix $\sigma_1$ of $S'_\delta - \mathit{do}(\mathit{test}(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c}), \mathit{do}(\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), S_\delta))$ such that for $S'_{\delta_1} = \mathit{do}(\sigma_1, \mathit{do}(\mathit{test}(th, \mathtt{i}{+}1, \mathtt{i}{+}2, \mathtt{c}), \mathit{do}(\mathit{noop}(th, \mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), S_\delta)))$ we have $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Hence, by outer induction hypothesis there is a program $\delta'_1$ and a continuation $\widehat{S'_1}$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_{\delta_1} \mathrel{\widehat{=}}_{th} \widehat{S'_1}$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta'_1, \widehat{S'_1})$ and $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1, \widehat{S'_1})$.

Hence, we get by definition of $\widehat{\mathit{Trans}}$ that

$$\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta'_1; \textbf{while } \phi \textbf{ do } \delta_1, S'_1)$$

and since $\widehat{\mathcal{D}} \models \mathit{Final}(\delta_1', \widehat{S}_1')$ further

$$\mathcal{D} \models \quad \widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, S_1') \equiv$$
$$\widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \textbf{while } \phi \textbf{ do } \delta_1, S_1').$$

Applying inner induction hypothesis we get that there is also a program $\delta'$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $S_\delta' \mathrel{\widehat{=}}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta, S, \delta', \widehat{S}')$. Further, again from inner induction hypothesis (in particular if case (a) was used to terminate the induction) if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S_\delta') = (\mathtt{i}', \mathtt{c})$, then also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$.

$\Leftarrow$:

By definition of $\widehat{\mathit{Trans}}$, there is either $\widehat{S}'$ and $\delta'$ such that

$$\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}')$$

or $\widehat{\mathcal{D}} \models \mathit{Final}(\textbf{while } \phi \textbf{ do } \delta_1, S)$.

In the second case, by definition $\mathcal{D} \models \neg\phi(S)$ and hence, as above, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, \mathit{do}([\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})], S_\delta))$, and by definition of **comp** (and **test** in particular), $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, \mathit{do}([\mathit{noop}(th, \mathtt{i}, \mathtt{i}{+}1, \mathtt{c}), \mathit{test}(th, \mathtt{i}{+}1, \mathtt{i}_1+1, \mathtt{c})], S_\delta)) = (\mathtt{i}', \mathtt{c})$, since $\mathtt{i}' = \mathtt{i}_1 + 1$.

The first case is again shown by induction over the cycles of the loop. We will again refer to this induction as the inner induction.

1. In the base case, $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\textbf{while } \phi \textbf{ do } \delta_1, S, \delta_1'; \textbf{while } \phi \textbf{ do } \delta_1, \widehat{S}')$ implies that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta_1', \widehat{S}')$. Hence, by outer induction hypothesis we get that for any thread name $th$, integer $\mathtt{i}+2$, and situation $S_{\delta_1}$ in $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1}$ such that $\mathit{filter}(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{Thread}(th, S_{\delta_1}) \wedge \mathit{state}(th, S_{\delta_1}) = (\mathtt{i}+2, \mathtt{c})$, there exists a continuation $S_{\delta_1}'$ of $S_{\delta_1}$ such that

$S'_{\delta_1} \widehat{=}_{th} \widehat{S'}$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S_{\delta_1}, S'_{\delta_1})$ and if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1, \widehat{S'})$, then also $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$.

For any $S_\delta$ as described in the lemma, we can hence choose $S'_\delta = \mathit{do}(S'_{\delta_1} - S_{\delta_1}, \mathit{do}([\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})], S_\delta))$.

If $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S'})$, which is the case when $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S'})$ and $\widehat{\mathcal{D}} \models \neg\phi(\widehat{S'})$, then we extend $S'_\delta$ by the action sequence $[\mathit{noop}(th, \mathtt{i}_1, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}_1+1, \mathtt{c})]$, which is executable in any situation $s$ for which $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, s) = (\mathtt{i}_1, \mathtt{c}) \wedge \neg\phi(s)$. The former is provided by the above application of (outer) induction hypothesis, and the latter is true, since $S'_\delta \widehat{=}_{th} \widehat{S'}$, which in particular means that these two situations contain the same domain actions (in the same order, ignoring marking), since pseudo-actions by construction do not affect domain fluents, and since $\phi$ cannot mention any bookkeeping fluents.

2. In the inner induction case, there is a shortest prefix $\sigma$ of $\widehat{S'} - S$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_1, S, \delta''_1, \mathit{do}(\sigma, S))$ and $\widehat{\mathcal{D}} \models \mathit{Final}(\delta''_1, \mathit{do}(\sigma, S))$.

   Then, by outer induction hypothesis, as above, for any thread name $th$, integer $\mathtt{i}+2$, and situation $S_{\delta_1}$ in $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1}$ such that $\mathit{filter}(S_{\delta_1}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{Thread}(th, S_{\delta_1}) \wedge \mathit{state}(th, S_{\delta_1}) = (\mathtt{i}+2, \mathtt{c})$, there exists a continuation $S'_{\delta_1}$ of $S_{\delta_1}$ such that $S'_{\delta_1} \widehat{=}_{th} \widehat{S'}$ and $\mathcal{D}_{\delta_1,\mathtt{i}+2,\mathtt{c},\mathtt{i}_1} \models \mathit{exec}_{th}(S_{\delta_1}, S'_{\delta_1}) \wedge \mathit{state}(th, S'_{\delta_1}) = (\mathtt{i}_1, \mathtt{c})$. Consider the situation $S''_\delta = \mathit{do}(\mathit{noop}(th, \mathtt{i}_1, \mathtt{i}+1, \mathtt{c}), \mathit{do}(S'_{\delta_1} - S_{\delta_1}, \mathit{do}([\mathit{noop}(th, \mathtt{i}, \mathtt{i}+1, \mathtt{c}), \mathit{test}(th, \mathtt{i}+1, \mathtt{i}+2, \mathtt{c})], S_\delta))$. This situation is executable in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ as argued above regarding the 'Final' case, and it is such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S''_\delta) = (\mathtt{i}+1, \mathtt{c})$. Hence, by inner induction hypothesis there is a continuation $S'_\delta$ of this situation such that $S'_\delta \widehat{=}_{th} \widehat{S'}$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{th}(S_\delta, S'_\delta)$. And if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta'_1; \mathbf{while}\ \phi\ \mathbf{do}\ \delta_1, \widehat{S'})$ then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Hence the thesis.

**$P(t_1, \ldots, t_n)$ where $P(x_1, \ldots, x_n)$ is a procedure:**

Following the steps of the compilation, we assume that any procedure $P'(x_1, \ldots, x_n)$ has been compiled into $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ and the returned integer was $\mathtt{i}_{P'}$. Note also that we do not consider nested procedure definitions. [1]

The treatment of actual procedure parameters is done the same as for program variables. Effectively, when a procedure is called, the parameters are evaluated and stored in the *map* fluent. Each time an action or condition refers to the formal parameters, reference is made to this map instead (more details can be found in the proof for programs with program variables below).

$\Rightarrow$:

The thesis is shown by induction over the recursions of $P$ (we call this induction again the inner induction, to distinguish it from the outer induction over the structure of programs).

In the base case, the currently executing procedure does not actually call other procedures nor itself. This case follows immediately by (outer) induction hypothesis (it is just a regular program without procedures).

In the induction case, we are assuming that the property holds for situation terms that contain $n$ procedure calls, and show it for $n+1$. By Lemma 2 the only action possible in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ in $th$ in $S_\delta$ is $call(th, P, \mathtt{i}+1, \mathtt{c})$. By definition of $\mathtt{ax}_{common}$, this action stores the return address on the stack and establishes the state $(0, P)$. Hence, $S'_\delta$ is such that $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models exec_{th}(do(call(th, P, \mathtt{i}+1, \mathtt{c}), S_\delta), S'_\delta)$. Hence, by inner induction hypothesis there is a program $\delta'$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $S'_\delta \widehat{=}_{th} \widehat{S}'$ and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_{P(t_1,\ldots,t_n)}, S, \delta', \widehat{S}')$. By definition of

[1] While it would be more descriptive to refer to the resulting theory, here and above, as $\mathcal{D}_{\{P_1(\vec{x_1},\delta_{P_1});\ldots;P_n(\vec{x_n},\delta_{P_n})\};\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, we choose to still refer to it as $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ for parsimony and readability, and understand that all referred procedures have been compiled as well. Note that this is not a problem, since the compilation is purely syntactic and in particular, the order of compiling the procedures does not depend on their semantic dependencies.

*Trans* we then also get $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(P(t_1,\ldots,t_n), S, \delta', \widehat{S}')$. Further, if $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}', \mathtt{c})$ then the last action in thread *th* in $S'_\delta$ can only be $\mathit{return}(\mathit{th})$, due to the definition of **comp**. This action, is only possible in situations $s$ where $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{state}(\mathit{th}, s) = (\mathtt{i}_P, \mathtt{c})$ (cf. Step 3 of the compilation). Since then also $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models \mathit{state}(\mathit{th}, S'_\delta) = (\mathtt{i}_P, P)$ we get again by inner induction hypothesis that also $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$.

$\Leftarrow$:

We again show this, by induction over the number of recursive procedure calls. The base case, where the currently executing program does not actually mention procedure calls, is again trivially given by outer induction hypothesis.

In the induction step we assume that there is a program $\delta'$ and a continuation $\widehat{S}'$ of $S$ in $\widehat{\mathcal{D}}$ such that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(P(t_1,\ldots,t_n), S, \delta', \widehat{S}')$. By definition of *Trans* it is hence the case that $\widehat{\mathcal{D}} \models \widehat{\mathit{Trans}}^*(\delta_{P(t_1,\ldots,t_n)}, S, \delta', \widehat{S}')$. Since there hence remain one less procedure call in the trail of configurations from $\langle \delta_{P(t_1,\ldots,t_n)}, S\rangle$ to $\langle \delta', \widehat{S}'\rangle$, we get by inner induction hypothesis that for any thread name *th*, integer $\mathtt{i}$, and situation $S_{\delta_P}$ in $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P}$ such that $\mathit{filter}(S_{\delta_P}, \mathcal{D}) = S$ and $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models \mathit{Thread}(\mathit{th}, S_{\delta_P}) \wedge \mathit{state}(\mathit{th}, S_{\delta_P}) = (0, P)$, there exists a continuation $S'_{\delta_P}$ of $S_{\delta_P}$ such that $S'_{\delta_P} \mathrel{\widehat{=}}_{\mathit{th}} \widehat{S}'$ and $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models \mathit{exec}_{\mathit{th}}(S_{\delta_P}, S'_{\delta_P})$.

The situation $\mathit{do}(\mathit{call}(\mathit{th}, P, \mathtt{i}+1, \mathtt{c}), S_\delta)$ satisfies this condition for any $S_\delta$ as described in the lemma, and by definition of **comp** $\mathit{call}(\mathit{th}, P, \mathtt{i}+1, \mathtt{c})$ is possible in any such $S_\delta$. Hence, $S'_\delta = \mathit{do}(S'_{\delta_P} - S_{\delta_P}, \mathit{do}(\mathit{call}(\mathit{th}, P, \mathtt{i}+1, \mathtt{c}), S_\delta))$ is such that $S'_\delta \mathrel{\widehat{=}}_{\mathit{th}} \widehat{S}'$ and $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models \mathit{exec}_{\mathit{th}}(S_\delta, S'_\delta)$.

Further, also by inner induction hypothesis, if $\widehat{\mathcal{D}} \models \mathit{Final}(\delta', \widehat{S}')$, then the mentioned situation $S'_{\delta_P}$ is such that $\mathcal{D}_{\delta_P,0,P,\mathtt{i}_P} \models \mathit{state}(\mathit{th}, S'_{\delta_P}) = (\mathtt{i}_P, \mathtt{c})$. In that case, we can append the action $\mathit{return}(\mathit{th})$ to it, which is executable in that situation in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$, due to Step 3 of the compilation, and which has the effect of establishing the state

denoted by the highest stack position. Following the definition of $\mathtt{ax}_{\mathrm{procs}}$, the action $call(th, P, \mathtt{i}+1, \mathtt{c})$ had the effect of establishing the value $(\mathtt{i}+1, \mathtt{c})$ for this. Hence, $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, do(return(th), S'_\delta)) = (\mathtt{i}+1, \mathtt{c})$, i.e. the thesis, since $\mathtt{i}' = \mathtt{i}+1$.

We only outline the remaining cases, since they are all quite similar to one of the above.

($\delta'^*$)**:**

This case follows in close analogy to the case of **while**–loops, just replacing the *test* actions by *noop* actions.

(**if** $\phi$ **then** $\delta_1$ **else** $\delta_2$)**:**

This case follows in close analogy to the case of non-deterministic choice of subprograms $(a|b)$, but replacing the initial *noop* actions with *test* actions.

($\delta^{\|}$)**:**

This case follows by induction over the number of concurrent iterations, where the base case is that of not actually executing $\delta$ even once, and the induction step is provided by the (outer) induction step for normal concurrency.

($\delta_1 \rangle\!\rangle\, \delta_2$)**:**

$\Rightarrow$:

This case follows in analogy to the normal concurrency case ($\delta_1 \parallel \delta_2$). If $\delta_1$ is executed until completion before any actions are executed in $\delta_2$, the case is just like for $\|$. On the other hand, if actions are performed in $th_2$ before then, then by definition of **comp** and Step 4 of the compilation, no actions were possible in that situation in $th_1$. From the latter, by induction hypothesis for the $\Leftarrow$ direction, it follows that also there don't exist $\delta'_1, \widehat{S}'$ such that $\widehat{S}' - S$ contains unmarked actions and $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1, S, \delta'_1, \widehat{S}')$. Hence, by definition of *Trans*, there is a program $\delta'_2, \widehat{S}'$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta_1 \rangle\!\rangle\, \delta_2, S, \delta_1 \rangle\!\rangle\, \delta'_2, \widehat{S}')$.

The remainder is again analogous to the $\|$ case.

$\Leftarrow$:

Again, the case where the first transition is over $\delta_1$ is analogous to the $\|$ case. If $\delta_2$ executes, then $\mathcal{D} \models (\nexists \delta_1', S').\mathit{Trans}(\delta_1, S, \delta_1', S')$, by definition of *Trans*. Hence, again by induction hypothesis of the $\Rightarrow$ direction, there is no executable continuation whose first domain action is in $th_1$, hence allowing actions in $th_2$ to execute (either directly, or after executing a number of pseudo-actions first, followed by the *backtrack*($th$) action). The remaining reasoning is as in the $\|$ case.

□

*Proof of Lemma 3 for programs with the $\pi$ construct:*

Our solution for treating programs with the $\pi$ construct and hence program variables is similar to Skolemization. In ConGolog, $\pi(v, \delta)$ is interpreted as the execution of the program $\delta$ where all occurrences of the constant $v$ are substituted by a new existentially quantified variables. This method is not possible in our compilation, since $\pi$ can appear in loops, whose body we only want to consider once during compilation. Instead, we replace these existentially quantified variables with functional fluents. Similar to Skolemization, these functions need to be relative to the context the variable appears in, namely the stack position (for $\pi$s in recursive procedures) and the thread (for $\pi$s inside of the $\delta^{\|}$ construct, or inside a procedure which is called in two concurrent threads).

By definition of **comp**, a $\pi$ construct causes the execution of the $pi(th, v, x, \mathtt{c}, \mathtt{i})$ action, whose effect due to $\mathtt{ax}_\pi$ is that $map(th, p, v) = x$, where $p$ is the current value of the stack pointer. Note that $x$ is a free parameter – not mentioned in the preconditions. It can hence be any object. Since the $\pi$ action is the only action affecting the *map* fluent, this value in *map* pertains until the same construct is visited again. Note that we disallow redefinitions of program variables, i.e. for instance $\pi(v, A(v); \pi(v, B(v)))$ is not allowed in the original program. This is not a restriction, since any such program

could be transformed to be free of such redefinitions, by simply renaming the program variables.

The thesis then follows by inspection and induction over the nesting depth of program variables. Recall that program variables can only occur in places of action parameters and in conditions. In both, as provided by **comp** for primitive actions and procedure calls and by the **test** and **rtest** functions for conditions, these occurrences are forced to be equal to the values stored in *map*.

The base case of the induction is for the case of a program without program variables and is immediately provided by the above proof for such programs.

For the induction step, let $\delta$ be a program with $n$ program variables. Then by above considerations, induction hypothesis, and the fact that *pi* is a pseudo-action and hence filtered out by *filter*: for any ground situation term $S$, for program $\pi(v, \delta)$, there is an object $O$, a situation $\widehat{S}'$, and a program $\delta'$ such that $\widehat{\mathcal{D}} \models \widehat{Trans}^*(\delta|_{v/o}, S, \delta', \widehat{S}')$ if and only if there is a sequence of ground action terms $\sigma$ in $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'}$ such that $S'_\delta = do(\sigma, do(pi([0], v, O, main, 1), S_\delta))$ such that $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models exec_{th}(S_\delta, S'_\delta)$ and $S'_\delta \mathrel{\widehat{=}}_{th} \widehat{S}'$ for any situation term $S_\delta$ as described in the lemma. And further, if $\widehat{\mathcal{D}} \models Final(\delta', \widehat{S}')$ then also $\mathcal{D}_{\delta,\mathtt{i},\mathtt{c},\mathtt{i}'} \models state(th, S'_\delta) = (\mathtt{i}', \mathtt{c})$. Hence the thesis holds for programs with $n + 1$ program variables and hence, by induction, for any program with any number of program variables. □

### C.3.3 Proof of the Theorem

The theorem follows from Lemma 3 for the special case of $S = S_0$, $th = [0]$, $\mathtt{i} = 0$, and $\mathtt{c} = main$: Since $S'_\mathcal{P}$ cannot mention any actions not in $[0]$ or its descendants, we have that $exec_{th}(S_0, S'_\mathcal{P})$ implies $executable(S'_\mathcal{P})$. Further, by definition of $\mathrel{\widehat{=}}_{th}$ also $\widehat{S}'$ cannot mention any marked actions, and hence, following Lemma 3

$$\mathcal{D}_\mathcal{P} \models executable(S'_\mathcal{P}) \wedge state([0], S'_\mathcal{P}) = (\mathtt{i}_{main}, main)$$

iff there exists a program $\delta'$ such that for $S' = \mathit{filter}(S'_{\mathcal{P}}, \mathcal{D})$ we have that

$$\mathcal{D} \models \mathit{Trans}^*(\mathcal{P}, S_0, \delta', S') \wedge \mathit{Final}(\delta', S')$$

which by definition is equivalent to $\mathcal{D} \models \mathit{Do}_2(\mathcal{P}, S_0, S')$. □

## C.4 Proofs of Theorems 10 and 11

These two theorems both rely on the fact that the compilation visits each program construct and logical connective mentioned in a condition exactly once, and that each time a constantly bounded number of additional axioms are introduced, each of size at most $n$. This is shown in the following lemma.

**Lemma 4.** Let $\delta$ be any ConGolog program of size $n$, whose set of free program variables `e` has size $k$, and let `i` be any integer, and `c` any procedure name. Let further be $l$ the maximal cardinality of formal parameters of all the procedures that are called in the program. The invocation of $\mathbf{comp}(\delta, \mathtt{i}, \mathtt{e}, \mathtt{c})$ makes at most $n-1$ recursive calls to **comp**, and each invocation adds at most a constant number of sentences to the set of returned sentences, each of size at most $O(max(k, l))$.

*Proof:* By inspection. It is easy to see from the definition of **comp** that the sum of the sizes of all sub-programs appearing as parameters in the recursive invocations is less or equal to $n-1$. Further, the cardinality of the set of sentences added to the eventually returned set `ax` is bound by a constant in all cases, since all auxiliary functions return only a constant number of sentences. The size of each such sentence is in $O(max(k, l))$, since the only parametrized connective appearing in the definitions is $\bigwedge_{\Psi}$, where $\Psi$ is a set of cardinality $\leq max(l, k)$. □

*Proof of Theorems 10 and 11*: Let as before be $\mathcal{P} = \{P_1(\vec{t_1}, \delta_{P_1}); \ldots; P_n(\vec{t_n}, \delta_{P_n}); \delta_{main}\}$, and let $m$ be the size of $\mathcal{P}$. Then the sum of the number of occurrences of $\pi$ constructs

$k$, and the maximal cardinality of formal procedure parameters $l$ must be $< m$. Hence, by Lemma 4, the set `AX` (see Step 2, p. 121) is of size $O(m^2)$. Since both the time and the space complexity of each remaining step of the compilation is linear in the size of `AX`, we get the thesis. □