DISSERTATION

Advances in Answer Set Planning

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

Institut für Informationssysteme E 184/3
Abteilung für Wissensbasierte Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Dipl.-Ing. Axel Florian Polleres
9225749
Klausgasse 35, 1160 Wien

27. August 2003

Dedicated to my parents, Mechthild and Herbert.

Kurzfassung

Seit den Anfängen der Forschung im Bereich der Künstlichen Intelligenz ist Automatisches Planen ein wichtiger Forschungsgegenstand. Als das "klassische Planungsproblem" bezeichnet man hier das Auffinden einer Folge vordefinierter Aktionen um von einem gegebenen Anfangszustand in einen gewünschten Zielzustand zu gelangen, wobei lediglich das Repertoire an Aktionen, sowie deren Vorbedingungen und Effekte bekannt sind. Konkrete Problemstellungen reichen hier von der klassischen Routenplanung bis hin zur automatischen Zusammenstellung von Web-Services zur Erledigung einer bestimmten Aufgabe, etc.

Neben der Suche nach konkreten Algorithmen zur Lösung solcher Planungsprobleme spielen auch formale Sprachen zur Beschreibung von Aktionen und Planungsdomänen eine ganz wesentliche Rolle. Während klassische Planungssprachen wie beispielsweise STRIPS [FN71] oder PDDL [GHK+98] von einem vollständig definierten Anfangszustand sowie deterministischen Effekten aller Aktionen ausgehen, gelten diese Annahmen in realistischen Szenarien oft nicht. Hier erweisen sich flexiblere Aktionsbeschreibungssprachen aus dem Bereich der Wissensrepräsentation als nützlich, sobald es darum geht, nicht-klassisches Planen unter unvollständigem Wissen zu modellieren. Allerdings bringt die höhere Ausdrucksstärke dieser Aktionsbeschreibungssprachen auch eine höhere Komplexität bei der Berechnung von Plänen mit sich: Klassische Suchalgorithmen können zum Lösen von Planungsproblemen, die in solchen Sprachen formuliert sind, nicht mehr ohne weiteres angewandt werden. Hier bieten sich stattdessen deklarative Methoden an, beispielsweise Methoden der deklarativen logischen Programmierung.

Die vorliegende Dissertation geht auf diese Problemstellungen in mehrerlei Hinsicht ein. Zu Beginn wird die deklarative Planungssprache $\mathcal{K}^c$ eingeführt, die auf bestehenden Aktionsbeschreibungssprachen basiert und diese um Konzepte aus dem Bereich der deklarativen Logikprogrammierung erweitert. Die Sinnhaftigkeit dieser Erweiterungen wird anhand zahlreicher Planungsprobleme demonstriert, welche sich in $\mathcal{K}^c$ zum Teil einfacher beschreiben lassen als in vergleichbaren Sprachen. Nach der Definition der Syntax von $\mathcal{K}^c$ werden verschiedene Semantiken für das Planen unter unvollständigem Wissen eingeführt: *optimistische* Pläne sind Aktionsfolgen, die den Zielzustand nur möglicherweise erreichen, während *sichere* Pläne das Erreichen des Zielzustandes unter allen Umständen garantieren. Weiters wird auf die Erweiterung dieser beiden Semantiken um Aktionskosten eingegangen. Hier spielt einerseits die Berechnung von kostenoptimalen Plänen eine Rolle, andererseits sind oft auch Pläne ausreichend, die lediglich das Einhalten eines bestimmten Kostenlimits garantieren.

Bei der Untersuchung dieser Aspekte wird besonders auf die genaue Analyse der Berechnungskomplexität der beschriebenen Semantiken der Sprache $\mathcal{K}^c$ Wert gelegt.

Anschließend wird die Übersetzung von Planungsproblemen in disjunktive logische Programme beschrieben, welche unter der sogenannten *Answer Set* Semantik ausgewertet werden können. Zunächst wird auf allgemeine Methoden zur Problemlösung mithilfe des "*guess and check*" Paradigmas der Answer Set Programmierung eingegangen. Dabei wird eine neu entwickelte Methode zur Generierung von integrierten Answer Set Programmen aus separaten "guess" und "check" Teilen vorgestellt, die zur Lösung komplexer Planungsprobleme, wie beispielsweise der Berechnung sicherer Pläne verwendet werden kann.

Answer Set Programmierung ist inzwischen ein weithin akzeptiertes Werkzeug des deklarativen Problemlösens geworden, was nicht zuletzt am Vorhandensein effizienter Systeme zur Evaluierung von Answer Set Programmen liegt, wie beispielsweise das am Institut seit etlichen Jahren entwickelte DLV System.

Anhand der beschriebenen, theoretischen Methoden wurde basierend auf DLV ein Planungssystem entwickelt und implementiert. Nach einer genauen Beschreibung des Systems $\texttt{DLV}^{\mathcal{K}}$ wird näher auf die experimentelle Evaluierung der vorgestellten Methoden eingegangen, wobei sich $\texttt{DLV}^{\mathcal{K}}$ als durchaus konkurrenzfähig gegenüber vergleichbaren Planungssystemen erweist.

Ein weiterer Abschnitt der Arbeit widmet sich der Wissensrepräsentation in der Sprache $\mathcal{K}^c$. Dabei wird die Modellierung von Planungsproblemen in $\mathcal{K}^c$ anhand zahlreicher Beispiele aus der Literatur aber auch anhand neuer Planungsdomänen erläutert.

Ein eigenes, in sich abgeschlossenes Kapitel behandelt abschließend ein praktisches Anwendungsszenario zur Verwendung von Planungsmethoden im Bereich der Überwachung und des Designs von Multi-Agenten-Systemen.

Abstract

Planning is a challenging research area since the early days of Artificial Intelligence. The *planning problem* is the task of finding a sequence of actions leading an agent from a given initial state to a desired goal state. Whereas classical planning adopts restricting assumptions such as complete knowledge about the initial state and deterministic action effects, in real world scenarios we often have to face incomplete knowledge and non-determinism. Classical planning languages and algorithms do not take these facts into account. So, there is a strong need for formal languages describing such non-classical planning problems on the one hand and for (declarative) methods for solving these problems on the other hand.

In this thesis, we present the action language $\mathcal{K}^c$, which is based on flexible action languages from the knowledge representation community and extends these by useful concepts from logic programming. We define two basic semantics for this language which reflect optimistic and secure (i.e. sceptical) plans in presence of incomplete information or nondeterminism. These basic semantics are furthermore extended to planning with action costs, where each action can have an assigned cost value. Here, we address optimal plans as well as plans which stay within a certain overall cost limit.

Next, we develop efficient (i.e. polynomial) transformations from planning problems described in our language $\mathcal{K}^c$ to disjunctive logic programs which are then evaluated under the so-called *Answer Set Semantics*. In this context, we introduce a general new method for problem solving in Answer Set Programming (ASP) which takes the genuine "*guess and check*" paradigm in ASP into account and allows us to integrate separate "guess" and "check" programs into a single logic program.

Based on these methods, we have implemented the ASP-based planning system $\texttt{DLV}^{\mathcal{K}}$ which we describe in detail. We furthermore discuss problem solving and knowledge representation in $\mathcal{K}^c$ using $\texttt{DLV}^{\mathcal{K}}$ by means of several examples from the literature but also novel elaborations. The proposed methods and the $\texttt{DLV}^{\mathcal{K}}$ system are also evaluated experimentally and compared against related approaches.

Finally, we present a practical application scenario from the area of design and monitoring of multi-agent systems. As we will see, this monitoring approach is not restricted to our particular formalism.

Acknowledgements

I am grateful to a number of people without whom writing this thesis would not have been possible. First of all, I would like to thank Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer who are co-authors of most of my publications for their long-term cooperation. Among these, I am most indebted to my supervisor Thomas Eiter who gave me the opportunity to work in his group as a research assistant and encouraged me throughout all phases of my work over the last few years. He always had an open ear for my (numerous) questions and let me profit a lot from his tremendous theoretical knowledge. Special thanks go also to Gerald and Wolfgang for their patience concerning technical questions and for many fruitful discussions. I would also like to thank Jürgen Dix, Michael Fink, and Yingqian Zhang for their collaboration in our work on planning and monitoring.

Furthermore, I want to thank Chitta Baral who was a guest lecturer at our institute some months ago and all the lecturers of the PLANET Summer schools 2002 in Greece and 2003 in Italy for deepening my interests in planning and reasoning about actions. Besides, I want to thank the organizers of the PLANET Summer schools 2002 and 2003 and the organizers of the ICAPS 2003 Doctoral Consortium for giving me the opportunity to participate in these events and to meet and discuss with experienced scientists and like-minded researchers.

I would like to express my gratitude to all members of our department for the friendly and amicable working atmosphere, especially to our secretary, Elfriede Nedoma, and to our technician, Matthias Schlögel, for their constant and reliable support.

Many thanks go also to my friends Monika Lanzenberger and Markus Rester for proof-reading parts of my thesis.

Last, but not least I want to thank my family and my close friends Andrea, Felix, Hermann, and Nicki for their "social support" all over the past few years.

This thesis was supported by the Austrian Science Funds (FWF) under the project number P14781-INF.

Contents

Chapter 1

Introduction

Planning has been a challenging problem since the early days of Artificial Intelligence research. The very beginning of research in this area dates back already to the late 50s, when McCarthy et al. coined the term "Artificial Intelligence" in the famous Dartmouth Conference in 1956. Driven by the ambition to model common-sense reasoning in an automated fashion, McCarthy's "Advice Taker"[McC58] or the "General Problem Solver" [NSS59, NS63] by Newell et al. may be viewed as starting points for a steadily evolving research branch in AI since then.

Precisely speaking, the classical "planning problem" is nowadays conceived as the task of finding a sequence of actions bringing an agent from a given initial state towards a given goal state. The "input" of a planning problem consists of a description of the initial state, the goal state, and the actions which can be performed by the agent in terms of their preconditions and effects on the environment. Solutions to such problems (i.e., plans) correspond to sequences of actions which guarantee to reach a state where the desired goal holds. However, as opposed to this "classical" definition of planning, the initial state might be specified incompletely, actions might have nondeterministic effects, and furthermore actions can have different costs which we want to minimize, etc.

From a knowledge representation perspective, there is a strong need for appropriate formalizations of such problems in high-level languages which enable the user to model planning domains in a declarative way. Last but not least, the definition of such languages based on a clear formal semantics is essential with respect to the development of algorithms for solving such problems.

Planning Languages

Existing Planning languages such as STRIPS [FN71] and its descendants ADL [Ped89] or PDDL [GHK+98, FL03] have become a quasi-standard for state-of-the-art planning systems. These languages are mainly tailored for solving classical planning problems with limiting assumptions such as complete knowledge about the world state and fully deterministic action effects. Furthermore, STRIPS like formalisms build up a very strict syntactic corset on formalizing actions and change. Nevertheless, thanks to the efforts made in language standardization, mainly with the PDDL language, this has lead to many successful plan-

ning systems based on various underlying approaches such as reductions to propositional satisfiability (SAT) or heuristic search methods.

On the other hand, more flexible, declarative action languages, for instance $\mathcal{A}$ [GL93], $\mathcal{C}$ [GL98a] or recently $\mathcal{K}$ [EFL$^+$03b] – which is a main subject of the present work – have been developed by the knowledge representation community. The main focus of these action languages is a flexible, "natural language" like description of actions and change. Furthermore, languages like $\mathcal{C}$ and $\mathcal{K}$ are well-suited for expressing nondeterminism and incomplete knowledge. The main concern of these languages is more on general reasoning about actions and change than on plan generation. So, whereas these languages are clearly defined in terms of formal semantics, implemented planning systems are rare. One reason is that the high flexibility of these languages in formalizing actions comes at a cost: Classical search algorithms and methods for planning in STRIPS-like domains are no longer directly applicable. Methods for declarative problem solving seem more appropriate for these languages.

Declarative Problem Solving

By "declarative" problem solving, as opposed to procedural methods, we refer to methods which allow us to specify the problem in a formal language and then deploy an inference mechanism rather than having to describe the operational structure of *how* to solve the problem beforehand. For instance, formalizations of problems as propositional logical formulae, such that valid truth assignments witness desired solutions, fall into this class, where we can use existing SAT checkers, like [BS97, Zha97], to solve the problem. A particular important formalism which allows for such declarative problem solving is logic programming under the Answer Set Semantics [GL91]. The problem at hand is formalized as a set of rules which are evaluated under a model based semantics, where these models are also called "Answer Sets". While the rules are syntactically similar to PROLOG, we emphasize that Answer Set Programming (ASP) is really a declarative formalism, i.e. the ordering of rules and literals within rule heads and bodies does not affect the semantics. The Answer Set Programming paradigm has been widely accepted as a knowledge representation formalism well capable of formalizing search problems involving default assumptions, incomplete knowledge, nondeterminism, etc. (cf. [PC01, Bar03]). The development of efficient solvers for ASP such as DLV [LPF$^+$02], SMODELS [SNS02], ASSAT [LZ02], or CMODELS [Bab02] has further promoted this method recently.

Declarative Planning

Strictly speaking, any system capable of solving arbitrary planning problems formally defined in a high-level action language can, by our definition, be considered as a tool for declarative problem solving. However, increasing expressiveness of these languages makes the development of such a system hard.

The basic idea behind declaratively solving planning problems specified in high-level action languages is as follows: For any formal language L_1 which allows for efficient (polynomial) reductions to another formal language L_2, any system or formalism capable of solving L_2 also can be used for solving L_1. We basically want to use such reductions in order to translate planning problems to another declarative formalism for which efficient

solvers already exist. This methodology has been successfully applied in planning by translations of classical planning problems into propositional satisfiability (SAT), resulting in competitive classical planning systems (e.g. [KS92, KS99]).

A key question at the start of this work was how far we can get, building up a planning system fully relying on ASP techniques, especially with respect to non-classical planning, where uncertainty comes into play. Concerning ASP, many ad hoc formalizations of planning have been proposed and several attempts have been made to translate existing action languages into logic programming (e.g. [SZ95, DNK97, LT99]). Here, the action language $\mathcal{K}$ which is one main subject of the present thesis plays an outstanding role:

$\mathcal{K}$ itself adopts many useful concepts of ASP. In the course of this work we will show the usefulness of these additional language features in the field of planning . Besides, $\mathcal{K}$ allows for a more direct compilation to ASP than previous languages which we have used and implemented in the $\texttt{DLV}^{\mathcal{K}}$ Planning System.

Contributions

The present thesis comprises the results of the author's work conducted on action languages and efficient translations to Answer Set Programming within a research project under the title "A Declarative Planning System Based on Logic Programming"[1]. The project was fundamentally based on the following preliminary work: First thoughts on using the DLV System for planning purposes appeared in the proceedings of the "Workshop on Logic Programming (WLP'99)" [EFL+00b]. Preliminary results on the action Language $\mathcal{K}$ and a first prototype of a planning system based on DLV have been published in the proceedings of the "First International Conference on Computational Logic (CL'2000)" and in the author's master's thesis [Pol01]. During the last two years this work has been extended by novel theoretical results and newly developed methods.

Main Results

In particular, we will make the following contributions:

(1) We will introduce the action language $\mathcal{K}$ and thoroughly analyze syntactic and semantic features. As for planning under incomplete knowledge and nondeterminism, we will define two semantics for planning problems in our language, namely optimistic and secure (conformant) plans. In analogy to brave and cautious reasoning in Answer Set Programming, the former denotes plans which reach the goal in some possible evolutions of the world when executing the plan, whereas the latter denotes plans which reach the goal under any contingencies. Furthermore, we will analyze both of these semantics complexitywise, where we set value on a more in-depth discussion of the results than in previous works.

(2) We extend the language with action costs which leads to the novel language $\mathcal{K}^c$. We analyze the impact of this extension on complexity and define two more semantics, namely admissible and optimal planning. These new semantics are orthogonal to the previously mentioned optimistic and secure plans. I.e., both semantics can be combined with optimistic

[1]Sponsored by the Austrian Science Funds (FWF) under the project number P14781-INF.

and secure planning. The terms "admissible" and "optimal" denote plans where action costs stay within some given cost limit or are optimal with respect to costs, respectively.

(3) We relate these results to logic programming by providing appropriate (polynomial) transformations from optimistic and secure planning problems in $\mathcal{K}^c$ to Answer Set Programming. We further extend these transformations with respect to admissible and optimal planning. Here, we will discuss interleaved computations via separate "guess" and "check" programs, where one logic program serves to guess an optimistic plan and another logic program is used to check whether a particular plan is secure. Next, we will discuss cases where these two separate programs can be combined into a single, integrated logic program directly encoding secure plans.

(4) To this end, we will use a novel, general method for automatic integration of separate "guess" and "check" programs by meta-interpretation of logic programs. Our approach reconciles pragmatic problem solving with the genuine "guess and check" approach in Answer Set Programming beyond NP problems. We will discuss the properties of this method and applicability in the field of planning. In particular, we are able to define a general transformation of checking plan security which itself is in general a problem infeasible by efficient reductions to SAT but solvable in ASP.[2]

For the integrated encodings of secure planning, as mentioned in **(3)**, we will define proper syntactic subclasses of our language $\mathcal{K}$ where a single logic program for computing secure plans can be generated.

As we will show, our general method can moreover be fruitfully applied to other hard problems. Particularly, we will show how to use this method on Quantified Boolean Formulae and a problem from the business domain: "Strategic Companies".

(5) We will provide an overview on how to model planning domains in our language $\mathcal{K}^c$ by means of several well-known and also novel planning examples. We will focus on the usefulness of our added features compared with other action languages, and exemplify design principles of our language.

(6) We will describe an implementation of our methods in form of the $\texttt{DLV}^{\mathcal{K}}$ planning system which, to our knowledge, is the only planning system based on Answer Set Programming so far. We will perform experimental evaluation of the proposed methods and compare the $\texttt{DLV}^{\mathcal{K}}$ system against other planning systems, namely as well as against the integrated encodings mentioned in **(4)**.

(7) Finally, we will show the applicability of our planning approach in a realistic scenario from the area of design and monitoring of Multi-Agent Systems. As we will see, this monitoring approach is not restricted to our particular formalism, but establishes a general method for using planning in monitoring of Multi-Agent Systems.

Original results contained in this thesis have been published as refereed articles in the proceedings of several international workshops and conferences: Progress of the $\texttt{DLV}^{\mathcal{K}}$ system

[2] ASP is capable of expressing problems on the second level of the Polynomial Hierarchy of complexity classes whereas in SAT only problems in the class NP can be expressed, which are widely believed to be easier (cf. Section 2.2).

has been reported in the proceedings of the *"IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information"* [EFL+01b] and the *"6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)"*[EFL+01a]. Preliminary results on our extended language $\mathcal{K}^c$ and a description of the extended system features appeared in the proceedings of the *"8th European Conference on Artificial Intelligence (JELIA)"*[EFL+02a, EFL+02b]. Furthermore, our planning approach for monitoring and design of Multi-Agent Systems will be presented at the *"26th German Conference on Artificial Intelligence"*[DEF+03]. A proposal for future system extensions has been presented in the *"Printed Notes of the ICAPS-03 Doctoral Consortium"* [Pol03]. Preliminary results on our general method for "guess" and "check" integration in Answer Set Programming will be presented at the *"APPIA-GULP-PRODE 2003 Joint Conference On Declarative Programming"* [EP03].

Furthermore, a description of the DLV$^{\mathcal{K}}$ system recently appeared in the *"Artificial Intelligence"* Journal [EFL+03a]. An article on syntax and semantics of our action language $\mathcal{K}$ will appear in *"ACM Transactions on Computational Logic (TOCL)"* [EFL+03b]. Another contribution concerning planning under action costs will appear in the upcoming issue of the *"Journal of Artificial Intelligence Research (JAIR)"* [EFL+03c] where the extended language $\mathcal{K}^c$ is described along with respective extensions of the DLV$^{\mathcal{K}}$ System.

The thesis also contains novel, previously unpublished results. Particularly, this applies to the proposal of further language extensions in Chapter 7 which partly have been described in [Pol03] and the general method for checking plan security in Chapter 4.

The remainder of this thesis is organized as follows. We will begin with a recapitulation of fundamental concepts in Chapter 2. Here, we will recall the necessary concepts of Answer Set Programming, Complexity, and Planning, and shortly review the most relevant planning languages and existing approaches.

Next, we will introduce syntax and semantics of the action language $\mathcal{K}$ and its extension $\mathcal{K}^c$. Semantics will be defined in terms of optimistic and secure plans for the basic language, which we will extend to admissible and optimal plans for planning with action costs. This will be followed by a thorough complexity analysis of the different semantics.

In Chapter 4, we will first review and extend general methods of problem solving in ASP as pinpointed in **(4)**. Then, we will deploy these methods in the field of planning by describing transformations of $\mathcal{K}^c$ planning problems to logic programs, such that answer sets of the resulting programs coincide with plans under the desired semantics (i.e. optimistic, secure, admissible, and optimal plans). We will further sketch algorithms for the interleaved computation of such plans, where direct transformations are not feasible.

Chapter 5 then gives an overview over the implemented DLV$^{\mathcal{K}}$ System, its usage, features and architecture. We demonstrate how we integrated the methods from the previous chapter in a fully operational planning system based on DLV.

The following chapter on knowledge representation discusses encodings of several well-known, but also some novel planning domains in $\mathcal{K}^c$. We particularly emphasize on the novel features of our language and system, showing how to solve classical planning problems, secure (conformant) planning under incomplete knowledge, and planning under various optimization criteria using action costs. We further discuss general design principles of the formalization of planning problems in $\mathcal{K}^c$.

Language and system extensions, which have not yet been implemented will be sketched in Chapter 7.

Experimental evaluation of the DLV$^{\mathcal{K}}$ system is reported in Chapter 8. Here, we first compare DLV$^{\mathcal{K}}$ against other conformant planning systems on benchmark problems from the well-known "Bomb in the Toilet" domain. For comparison, we used CPLAN [CGT02], CMBP [CR00], GPT [BG00] and SGP [WAS98], some recent conformant planners. Next, we report on experiments concerning optimal planning. Since the number of comparable systems wrt. optimal planning in the form we propose it is rare, we restrict ourselves to domains which are hard to express for others. We show the limits of our system on some special elaborations of the "Blocks World" and "Traveling Salesperson" domains. We compare DLV$^{\mathcal{K}}$ against CCALC [McC99a], a generic system for reasoning about actions and change which offers features similar to DLV$^{\mathcal{K}}$, where possible. Furthermore we test the integrated encodings mentioned in **(4)**. On the one hand, we experimentally compare this method against existing ad hoc encodings for Quantified Boolean Formulae and "Strategic Companies". On the other hand, we are interested in the performance of integrated encodings for secure planning compared with the interleaved computation of DLV$^{\mathcal{K}}$.

Our general monitoring approach for Multi-agent-systems mentioned in **(7)** will be introduced in Chapter 9.

A detailed discussion of related works and systems follows in Chapter 10. Here we discuss the correspondence to other planning languages, related complexity results, and related approaches wrt. logic-based planning, planning under incomplete knowledge, and optimal planning.g

Chapter 11 concludes the thesis with a short summary and final remarks.

Chapter 2

Preliminaries

2.1 Answer Set Programming

In this section, we will review a commonly used semantics for a wide class of logic programs known as the *Answer Set Semantics* [GL91] and present two efficient engines for computing answer sets (i.e. solutions of such programs), namely DLV and SMODELS (and its extension GNT). Syntactically, we will consider so called Extended Disjunctive Logic Programs (EDLPs). These are logic programs which allow for disjunction in rule heads and two forms of negation, classical and default negation (also known as "negation as failure" from Prolog).

2.1.1 Historical Overview

Answer Set Programming (ASP) [PC01, Bar03], i.e. logic programming under the Answer Set Semantics [GL90, GL91] is widely proposed as a useful tool for various problem solving tasks in Artificial Intelligence. *Historically speaking, the Answer Set Semantics for logic programs has its roots in the stable model semantics [GL88] of normal logic programs (also known as* general logic programs *[Llo87]). This class is obtained from ordinary logic programs (that consist of rules that are effectively Horn clauses) by allowing the use of a form of negation* – negation as failure to prove *[Llo87] – in the bodies of rules. Due to close interconnections to Reiter's* Default Logic *[Rei80], this form of negation is also known as* default negation. *Default negation differs from classical negation in propositional logic and it is therefore quite natural that Gelfond and Lifschitz proposed a logic programming approach which allows for both negations [GL90]* [3] where the term "answer sets" was introduced. Furthermore, in [GL91] Gelfond and Lifschitz extended their semantics to disjunction in rule heads. Przymusinski [Prz91] presented similar ideas, but in a more general setting.

The latest generalization of Answer Set Programming allows for negation as failure also in the heads of rules [LW92, Lif02]. While this extension is syntactically and semantically straightforward it is (so far) not supported by the existing solvers such as DLV and SMODELS. In fact, it has been shown that negation as failure in rule heads does not increase the

[3] quoted from [Jan01]

expressive power of logic programs [Jan01]. Here, we consider negation as failure only in rule bodies.

2.1.2 Syntax

Let σ^{pred}, σ^{con} and σ^{var} be disjoint sets of predicate, constant, and variable symbols, respectively. In accordance with DLV notations which we will often use in the following we assume that σ^{con} is a set of integer constants, string constants beginning with a lower case letter, and '"' quoted strings, σ^{pred} is a set of string constants, and σ^{var} is the set of string constants beginning with an upper case letter. More precisely, σ^{pred}, σ^{con} and σ^{var} are defined by the following regular expressions:

$$\begin{array}{lcl} \sigma^{con} & = & ([0-9]+)|([a-z][A-Za-z\_0-9]*)|("[A-Za-z\_0-9]*") \\ \sigma^{pred} & = & ([A-Za-z][A-Za-z\_0-9]*) \\ \sigma^{var} & = & ([A-Z][A-Za-z\_0-9]*) \end{array}$$

Given $p \in \sigma^{pred}$ an *atom* is defined as $p(t_1, \ldots, t_n)$, where n is called the arity of p and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$.[4] Whenever arity $n = 0$, we say that p is a *propositional* atom. A *literal* is an atom a or its negation $\neg a$, where "$\neg$" is the true (classical) negation symbol, for which we also use the customary "–". We write $|a| = |\neg a| = a$ to denote the atom of a literal. Further, we write $\neg.l$ to denote the complement of a literal, i.e. $\neg.a = \neg a$ and $\neg.\neg a = a$, respectively.

Definition 2.1. *Finally, a* rule *is of the form*

$$h_1 \texttt{ v } \ldots \texttt{ v } h_l \texttt{ :- } b_1, \ldots, b_m, \texttt{ not } b_{m+1}, \ldots \texttt{ not } b_n. \tag{2.1}$$

where each h_i (b_j) is a literal and not *is the symbol for negation as failure.*

Definition 2.2. *An* extended disjunctive logic program *(EDLP, or simply logic program)* Π *is defined as a set of rules r of the form (2.1).*

We denote $Head(r) = \{h_1, \ldots h_l\}$, $Body^+(r) = \{b_1, \ldots, b_m\}$, $Body^-(r) = \{b_{m+1}, \ldots, b_n\}$, and $Body(r) = Body^+(r) \cup Body^-(r)$. Furthermore, we denote $Lit(\Pi)$ as the set of all literals occurring in Π. A rule with $|Head(r)| = 1$ and $Body(r) = \emptyset$ is called a *fact*. Rules with $Head(r) = \emptyset$ are called *constraints*. Programs without disjunction, i.e. for all $r \in \Pi$ $|Head(r)| \leq 1$ are called *normal* logic programs. For normal rules with a non-empty head, we occasionally write $Head(r) = l$ instead of $Head(r) = \{l\}$ when clear from the context. Programs without negation as failure,i.e. for all $r \in \Pi$ $Body^-(r) = \emptyset$ are called *positive*. Programs without classical negation are called (non-extended) DLPs.

2.1.3 Ground Instantiation

The semantics of EDLPs is defined in terms of programs without variables. In order to eliminate variables from a program, we will now define the ground instantiation of a program.

[4] Note that as opposed to Prolog we restrict ourselves to the function-free case

Definition 2.3. *An atom (or literal, resp.) is called* ground *if it does not contain variables. A rule is called* ground *if it only consists of ground literals. A program* Π *is called* ground *if none of its rules contains variables.*

The *Herbrand universe* HU_Π of a logic program Π is the set of all constants $c \in \sigma^{con}$ appearing in Π. The *Herbrand base* HB_Π is the set of all literals constructible from predicates $p \in \sigma^{pred}$ occurring in Π and the constants in HU_Π.

Definition 2.4. *The* ground instantiation $\Pi\downarrow$ *of a logic program* Π *finally is the set of all rules* r' *constructible from rules* $r \in \Pi$ *by replacing all variables with elements from* HU_Π.

Note that implemented systems like DLV and SMODELS provide efficient and intelligent methods for grounding a program, which do not necessarily instantiate over the whole Herbrand universe, but only generate those ground rules which are relevant wrt. to the semantics defined in the following.

2.1.4 Semantics

The semantics of EDLPs will now be defined in terms of ground programs. For a program with variables the semantics is defined by the semantics of its ground instantiation as defined above.

We say that a set of literals S *satisfies* a rule r if $Head(r) \cap S \neq \emptyset$ whenever $Body^+(r) \subseteq S$ and $Body^-(r) \cap S = \emptyset$.

The semantics of EDLPs is defined first for positive ground programs (i.e., for all rules $Body^-(r) = \emptyset$): Let $S \subseteq Lit(\Pi)$ be a set of consistent literals where we call a set of literals consistent whenever it does not contain a literal l and its complement $\neg.l$. An *answer set* for a positive program then is a minimal (relative to set inclusion) consistent set S satisfying all rules. [5]

To extend this definition to programs with negation as failure, we define the *reduct* Π^S (also often referred to as the *Gelfond-Lifschitz reduct*) of a program Π wrt. a set of literals S as the set of rules r'

$$h_1 \ \mathtt{v} \ \ldots \ \mathtt{v} \ h_l \ \mathtt{:-} \ b_1, \ \ldots, \ b_m$$

obtained from all rules r of the form (2.1) in Π such that $S \cap Body^-(r) = \emptyset$. We say that S is an answer set of Π if S is an answer set for Π^S. By $\mathcal{AS}(\Pi)$ we shall denote the set of all answer sets of program Π.

Example 2.1. For instance, the program Π_1:

$$p \ \mathtt{v} \ q. \quad \neg r \ \mathtt{:-} \ p.$$

has two answer sets: $S_1 = \{p, \neg q\}, S_2 = \{q\}$. Constraints can be used to "eliminate" unwanted answer sets in EDLPs, for instance, if we add the constraint

$$\mathtt{:-} \ p.$$

[5] We only consider *consistent answer sets*, while in [GL91, Lif96] also the (inconsistent) set HB_Π may be an answer set. Technically, we assume that negative classical literals $\neg a$ are viewed as new atoms $\neg a$, and constraints $\mathtt{:-}a, \neg a$ are implicitly added. This is the standard way how true negation is implemented in answer set systems like DLV or SMODELS.

to Π_1, then only S_2 remains a valid answer set. ◇

We will now define some syntactical restrictions which share some "good semantic properties" on EDLPs.

2.1.4.1 Head-Cycle-Free Logic Programs

In [BED94] an alternative definition for answer sets is given for so called *head-cycle free* EDLPs (HEDLPs). For that, we first have to define the dependency graph of an EDLP: The *dependency graph* of an EDLP Π is a directed graph where each literal occurring in Π is a node and there is an edge from l' to l if there is a rule in Π such that in $l \in Head(r)$ and $l' \in Body^+(r)$. We now say that Π is head-cycle free iff its dependency graph does not contain directed cycles that go through two literals occurring in the same rule head. For such HEDLPs Ben-Eliyahu and Dechter [BED94] showed the following:

Theorem 2.1 (cf. [BED94]). *Given a HEDLP Π, a consistent set $S \subseteq Lit(\Pi)$ is an answer set iff*

1. *S satisfies each rule in Π, and*
2. *there is a function $\phi : Lit(\Pi) \mapsto \mathbb{N}^+$ such that for each literal l in S there is a rule r in Π with*
   (a) $Body^+(r) \subseteq S$
   (b) $Body^-(r) \cap S = \emptyset$
   (c) $l \in Head(r)$
   (d) $S \cap (Head(r) \setminus \{l\}) = \emptyset$
   (e) $\phi(l') < \phi(l)$ *for each* $l' \in Body^+(r)$

The key essence of this theorem is that HEDLPs (in contrast to EDLPs in general) allow for a leveled evaluation of the logic program if we know the function ϕ. Later results by Babovich et.el. [BEL00] which generalize Fages' theorem [Fag94] resemble this theorem resulting in similar alternative definitions of answer sets for normal logic programs. Head-cycle-free disjunction in fact does not increase the expressive power compared with normal logic programs, as head-cycle-free negation can be shifted to the rule bodies by a semantically equivalent rewriting where any disjunctive rule:

h_1 v ... v h_l :-*Body*.

is substituted by l normal rules:

h_1 :- not h_2, not h_3 ... not h_l, *Body*.
h_2 :- not h_1, not h_3 ... not h_l, *Body*.
⋮
h_l :- not h_1, not h_2 ... not h_{l-1}, *Body*.

For programs with head-cycles, this rewriting is not possible as easily verified by the following example:

Example 2.2. Let Π_2 be the following simple DLP:

p:-q. q:-p. p v q.

Obviously, Π_2 has the single answer set $S = \{p, q\}$. On the other hand, when substituting the last rule with the pair of rules p:-not q. q:-not p. the resulting program has no answer sets at all. ◇

2.1.4.2 Stratified Logic Programs

An even stronger restriction is stratification. The concept of stratification was introduced for logic programs independently by Apt, Blair, and Walker [ABW88] and by van Gelder [van88]. Przymusinski generalized it to constraint-free DLPs [Prz88, Prz91].

We say that a constraint-free DLP Π is *stratified* iff there is a function $Strat : Lit(\Pi) \mapsto \mathbb{N}^+$ such that for every rule r of the form (2.1) there exists a $c \in \mathbb{N}$ with

1. $Strat(h) = c$ for all $h \in Head(r)$
2. $Strat(b) \leq c$ for all $b \in Body^+(r)$
3. $Strat(b) < c$ for all $b \in Body^-(r)$

It is well-known that such a stratification $Strat$ can efficiently be found, if existent. In particular, positive programs are always stratified. Note that stratification does not imply head-cycle freeness or vice versa. However, stratified programs also allow for an even more efficient evaluation. In case Π is free of integrity constraints, stratified programs always have at least one answer set. Note that EDLPs are not considered, since extended programs with classical negation always contain "implicit" integrity constraints $:\!-a, \neg a.$ for any complementary pair of literals (cf. Footnote 5).

2.1.5 Available Systems: Restrictions and Extensions

Among the available systems for computing answer sets of logic programs the two most successful over the past years have been DLV [ELM+98b, EFLP00, LPF+02] and SMODELS [Nie99, SNS02] which allow for efficient declarative problem solving.

2.1.5.1 DLV

The DLV system [6] is being developed for several years as joint work of the University of Calabria and Vienna University of Technology.

It is an efficient engine for computing answer sets accepting as core input language logic programs as defined above which fulfill the following *safety restriction* (cf. [Ull89]):

Definition 2.5. *A rule r of the form (2.1) is called* safe *if every variable X occurring in literals in $Head(r) \cup Body^-(r)$ also occurs in at least one literal $Body^+(r)$. A logic Program Π is* safe *if all of its rules are safe.*

Note that this restriction is only syntactical but does not really affect the expressive power of the language in any way. We refer for instance to [Pfe00] for a detailed discussion.

[6] <URL:http://www.dlvsystem.com>

Weak Constraints Furthermore, DLV extends the logic programs by so-called *weak constraints*, cf. [BLR97, BLR00]:

Definition 2.6. *A* weak constraint *is a construct*

$$:\sim\ b_1, \cdots, b_k,\ \texttt{not}\ b_{k+1}, \cdots,\ \texttt{not}\ b_m.\ [w : l] \tag{2.2}$$

where w (weight) and l (level) are integer constants or variables occurring in $b_1, \ldots, b_k$ and all b_i are classical literals. If l is not specified, it defaults to 1, and we can just write $[w :]$. $Body(r)$ is defined as for (2.1).

The level l intuitively allows to specify a priority layer after the colon, where 1 is the lowest priority.

The syntactical safety restriction from above is extended to weak constraints as follows: A weak constraint c is *safe* if in addition to the conditions above whenever w (or l, resp.) is a variable it has to occur in at least one literal $Body^+(r)$.

An *extended disjunctive logic program with weak constraints* (EDLP^w) is then a finite set of rules, constraints and weak constraints.

The *ground instantiation* for an EDLP^w is defined like in Definition 2.4 with the obvious extension to weak constraints. Furthermore, we impose another syntactical restriction on weak constraints related to safety mentioned above: A weak constraint c is only admissible, if all possible weights and levels are integers. Thus, if either w or l is a variable, then Π must guarantee that w, l can only be bound to integers. This restriction can also easily be checked (for instance during grounding) which is done by DLV.

The answer sets of an EDLP^w Π without weak constraints are defined as above. The answer sets of a program Π with weak constraints are defined by selection of so called *optimal answer sets* from the answer sets S of the weak-constraint free part Π' of Π (referred to as *candidate answer sets*).

Again, we will define the semantics of optimal answer sets in terms of the ground instantiation of a program. We will first define the semantics without levels (i.e. for all weak constraints c of form (2.2) in Π we uniquely assume $l = 1$).

A weak constraint c of the form (2.2) is *violated*, if it is satisfied with respect to the candidate answer set S, i.e., $\{b_1, \cdots, b_k\} \subseteq S$ and $\{b_{k+1}, \cdots, b_m\} \cap M = \emptyset$; we then define the *violation cost* $v_S(c)$ of c wrt. S as:

$$v_S(c) = \begin{cases} w & \text{if } c \text{ of form (2.2) is violated wrt. } S. \\ 0 & \text{otherwise.} \end{cases}$$

The *1-cost* of S, denoted $\mathsf{cost}_{1,\Pi}(S)$, is then

$$\mathsf{cost}_{1,\Pi}(S) = \sum_{c \in \Pi} v_S(c)$$

i.e., the sum of violation costs of weak constraints in Π wrt. S. An answer set M of Π is now selected (called an *optimal answer set*), if $\mathsf{cost}_1, \Pi(M)$ is minimal over all candidate answer sets of Π.

In case that cost levels $l > 1$ occur in the ground instantiation of Π, violation costs can intuitively have different priorities: Violation costs of a candidate answer set S are then

summed up and ordered *per level*, where violations at higher level have greater priority. The cost $\mathsf{cost}_\Pi(S)$ of S can then be defined as the tuple

$$\mathsf{cost}_\Pi(S) = \langle \mathsf{cost}_{l,\Pi}(S), \mathsf{cost}_{l-1,\Pi}(S), \ldots, \mathsf{cost}_{1,\Pi}(S) \rangle$$

where the *i-cost* of S, $\mathsf{cost}_{i,\Pi}(S)(i \in \{1, \ldots, l\})$, is defined in analogy to the definition of the *1-cost* of S above and l is the maximal cost level occurring in Π.

We then select the answer sets with the *lexicographic smallest* such tuple as the optimal ones. Obviously, for programs without levels $\mathsf{cost}_{1,\Pi}$ and $\mathsf{cost}_\Pi(S)$ coincide.

Example 2.3. Let us reconsider the original program Π_1 from Example 2.1. If we add the weak constraints

$$:\sim p.[1:1] \quad :\sim q.[3:1]$$

then $\mathsf{cost}_{\Pi_1}(S_1) = \langle 1 \rangle$ and $\mathsf{cost}_{\Pi_1}(S_2) = \langle 3 \rangle$. Thus, the single optimal answer set is S_1. $\diamond$

In the following, if there are costs of level 1 only in a program Π we write short $\mathsf{cost}_\Pi(S) = c$ instead of $\mathsf{cost}_\Pi(S) = \langle c \rangle$ when clear from the context.

Built-In Predicates The built-in predicates "`A < B`", "`A <= B`", "`A > B`", "`A >= B`", "`A != B`" with the obvious meaning of less-than, less-or-equal, greater-than, greater-or-equal and inequality for strings and numbers can be used in the positive bodies of DLV rules and constraints.

DLV currently does not support full arithmetics but supports some built-in predicates, which can be used to "emulate" range restricted integer arithmetics: The arithmetic built-ins "`A = B + C`" and "`A = B * C`" which stand for integer addition and multiplication, and the predicate "`#int(X)`" which enumerates all integers (up to a user-defined limit).

Furthermore, borrowing from database query languages, DLV has recently been extended by aggregate functions [DFI+03], such as `#count`, `#sum`, `#min`, and `#max`.

2.1.5.2 SMODELS and GNT

SMODELS [Nie99, SNS02][7] allows for the computation of answer sets for normal logic programs. However, there is an extended prototype version for the evaluation of disjunctive logic programs as well, called GNT [JNSY00][8].

Syntactically, SMODELS imposes an even stronger restriction than rule safety in DLV by demanding that any variable in a rule of the form (2.1) is bounded to a so-called domain predicate $d \in Body^+(r)$ which is, intuitively, a predicate which is defined only positively (cf. [Nie99] for details). Again, this restriction does not affect the expressive power of the language itself, but in some cases the weaker safety restriction of DLV allows for more concise problem encodings.

SMODELS is also capable of optimal model computation. However, the syntactic/semantic concept here is a little bit different from weak constraints in DLV: SMODELS supports another

[7] <URL:http://www.tcs.hut.fi/Software/smodels/>
[8] <URL:http://www.tcs.hut.fi/Software/gnt/>

extension to pure answer set programming allowing to minimize over sets of predicates (cf. [SNS02] for details) by adding statements *min* of the form:

$$\texttt{minimize}\{b_1 = w_1,\ \ldots,\ b_m = w_m,\ \texttt{not}\ b_{m+1} = w_{m+1},\ \ldots\ \texttt{not}\ b_n = w_n\}.$$

where $b_1, \ldots b_n$ are ground literals and $w_1, \ldots w_n$ are constants. Here, similarly to weak constraints, an answer set S is considered to be optimal if

$$\mathsf{cost}_\Pi(S) = \sum\{w|((b_i = w) \in min \wedge b_i \in S) \vee (\texttt{not}\ b_i = w) \in min \wedge b_i \notin S)\}$$

is minimal. If there are more than one `minimize` statements in SMODELS they are considered in fixed order, the last one being the strongest, similar to levels of `DLV` weak constraints, but missing full declarativity in some sense (since rule order has a semantic impact here).

For `minimize` statements with variables, SMODELS offers the following shorter notation:

$$\begin{aligned}\texttt{minimize}[&a_1(\vec{X}_1) : b(\vec{Y}_1) = C_1, \ldots a_m(\vec{X}_m) : b(\vec{Y}_m) = C_m,\\ &\texttt{not}\ a_{m+1}(\vec{X}_{m+1}) : b(\vec{Y}_{m+1}) = C_{m+1}, \ldots \texttt{not}\ a_n(\vec{X}_n) : b(\vec{Y}_n) = C_n].\end{aligned}$$

where $\vec{X}_i$, $\vec{Y}_i$ are lists of variables or constants, and all the variables in $\vec{X}_i$ have to occur in $\vec{Y}_i$. C_i is either a variable from $\vec{Y}_i$ or a constant, and b_i is a domain predicate, for $i \in \{1, \ldots, n\}$. This statement informally complies with

$$\texttt{minimize}\{a_1(\vec{X}_1) = C_1 \mid b_1(\vec{Y}_1)\} \cup \ldots \cup \{\texttt{not}\ a_n(\vec{X}_n) = C_n \mid b_n(\vec{Y}_n)\}.$$

which however is not a valid notation of SMODELS.

Observe that during model computation, the behavior of SMODELS is not the one that would come to mind first: Instead of computing only optimal answer sets it first computes an arbitrary model and then incrementally only returns "better" answer sets, such that the last answer set found by SMODELS is optimal.

As an additional feature SMODELS provides a dual `maximize` statement as well with the obvious semantics.

Similar to `DLV`, SMODELS allows for a restricted form of integer arithmetics and lexicographic comparison predicates.

2.2 Complexity

We will now review the most important problem classes for the computational complexity of the problems addressed in the course of this work. Furthermore, we will review some results on the computational complexity of Answer Set Programming.

2.2.1 Complexity Classes

We assume that the reader is familiar with the concept of Turing Machines and basic notions of complexity theory, such as problem reductions and completeness; see e.g. [Pap94] and references therein. We recall P, resp. NP, is the class of decision problems (i.e., problems

where the answer is "yes" or "no") computable on a deterministic, resp. nondeterministic, Turing Machine in polynomial time. Further, PSPACE is the class of problems computable on deterministic Turing Machines with polynomial storage space.

The classes Σ_k^P (resp. Π_k^P, Δ_k^P,), $k \geq 0$ of the so called Polynomial Hierarchy PH = $\bigcup_{k\geq 0} \Sigma_k^P$ are defined by $\Sigma_0^P = \Pi_0^P = \Delta_0^P = \mathrm{P}$ and $\Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}$ (resp. $\Pi_k^P = \text{co-}\Sigma_k^P$, , $\Delta_k^P = \mathrm{P}^{\Sigma_{k-1}^P}$), for $k \geq 1$. The latter model nondeterministic polynomial-time computation with an oracle for problems in Σ_{k-1}^P. Here, co− stands for the class of complementary problems. In particular, $\Sigma_1^P = \mathrm{NP}$, $\Pi_1^P = \text{co-NP}$ and $\Delta_2^P = \mathrm{P}^{\mathrm{NP}}$.

Furthermore, $\mathrm{D}^P = \{L \cap L' \mid L \in \mathrm{NP}, L' \in \text{co-NP}\}$ is the logical "conjunction" of NP and co-NP.[9] Finally, NEXPTIME and NEXPSPACE denote the class of problems decidable by nondeterministic Turing machines in exponential time, resp. space.

We recall that NP $\subseteq \mathrm{D}^P \subseteq$ PH $\subseteq$ PSPACE=NPSPACE $\subseteq$ NEXPTIME holds, where NPSPACE is the nondeterministic analog of PSPACE. It is generally believed that these inclusions are strict, and that PH is a true hierarchy of problems with increasing difficulty. Note that NEXPTIME-complete problems are *provably intractable*, i.e., exponential lower bounds can be proved, while no such proofs for problems in PH or PSPACE are known today.

While many interesting problems are decision problems, computing answer sets or plans are *search problems*, where for each problem instance I a (possibly empty) finite set $S(I)$ of solutions exists. To solve such a problem, a (possibly nondeterministic) algorithm must compute the alternative solutions from this set in its computation branches, if $S(I)$ is not empty. More precisely, search problems are solved by transducers, i.e., Turing machines equipped with an output tape. If the machine halts in an accepting state, then the content of the output tape is the result of the computation. Observe that a nondeterministic machine computes a (partial) multi-valued function.

As an analog to NP, the class NPMV contains those search problems where $S(I)$ can be computed by a nondeterministic Turing machine in polynomial time; for a precise definition, see [Sel94]. In analogy to Σ_{i+1}^P, by $\Sigma_{i+1}^P\mathrm{MV} = \mathrm{NPMV}^{\Sigma_i^P}$, $i \geq 0$, we denote the generalization of NPMV where the machine has access to a Σ_i^P oracle.

Analogs to the classes P and Δ_{i+1}^P, $i \geq 0$, are given by the classes FP and $\mathrm{F}\Delta_{i+1}^P$, $i \geq 0$, which contain the partial single-valued functions (that is, $|S(I)| \leq 1$ for each problem instance I) computable in polynomial time possibly using a Σ_i^P oracle. We say, abusing terminology, that a search problem A is in FP (or $\mathrm{F}\Delta_{i+1}^P$), if there is a partial (single-valued) function $f \in \mathrm{FP}$ (or $f \in \mathrm{F}\Delta_{i+1}^P$) such that $f(I) \in S(I)$ and $f(I)$ is undefined iff $S(I) = \emptyset$. For example, computing a satisfying assignment for a propositional CNF (FSAT) and computing an optimal tour in the Traveling Salesperson Problem (TSP) are in $\mathrm{F}\Delta_2^P$ under this view, cf. [Pap94].

A partial function f is polynomial-time reducible to another partial function g, if there are polynomial-time computable functions h_1 and h_2 such that $f(I) = h_2(I, g(h_1(I)))$ for all I and $g(h_1(I))$ is defined whenever $f(I)$ is defined. Hardness and completeness are defined as usual.

[9] Note that D^P is *not* NP $\cap$ co-NP (cf. [Pap94]).

2.2.2 Complexity of Logic Programming

We will now consider the following problems: Given a logic program Π, decide whether Π has a model under the Answer Set Semantics.

We restrict ourselves to finite propositional, i.e. ground, (function-free) EDLPs as defined above. Non-ground programs are not considered as grounding might already be exponential and deciding answer set existence thus becomes provably intractable even for simple positive normal programs (cf. [Imm87, Var82, DEGV01]). Furthermore, note that when allowing function symbols most of the problems outlined in this section become undecidable in general which is basically explained by the undecidability of first-order logic.

Theorem 2.2 (cf. [EGM97, DEGV01]). *Deciding whether a propositional EDLP has an answer set is Σ_2^P-complete. Computing such an answer set is by our notation Σ_2^PMV-complete.*

For clearness and as we will occasionally refer to it later on, we review the proof idea:

Proof. Membership: Given a program Π, an answer set S can be guessed and checked in polynomial time by an NP oracle: In particular, the reduct Π^S can clearly be computed in polynomial time. Since Π^S is a positive program, its answer sets coincide with its minimal models. Testing whether S is a minimal model is in co-NP (cf. [Cad92]) and therefore decidable in polynomial time by a single call to an NP oracle.

Hardness: For the hardness proof we will review an encoding of deciding the satisfiability of a Quantified Boolean Formula (QBF)

$$F = \exists x_1 \dots \exists x_m \forall y_1 \dots \forall y_n \Phi$$

with one quantifier alternation in an answer set program, which is a well known reference problem hard for the class Σ_2^P. Here, $\Phi = c_1 \vee \cdots \vee c_k$ is a propositional formula over $x_1, \dots x_m, y_1, \dots, y_n$ in disjunctive normal form, i.e. each $c_i = a_{i,1} \wedge \cdots \wedge a_{i,l_i}$ and $|a_{i,j}| \in \{x_1, \dots, x_m, y_1, \dots, y_n\}$. Satisfiability is here defined as the existence of an assignment to the variables $x_1, \dots x_m$ which witness that F evaluates to true.

We will now present an encoding of this formula as an answer set program Π_{QBF} such that Π_{QBF} has an answer set if and only if F is satisfiable:

x_1 v nx_1. ...x_m v nx_m.
y_1 v ny_1. ...y_n v ny_n.
sat :- $a_{1,1}, \dots, a_{1,l_1}$.
⋮
sat :- $a_{k,1}, \dots, a_{k,l_k}$.
y_1 :- sat. ny_1 :- sat. ...y_n :- sat. ny_n :- sat.
:- not sat.

This encoding is maybe not intuitive at first sight, but in principle can be explained as follows: For any assignment guessed for $x_1, \dots x_m$, `sat` will **not** be derived if there is a bad assignment for $y_1, \dots, y_n$ such that all clauses are unsatisfied. However, since all answer sets not containing `sat` are invalidated by the last constraint, only those assignments for $x_1, \dots x_m$ "survive" which do not allow for such a bad assignment for $y_1, \dots y_n$. This can be argued by minimality of answer sets together with rules in the one but last line, which "saturate" any good assignment for $y_1, \dots y_n$ to an answer set uniquely determined

by $x_1, \ldots x_m$. Note that, this encoding does not only represent the satisfiability problem, but moreover the answer sets of Π_{QBF} uniquely encode the valid assignments for variables $x_1, \ldots x_m$, which proofs hardness for Σ_2^P and Σ_2^PMV, respectively. For the details of this encoding we refer to [EG95]. □

The main impact of this result in the context of the present work is that the computational power of answer set solvers such as DLV and GNT which support full disjunctive logic programming is indeed higher than solvers for propositional Satisfiability (SAT) (unless the PH collapses): In the course of this work we will show that using Answer Set Programming we can encode and solve hard problems from the area of planning not expressible as a simple propositional logic formula. As a "side-product" we will also present a method for encoding hard problems on the second level of the PH which, compared to the ad hoc QBF encoding above is more generally applicable.

Note that the complexity boils down to lower complexity classes as soon as we impose some of the syntactical restrictions mentioned above:

Proposition 2.3. *(cf. [BED94, Fag94]) For head-cycle free (resp. normal) logic programs deciding answer set existence is* NP*-complete.*

The essence of this result is that head-cycle free and normal logic programs can intuitively be evaluated by guessing an order of rule evaluation as reflected in Theorem 2.1. Moreover, it states that SMODELS without its disjunctive extension GNT, i.e. answer set solvers which only accept normal or head-cycle-free logic programs cover the same class of problems as SAT solvers.

Furthermore, for *stratified* (especially positive) DLPs the answer sets correspond to the minimal models of a program, i.e. answer set existence is trivial (cf. [Prz88, Prz91]). For non-disjunctive programs this model even is unique and by well-known results computable in polynomial time.

Finally, as for *optimal answer sets* wrt. to weak constraints in DLV, we know from [BLR00] that deciding whether a query q is true in some optimal answer set of an EDLP^w Π is Δ_3^P-complete and Δ_2^P-complete for head-cycle-free programs. The respective class for computing such an optimal answer set is FΔ_3^P, and FΔ_2^P for head-cycle-free programs. These results equally apply to minimization in SMODELS with minor adaptions.

2.3 Planning – An Overview

In this section, we will give a brief informal overview of the field of planning and existing approaches for automated plan generation.

2.3.1 Terminology

When talking about planning, we first have to review the problems and the common terminology used in this field. Formal definitions wrt. our formalism will be introduced in Chapter 3.

The *classical planning* problem consists of the following task: Given a *state of the world*, several *actions*, their preconditions and (deterministic) effects, find a sequence of actions (viz. a plan) to achieve a state in which some certain *goal* holds.

2.3.1.1 States and Fluents

States are described by the truth values of state variables, so called *fluents*. We distinguish between so called *world states* and *knowledge states*: Assume that the set of fluents $F = \{f_1, f_2, \ldots, f_n\}$ describes the relevant (from a subjective point of view) clipping of the world. Then, the current state of the world wrt. this clipping can be defined as a function $s : F \rightarrow \{true, false\}$ or in other words as a complete set of literals which contains either f or $\neg f$ for any $f \in F$. From an agent's point of view, states can also be seen less restricted as only a partial function s', resp. any consistent set of fluent literals, where for a particular fluent neither f nor $\neg f$ must hold, i.e. s' consists only of the subset of s which is *known*. We denote s' as *state of knowledge* as opposed to the actual state of the world.

Note that this view of the epistemic state of an agent differs from other approaches where incomplete knowledge states are defined as the set of all possible worlds an agent might be in (for instance, cf. [SB01, BG00, BCPT01]). Such sets of (compatible) world states are also often referred to as *belief states* in the planning literature(e.g. [BG00, BCPT01]. However, as we will see, both views (knowledge states and belief states) can be modeled in the formalism proposed in the present work.

Remark 2.1. *We remark here that the above-mentioned terminology for knowledge states and belief states is disputable. For example, Son and Baral [SB01] use the term "states of knowledge" in their formalism describing the set of reachable worlds in a Kripke structure. As mentioned above, this rather amounts to what we call "belief states" in our terminology. We adopted the planning community terminology in this work. An in-depth discussion of the terms "knowledge" and "belief" can be found e.g. in [Hin62].*

A useful generalization of fluents is to allow not only Boolean fluents which are either true or false in each state but also *multi-valued fluents* [GLLT01] which take a certain value of a specific (finite) *domain* in each state. A state can then be seen as a set of functions which assigns to each fluent f a value of its domain D_f where Boolean fluents have the domain $\{true, false\}$. Although a multi-valued fluent f with finite domain $D_f = \{d_1, \ldots, d_n\}$ can be "emulated" by a set of Boolean fluents $f_{d_1}, \ldots f_{d_n}$ plus constraints which prohibit concurrent truth of two distinct f_{d_i}, f_{d_j}, multi-valued fluents often allow for a more concise representation (cf. Section 7.1.3).

2.3.1.2 State Transitions and Plans

By executing actions, the state of the world (or state of knowledge, resp.) changes. In the present work we tacitly assume that transitions between actions are "step-by-step", i.e. defined by the current state, a set of (concurrently) executed actions and the possible subsequent states where as a simplifying but commonly used assumption we take that all actions have unique duration and that all effects materialize in the successor state (i.e., we use a discrete notation of time). Given these assumptions, a *plan* is a sequence of sets of

actions, and executing the plan in some state corresponds to possibly various "trajectories", i.e. sequences of states.

For describing general reasoning about actions and such transitions, we have to describe the actions and their effects in an appropriate formalism. Here, among others, the following basic considerations play a role:

- How to describe executability of actions (Qualifications)
- How to describe effects of actions
- How to describe indirect effects, i.e. interdependencies of fluents (Ramifications)
- How to describe which fluents remain unchanged over a transition (known as the Frame Problem [MH69, RN95])

Action languages as introduced in Section 2.4 provide an expressive tool to describe these cohesions between fluents and actions. In the course of this work we will discuss the solution and formalization of all these problems in our particular formalism.

2.3.1.3 Planning Problems

We now generalize the classical definition of a planning problem a bit: For a *planning problem*, a possibly incompletely defined (set of) *initial state(s)* is given from which the goal shall be reached. Here, we define the *goal* as a set of fluent literals. As opposed to the classical definition, we no longer necessarily assume a single, completely defined initial state and do no longer restrict ourselves to sequences of deterministic actions. Rather, actions might occur in parallel, might have nondeterministic effects, etc.

In general, solutions for such a planning problem, i.e. plans, consist of a strategy for executing actions reaching a state where the goals (are known to) hold. In the following, we discuss different notions of such plans.

2.3.1.4 Classical vs. Non-classical Planning

In classical planning restrictive and often unrealistic assumptions are made, such as full knowledge about the initial state of the world; plans are simply sequences of fully deterministic atomic actions. Several proposals for extensions for planning under incomplete knowledge or possible nondeterministic action effects have been made:

Conformant planning has been introduced in [GB96], describing the problem to find a plan which works in any initial situation even under incomplete knowledge or when nondeterministic effects of actions are allowed. There are several systems available dealing with conformant planning, such as CMBP [CR00], CPLAN [FG00, Giu00, CGT02] (a conformant planner based on CCALC [MT97, MT98]), GPT [BG00], and SGP [WAS98]. Finally, the only Answer Set Programming based planning system available DLV^K [EFL$^+$03a, EFL$^+$03c] which will be presented in this thesis, falls in this class. Conformant plans, however, do not always exist.

On the other hand, in *conditional planning* every contingency in a plan is considered and a branching plan(-tree) is constructed. There are several implemented planners capable of computing such plans, such as CNLP [PS92], CBURIDAN [DHW94], Cassandra [PC96], or

recently MBP [BCPT01]. Despite the success of logic-based planning approaches for classical planning (without uncertainty), or conformant planning (under uncertainty or incomplete information) to our knowledge, there is no conditional planner based on logic programming.

Furthermore, there have been proposals to generate a more general *reactive* behavior than a plan in the classical sense based on *Markov Decision Processes* [BDH99] or by so called *Universal Plans* [Sch87, Gin89] where plans are not viewed as sequences or conditional trees of actions but rather describe reactive patterns for any possible situation. Also, for such notions of plans no logic programming implementations are available.

In the present work, for the nondeterministic case we will focus on conformant planning.

2.3.2 Approaches

The most successful techniques for *classical planning*, i.e. planning without uncertainty under complete knowledge in closed environments are based on Graphplan [BF97] and descendants [SW98] and Planning via Heuristic Search [BG01]. Whereas all these approaches try to find plans as (partially ordered or totally ordered) courses of actions, the Hierarchical Task Network (HTN) [EHN94] planning paradigm views a plan as a hierarchical network of compound tasks. HTN has recently becomes more important and some of the successful planners are based on hierarchical plan decomposition, e.g. SHOP [NCLMA99].

In recent years, several logic-based approaches to classical planning have been proposed such as planning based on propositional SAT [KS92, FG00, Giu00] and QBF-SAT [Rin99a] solvers, based on Symbolic Model Checking techniques [CGGT97, CRT98, CR99, EH00] or based on declarative logic programming [SZ95, DNK97, Lif99b, EFL$^+$03a, LRS01, DKN02a]. In this work we will especially focus on the latter.

2.4 Planning Languages

From the knowledge representation perspective, expressive *planning languages* such as STRIPS [FN71] and extensions [Wel94], PDDL [GHK$^+$98] or action languages such as $\mathcal{A}$ [GL93], $\mathcal{A}_R$ [GKL97], $\mathcal{A}_K$ [SB01], $\mathcal{C}$ [GL98a] and its successor $\mathcal{C}+$ [GLL$^+$03], or finally $\mathcal{K}$ [EFL$^+$03b] (which is subject of the present work) have been developed, in order to be able to describe involved planning problems in a declarative way. Most implementations of the above-mentioned approaches accept problem descriptions in such a high-level language as input.

In the following we will give a (probably incomplete) overview of some of the most common languages and their features. We assume basic familiarity with the concepts of first order logic.

2.4.1 Situation Calculus

One of the first knowledge representation approaches for formalizing actions and state transitions is McCarthy's *Situation Calculus* [MH69]. It is basically a method of describing change in first-order logic conceiving the world as consisting of a sequence of situations. Situation Calculus uses logic connectives and the distinct function $res(Action, Sitiation)$ to describe the situation resulting from executing an action. Every predicate describing a fluent is given an extra situation argument.

Example 2.4. As a simple example for such a situation calculus formula we take the formalization of the effect of moving a block in the Blocks World (cf. Section 6.2):

$$\forall b, l, s \; clear(b, s) \wedge clear(l, s) \Rightarrow on(b, l, res(move(b, l), s))$$

This Situation Calculus formula intuitively states that whenever a *clear* block b is *moved* to a *clear* location l in situation s then b is *on* l in the resulting situation $res(move(b, l))$. ◇

One major drawback of this formalization is the necessity of a number of so-called *frame axions*, i.e. all facts which remain unaffected have to be explicitly stated, for instance,

$$\forall a, b, l, s \; clear(l, s) \wedge a \neq move(b, l) \Rightarrow clear(l, res(a, s))$$

Depending on the number of fluents, many of those frame axioms might be necessary even in a closed, fully known environment with few actions.

2.4.2 STRIPS and Descendants

A later approach for formalizing actions is the operator concept of STRIPS [FN71]. Here, an action (often called *operator*) a is defined by lists of preconditions $pc(a)$, negative (delete-list) effects $del(a)$, and positive (add-list) effects $add(a)$, where these lists are simple conjunctions of fluents.

Example 2.5. The example from above could be modeled in STRIPS as follows (where we use a LISP-like simplified PDDL notation):

```
(:action move
 (:parameters ?block ?from ?to)
 (:preconditions (and (clear ?block) (clear ?to)
                      (on ?block ?from)))
 (:effect        (and (clear ?from) (on ?block ?to)
                      (not (clear ?to)))) )
```

◇

The semantics is clearly defined with implicit frame axioms for all fluents by the following definition of transitions between world states S_i and S_{i+1} on execution of operator a:

$$S_{i+1} = trans_{\text{STRIPS}}(S_i, a) = \begin{cases} (S_i \setminus del(a)) \cup add(a), & \text{if } pc(a) \subseteq S_i \text{ and} \\ & add(a) \cup del(a) = \emptyset \\ \text{undefined}, & \text{otherwise} \end{cases}$$

However, the expressive power of pure STRIPS is quite restricted by that only conjunctions of preconditions and effects are allowed. Alternative qualifications for an action have to be modeled by different operators. Furthermore, only sequential execution of actions is assumed, i.e., no parallel execution of operators is allowed. Effects can only be modeled as consequences of actions, but there are no means of describing ramifications, i.e., state axioms or indirect effects.

Nevertheless, STRIPS and its descendants ADL [Ped89] and later PDDL [GHK+98, FL03], etc. have gained great popularity in the planning community. These descendants also include extensions to express conditional effects, state axioms, durative actions, etc. Last but not least, PDDL has become a quasi standard, being the official language used in the International Planning Competition (IPC) held every second year in the course of the "International Conference on Automated Planning and Scheduling (ICAPS)" (formally, "Artificial Intelligence Planning and Scheduling Conference (AIPS)").

A major drawback of all these languages is their strict operator-based syntax which is far from natural language descriptions of transitions between states. Furthermore, all these languages are restricted to complete knowledge on the initial state and deterministic effects of actions. Extensions of these classical planning languages by means of expressing nondeterminism or incomplete knowledge are manyfold (e.g. by the language NPDDL [BCLP03]), but no proposal has made its way to the PDDL standard yet. Since providing a standard, widely accepted language for the IPC is the main focus of PDDL, extensions are accepted only very cautiously.

2.4.3 Action Language $\mathcal{A}$ and Descendants

In the planning community, the development of formal languages to describe planning problems is driven by a clear focus on special-purpose algorithms and systems, where ease of structural analysis of the problem description at hand is a main issue. On the other hand, expressive languages for formalizing actions and change in a more general context have emerged from the field of knowledge representation.

In the course of this thesis we will concentrate on such high-level action languages which allow for a quasi natural-language like formalization of actions and change, the first of which introduced was the language $\mathcal{A}$ [GL93]. To give a flavor of such action languages, we will exemplify the most important ones and briefly discuss their semantic features in this section.

Action Language $\mathcal{A}$

Action language $\mathcal{A}$ represents from the viewpoint of expressiveness essentially the propositional fragment of Pednault's ADL, i.e. STRIPS enriched with conditional effects.

Effects and preconditions are expressed by causation rules and executability conditions of the following form:

a `causes` l `if` F`.`
`executable` a `if` F`.`

where a is an action name, l is a fluent literal, and F is a conjunction of fluent literals. An action description D consists of a set of such propositions.

We then define the positive and negative effects of of an action a as follows: $eff(a, S)$ is the set of all l such that D contains a statement a `causes` l `if` F. and $F \subseteq S$. Given a world state S_i and an action a, the transition function can then be described as follows:

$$trans_{\mathcal{A}}(S_i, a) = \begin{cases} (S_i \setminus \neg.eff(a, S_i)) \cup eff(a, S_i), & \text{if } D \text{ contains a statement} \\ & \texttt{executable } a \texttt{ if } F. \\ & \text{such that } F \subseteq S_i \\ & \text{and } eff(a, S_i) \text{ is consistent} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Furthermore, $\mathcal{A}$ offers the possibility to include "observations" of the form

f `after` $a_1; \ldots; a_n$

where $a_1; \ldots; a_n$ is a sequence of actions. These observations are used to express goal conditions when verifying a plan or initial observations ($n = 0$).

Example 2.6. A clipping of the Blocks World example from above could alternatively be described in $\mathcal{A}$ as follows:

`executable` $\texttt{move}_{b,a}$ `if` $\texttt{clear}_a$`,` $\texttt{clear}_b$`.`
$\texttt{move}_{b,a}$ `causes` $\texttt{clear}_c$ `if` $\texttt{on}_{b,c}$`.`

Since $\mathcal{A}$ only allows for propositional fluents and actions, we have chosen the sample moving of a block a to a block b with a conditional effect: Whenever block b is on block c, block c is clear after b has been moved away to a. ◇

Extensions of $\mathcal{A}$

Language $\mathcal{A}_R$ A further step in the development of action languages was the language $\mathcal{A}_R$ [GKL97]. This language extends $\mathcal{A}$ by allowing to model indirect effects by introducing laws of the form

`always` F`.`

where F is a propositional formula, which has to be valid in any state. Moreover, $\mathcal{A}_R$ allows for arbitrary propositional formulae for C and F in statements of the form

a `causes` C `if` F`.`

A further enhancement of $\mathcal{A}_R$ is the capability of modeling nondeterministic actions by statements of the form

a `possibly changes` l `if` F`.`

Language $\mathcal{B}$ Alternatively to modeling state axioms by means of `always` in $\mathcal{A}_R$, another extension, the language $\mathcal{B}$ [GL98a] suggests "static laws" of the form:

l `if` F`.`

As opposed to `always` in $\mathcal{A}_R$ the semantics of these static laws informally reflect the principle of "minimal change" where a successor state is defined as a minimal consistent set of literals such that all these static rules hold. In particular, static laws do *not* correspond to classical implication. The difference becomes obvious by the following example known as Lin's Suitcase [Lin95]:

Example 2.7. Assume we have a spring suitcase with two latches. Flipping a latch turns its position to "up", and as an indirect effect the suitcase opens as soon as both latches are up. This can be modeled by the following $\mathcal{B}$ action description:

$flip_1$ `executable`.
$flip_2$ `executable`.
$flip_1$ `causes` up_1.
$flip_2$ `causes` up_2.
$open$ `if` up_1, up_2.

where $trans_{\mathcal{B}}(\{up_1, \neg up_2, \neg open\}, flip_2) = \{up_1, up_2, open\}$ as intuitively expected in language $\mathcal{B}$. Note that applying the original transition function $trans_{\mathcal{A}}$ for language $\mathcal{A}$ (ignoring the final static law) we would gain state $\{up_1, up_2, \neg open\}$. When replacing the final static law by the $\mathcal{A}_R$ statement

`always` $up_1 \wedge up_2 \Rightarrow open$.

(which is logically equivalent to `always` $up_1 \wedge \neg open \Rightarrow \neg up_2$.) we would then end up with inconsistency instead of the intuitive transition. ◇

Language $\mathcal{A}_K$ Another remarkable extension of action languages $\mathcal{A}_R$ and $\mathcal{A}$ was proposed by Son and Baral with language $\mathcal{A}_K$ [SB01]. This language is intended to formalize sensing actions. For this purpose $\mathcal{A}_K$ provides additional knowledge determining propositions of form a `determines` f, which intuitively means that after executing action a, the value of fluent f is known. The concept of knowledge differs from what we referred to as knowledge states in Section 2.3 which will be further discussed below.

Action Language $\mathcal{C}$

The most recent and evolved languages in the line of action languages from the Texas Action Group (TAG) which is also responsible for language $\mathcal{A}$,$\mathcal{B}$ and $\mathcal{A}_R$ are the languages $\mathcal{C}$ [GL98b] and its extension $\mathcal{C}+$ [GLL+03]. $\mathcal{C}$ is similar to $\mathcal{B}$ in that it also distinguishes between static and dynamic laws. It is in some ways more expressive than $\mathcal{B}$ and $\mathcal{A}_R$ though strictly speaking $\mathcal{C}$ is not a superset of either $\mathcal{B}$ or $\mathcal{A}_R$.

$\mathcal{C}$ action descriptions consist of a set of causation laws c of the form

$$\texttt{caused}\ F\ \texttt{if}\ G\ \texttt{after}\ H. \tag{2.3}$$

where the `after`-part is optional: c is called *static* if it has no `after`-part and *dynamic* otherwise. These rules are more flexible than the abovementioned approaches in that F and G are arbitrary propositional formulae over fluent literals and H is a propositional formula over fluent and action literals. Furthermore, constraints and qualifications can be expressed

via $F = f \wedge \neg f$ in $\mathcal{C}$ which is written as "`caused` $\bot$ `if` G `after` H. These rules encode inconsistency similar to constraints in logic programming as mentioned in Section 2.1.

A $\mathcal{C}$ action description D consists of a set of static and dynamic causation laws. The semantics of D is given by the following definition of *causally explained* transitions:

A transition ⟨s,a,s'⟩ is causally explained according to D if its resulting state s' is the only interpretation that satisfies all rules caused in this transition, where a formula F is caused if it is

- the head of a static law (2.3) from D such that $s' \models G$ or
- the head of a dynamic law (2.3) from D such that $s' \models G$ and $s \cup a \models H$

Note that this definition allows for nondeterministic actions, i.e., valid transitions $\langle s, a, s' \rangle$, $\langle s, a, s'' \rangle$ with $s' \neq s''$ are possible. The definition of causally explained transitions is closely related to causal theories as defined by McCain and Turner [MT97] and the underlying concept of causal explanation [Lif97].

Moreover, $\mathcal{C}$ offers a bunch of useful macros, some of which shall be explained: Remarkably, inertia of a fluent (i.e., that a fluent remains unchanged) has to be explicitly encoded in $\mathcal{C}$ which means that frame axioms are not implicit like in the previously discussed approaches. However, inertia can be easily expressed in $\mathcal{C}$ by the following macro

$$\texttt{inertial } F. \quad \Leftrightarrow \quad \texttt{caused } F \texttt{ if } F \texttt{ after } F.$$

which offers a convenient method for encoding frame axioms in $\mathcal{C}$, as opposed to encoding frame axioms explicitly in the situation calculus.

A further macro in $\mathcal{C}$ for modeling qualifications of actions is

$$\texttt{nonexecutable } A \texttt{ if } G. \quad \Leftrightarrow \quad \texttt{caused } \bot \texttt{ after } A \wedge G.$$

In the following, when referring to $\mathcal{C}$ we sometimes refer to a restricted fragment of the language: We call an action description in $\mathcal{C}$ *definite* if in laws of the form (2.3) only $\bot$ or fluent literals are allowed for F.

A recent extension of $\mathcal{C}$ called $\mathcal{C}+$ allows for multi-valued, additive fluents which can be used for example in order to encode resources [GLLT01, GLL+03] and allows for a more compact representation of several practical problems.

Action Language $\mathcal{K}^c$

In the course of this work, we will introduce the action language $\mathcal{K}$ and its extension $\mathcal{K}^c$ which will be presented in Chapter 3 in detail. $\mathcal{K}^c$ is similar to the action languages discussed in this section by adopting some of their features, but, as we will see, semantically closer to Answer Set Programming. The aim of the development of a new language was to adopt useful concepts of existing high-level action languages together with some of the features of Answer Set Programming in terms of syntactic features and semantics.

Chapter 3

Combining Action Languages and ASP

Most of the previous approaches to use answer set programming in the field of planning hinge on previously existing action languages mapping the particular semantics of these languages to answer set semantics by appropriate transformations. In this chapter, we introduce the action language $\mathcal{K}^c$ which substantially distinguishes from the abovementioned languages in that its semantics is closely related to Answer Set Programming by definition. It adapts useful concepts and knowledge representation features from Answer Set Programming, such as state minimality and default negation.

The transition-based semantics of $\mathcal{K}^c$ is defined by "states of knowledge" representing only a "known" clipping of the complete state of the world. We will show the usefulness of this view by several problem encodings in the subsequent chapters.

In the first section of this chapter we will introduce syntax and semantics of the core language $\mathcal{K}$ [Pol01, EFL$^+$03b]. We have further enriched this core language by useful macros. We will define a transition based semantics and two possible semantics for plans will be discussed in detail: optimistic and secure (i.e., conformant) plans.

In a further Step we will add a modular extension of our language incorporating the possibility to model action costs and introduce the notions of optimal and admissible plans wrt. costs. We call this extended language $\mathcal{K}^c$.

The following section will give a detailed picture of the computational complexity of the various planning tasks in our language, i.e. optimistic and secure planning with and without action costs. Together with the results in Section 2.2.2 this shall give us an idea of which problems in $\mathcal{K}$ can be solved by Answer Set Programming techniques, giving hints towards the translations provided in the Chapter 4.

Running Example: Crossing the Bridge

For illustration, we shall use variants of the following planning problem as a running example:

Problem 3.1. [**Simple Bridge Crossing Problem**] *Four persons, Joe, Jack, William and Averell, want to cross a river at night over a plank bridge, which can only hold up to two*

persons at a time. They have a lamp, which must be used when crossing. As it is pitch-dark and some planks are missing, someone must bring the lamp back to the others; no tricks (like throwing the lamp or halfway crosses, etc.) are possible.

3.1 $\mathcal{K}$ – An Action Language Based on ASP

3.1.1 Basic Syntax

In this section we will define the syntax of the basic fragment $\mathcal{K}$ of our language. This basic language is already capable of expressing various complex planning problems as will be exemplified. It is originally based on similar action languages $\mathcal{A}$,$\mathcal{B}$, and $\mathcal{C}$ but semantically closer to answer set programming.

3.1.1.1 Actions, Fluents, and Types

Let σ^{act}, σ^{fl}, and σ^{typ} be disjoint sets of action, fluent and type names, respectively. These names are effectively predicate symbols with associated arity (≥ 0). Here, σ^{fl} and σ^{act} are used to describe *dynamic knowledge*, whereas σ^{typ} is used to describe *static background knowledge*. For convenience, we tacitly assume that σ^{typ} contains built-in predicates, in particular "<","<=" ,">" , ">=", "!=" with arity 2 and the obvious meanings as in `DLV` (cf. Section 2.1.5.1), which are not explicitly shown. Furthermore, let σ^{con} and σ^{var} be the disjoint sets of constant and variable symbols, respectively. [10]

Definition 3.1. *Given $p \in \sigma^{act}$ (resp. σ^{fl}, σ^{typ}), an* action (resp. fluent, type) atom *is defined as $p(t_1, \ldots, t_n)$, where n is the arity of p and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var}$. An action (resp. fluent, type) literal is an action (resp. fluent, type) atom a or its negation $\neg a$, where "$\neg$" is the true negation symbol, for which we also use the customary "–".*

As usual, a literal (and any other syntactic object) is *ground*, if it does not contain variables.

Given a literal l, let $\neg.l$ denote its complement, i.e., $\neg.l = a$ if $l = \neg a$ and $\neg.l = \neg a$ if $l = a$, where a is an atom. A set L of literals is *consistent*, if $L \cap \neg.L = \emptyset$. Furthermore, L^+ (resp. L^-) denotes the set of positive (resp. negative) literals in L.

The set of all action (resp. fluent, type) literals is denoted as $\mathcal{L}_{act}$ (resp. $\mathcal{L}_{fl}$, $\mathcal{L}_{typ}$). Furthermore, $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$; $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup \mathcal{L}^+_{act}$ (*dyn* stands for *dynamic literals*); and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup \mathcal{L}^+_{act}$. [11]

3.1.1.2 Static Background Knowledge: Type Definitions

Static knowledge which is invariant over time in a $\mathcal{K}$ planning domain is specified in a normal (disjunction-free) logic program Π which is assumed to be safe in the standard LP sense (cf. [Ull89]) and that has a single answer set. This is the case if, for instance, the well-founded

[10] Following logic programming conventions, constant and variable symbols are denoted as strings starting with a lower or upper case letter, respectively, i.e. the same restrictions for σ^{var}, σ^{con} and $\sigma^{pred} = \sigma^{act} \cup \sigma^{fl} \cup \sigma^{typ}$ apply as in Section 2.1).

[11] Note that this definition only allows positive action literals.

model of Π, cf. [vRS91], is total, which is guaranteed for stratified logic programs and other (syntactic) classes of programs. Therefore, Π can informally be viewed as a set of facts.

Example 3.1. For the bridge crossing problem (see Problem 3.1), the background knowledge Π_{BCP} consists of the following rules and facts which specify the four persons and the two opposite sides of the bridge:

```
person(joe). person(jack). person(william). person(averell).
side(here). side(across).
otherSide(X,Y) :- side(X), side(Y), X != Y.
```

This program obviously has a unique Answer Set S, corresponding to the set of static background facts:

$$S = \{\texttt{person(joe)}, \texttt{person(jack)}, \texttt{person(william)}, \texttt{person(averell)}, \\ \texttt{side(here)}, \texttt{side(across)}, \texttt{otherSide(here, across)}, \texttt{otherSide(across, here)}\}$$

◇

3.1.1.3 Dynamic Knowledge: Fluents and Action Declarations

All actions and fluents in the domain of discourse must be declared using statements as follows:

Definition 3.2. *An* action *(resp.* fluent*)* declaration*, is of the form:*

$$p(X_1, \ldots, X_n) \texttt{ requires } t_1, \ldots, t_m. \tag{3.1}$$

where

(1) $p \in acts$ *(resp.* $p \in \sigma^{fl}$*),*

(2) $X_1, \ldots, X_n \in \sigma^{var}$ *where* $n \geq 0$ *is the arity of* p*,*

(3) $t_1, \ldots, t_m \in \mathcal{L}_{typ}$*,* $m \geq 0$*, such that every* X_i *occurs in* $t_1, \ldots, t_m$*.*

If $m = 0$, the keyword `requires` may be omitted. In the following, we generically refer to action and fluent declarations as *type declarations* when no further distinction is necessary.

Example 3.2. In our running example, we have three actions for persons crossing alone, crossing in two and taking the lamp, which could be declared in $\mathcal{K}$ as follows:

```
crossTogether(X,Y) requires person(X), person(Y), X != Y.
cross(X) requires person(X).
takeLamp(X) requires person(X).
```

Furthermore, we declare fluents indicating at which side of the bridge a person currently is and who has got a lamp:

```
at(X,S) requires person(X), side(S).
hasLamp(X) requires person(X).
```

◇

3.1.1.4 Causation Rules and Executability Conditions

We next define causation rules, by which static and dynamic dependencies of fluents on other fluents and actions are specified. These are syntactically borrowed from causation laws of $\mathcal{C}$, extending them by default negation in the `if`- and `after`-parts:

Definition 3.3. *A* causation rule *(*rule*, for short) is an expression of the form*

$$\begin{aligned} &\texttt{caused}\ f\ \texttt{if}\ b_1,\ldots,b_k,\texttt{not}\ b_{k+1},\ldots,\texttt{not}\ b_l \\ &\qquad\texttt{after}\ a_1,\ldots,a_m,\texttt{not}\ a_{m+1},\ldots,\texttt{not}\ a_n. \end{aligned} \tag{3.2}$$

where $f \in \mathcal{L}_{fl} \cup \{\texttt{false}\}$, $b_1,\ldots,b_l \in \mathcal{L}_{fl,typ}$, $a_1,\ldots,a_n \in \mathcal{L}$, $l \geq k \geq 0$, *and* $n \geq m \geq 0$.

Rules where $n = 0$ are referred to as *static rules*, all other rules as *dynamic rules*. When $l = 0$, the keyword `if` is omitted; likewise, if $n = 0$, the keyword `after` is dropped. If both $l = n = 0$ then `caused` is optional.

Example 3.3. For instance the effect of two persons crossing the bridge together, can be described in $\mathcal{K}$ as follows:

```
caused at(X,S1) after crossTogether(X,Y), at(X,S), otherSide(S,S1).
caused at(Y,S1) after crossTogether(X,Y), at(Y,S), otherSide(S,S1).
caused -at(X,S) after crossTogether(X,Y), at(X,S).
caused -at(Y,S) after crossTogether(X,Y), at(Y,S).
```

where the latter two rules express that a person is no longer where it was before after crossing. Inertia of fluent `at` on the other hand could be expressed by the following rule:

```
caused at(X,S) if not -at(X,S) after at(X,S).
```

Note that this differs from the notion of inertia in $\mathcal{C}$, and we refer to Section 3.1.2 for the details. ◇

To access the parts of a causation rule r, we use the following notations: $\mathsf{h}(r) = \{f\}$, $\mathsf{post}^+(r) = \{b_1,\ldots,b_k\}$, $\mathsf{post}^-(r) = \{b_{k+1},\ldots,b_l\}$, $\mathsf{pre}^+(r) = \{a_1,\ldots,a_m\}$, $\mathsf{pre}^-(r) = \{a_{m+1},\ldots,a_n\}$, and $\mathsf{lit}(r) = \{f, b_1,\ldots,b_l, a_1,\ldots,a_n\}$. Intuitively, $\mathsf{pre}^+(r), \mathsf{pre}^-(r)$ access the state before some action(s) happen, and $\mathsf{post}^+(r), \mathsf{post}^-(r)$ the part after the actions have been executed.

While the scope of general static rules is over all knowledge states, it is often useful to specify rules only for the initial states.

Definition 3.4. *An* initial state constraint *is a static rule of the form (3.2) preceded by the keyword* `initially`.

For an initial state constraint ic, $\mathsf{h}(ic)$, $\mathsf{post}^+(ic)$, and $\mathsf{post}^-(ic)$, are defined as for its rule part, and $\mathsf{pre}^+(ic) = \mathsf{pre}^-(ic) = \emptyset$.

Example 3.4. In order to express for instance that joe initially has the lamp, in $\mathcal{K}$ we would simply write:

```
initially caused hasLamp(joe).
```

where `caused` here is optional. ◇

The language $\mathcal{K}$ allows STRIPS-style [FN71] conditional execution of actions, where $\mathcal{K}$ allows several alternative executability conditions for an action; this is beyond the repertoire of standard STRIPS and more or less borrowed from $\mathcal{A}$.

Definition 3.5. *An* executability condition *is an expression of the form*

$$\texttt{executable}\ a\ \texttt{if}\ b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_l \tag{3.3}$$

where $a \in \mathcal{L}^+_{act}$ *and* $b_1, \ldots, b_l \in \mathcal{L}$, *and* $l \geq k \geq 0$.

If $l = 0$ (which means that the executability is unconditional), then the keyword `if` is skipped.

Example 3.5. In our example, crossing the bridge in two is only possible if either of the persons crossing has got a lamp and both are at the same side of the bridge. This is expressible in $\mathcal{K}$ by two executability conditions:

```
executable crossTogether(X,Y) if hasLamp(X), at(X,S), at(Y,S).
executable crossTogether(X,Y) if hasLamp(Y), at(X,S), at(Y,S).
```

◇

Given an executability condition e, we access its parts with $\mathsf{h}(e) = \{a\}$, $\mathsf{pre}^+(e) = \{b_1, \ldots, b_k\}$, $\mathsf{pre}^-(e) = \{b_{k+1}, \ldots, b_l\}$, and $\mathsf{lit}(e) = \{a, b_1, \ldots, b_l\}$. Intuitively, $\mathsf{pre}^+(e), \mathsf{pre}^-(e)$ refer to the state at which some action's suitability is evaluated. Here, as opposed to causation rules we do not consider a state after the execution of actions, and so no part $\mathsf{post}^+(r)$ is needed. Nonetheless, for convenience we define $\mathsf{post}^+(e) = \mathsf{post}^-(e) = \emptyset$.

Furthermore, for any executability condition, a rule, or an initial state constraint r, we define $\mathsf{post}(r) = \mathsf{post}^+(r) \cup \mathsf{post}^-(r)$, $\mathsf{pre}(r) = \mathsf{pre}^+(r) \cup \mathsf{pre}^-(r)$, and $\mathsf{b}(r) = \mathsf{b}^+(r) \cup \mathsf{b}^-(r)$, where $\mathsf{b}^+(r) = \mathsf{post}^+(r) \cup \mathsf{pre}^+(r)$, and $\mathsf{b}^-(r) = \mathsf{post}^-(r) \cup \mathsf{pre}^-(r)$.

In order to keep things simple, we will now leave our concrete Bridge crossing instance. We will occasionally explain some of the concepts of $\mathcal{K}$ in short by the following abstract example between the formal definitions. By the end of this section we will apply these concepts discussing a full $\mathcal{K}$ encoding of the Bridge Crossing Problem.

Example 3.6. Summarizing the definitions for the parts of $\mathcal{K}$ statements, consider the following type declarations, causation rule, and executability condition, respectively, where $\sigma^{typ} = \{\mathtt{r}, \mathtt{s}\}$, $\sigma^{fl} = \{\mathtt{f}\}$, and $\sigma^{act} = \{\mathtt{ac}\}$:

```
d1 : f(X) requires -r(X,Y), s(Y,Y).
d2 : ac(X,Y) requires s(X,Y).
r1 : caused f(X) if s(X,X), not -f(X) after ac(X,Y), not -r(X,X).
e1 : executable ac(X,Y) if s(Z,Y), not f(X), Z != Y.
```

Then, we have $\mathsf{h}(r_1) = \{\mathtt{f(X)}\}$, $\mathsf{pre}(r_1) = \{\mathtt{ac(X,Y)}, \mathtt{-r(X,X)}\}$, $\mathsf{post}(r_1) = \{\mathtt{s(X,X)}, \mathtt{-f(X)}\}$, and $\mathsf{lit}(r_1) = \{\mathtt{f(X)}, \mathtt{ac(X,Y)}, \mathtt{-r(X,X)}, \mathtt{s(X,X)}, \mathtt{-f(X)}\}$; Furthermore, $\mathsf{h}(e_1) = \mathtt{ac(X,Y)}$, $\mathsf{pre}(e_1) = \{\mathtt{s(Z,Y)}, \mathtt{f(X)}, \mathtt{Z != Y}\}$, $\mathsf{post}(e_1) = \emptyset$, and $\mathsf{lit}(e_1) = \{\mathtt{ac(X,Y)}, \mathtt{s(Z,Y)}, \mathtt{f(X)}, \mathtt{Z != Y}\}$ ◇

3.1.1.5 Safety Restriction

All rules (including initial state constraints and executability conditions) have to satisfy the following syntactic restriction, which is similar to the notion of safety in logic programs [Ull89], see also Section 2.1.5.1.

Definition 3.6. *An executability condition, rule, or initial state constraint r is* safe, *if every variable occurring in literals in $U = (\mathsf{post}^-(r) \cup \mathsf{pre}^-(r)) \cap \mathcal{L}_{typ}$ also occurs in at least one literal in $\mathsf{lit}(r) \setminus U$.*

In other words, all variables in a default-negated type literal must also occur in some literal which is not a default-negated type literal.

Thus, safety is required only for variables appearing in default-negated type literals, while it is not required at all for variables appearing in fluent and action literals. The reason is that the range of the latter variables is implicitly restricted by the respective type declarations. Observe that the rules in the examples above are all safe.

3.1.1.6 Planning Domains and Planning Problems

We now define planning domains and problems. Let us call any pair $\langle D, R\rangle$ where D is a finite set of action and fluent declarations and R is a finite set of safe causation rules, safe initial state constraints, and safe executability conditions an *action description.*

Definition 3.7. *A* planning domain *is a pair $PD = \langle \Pi, AD\rangle$, where Π is a normal logic program over the literals of $\mathcal{L}_{typ}$ (referred to as* background knowledge*) as defined above, and AD is an action description. We say that PD is* positive, *if no default negation occurs in AD.*

We recall that we require the program Π to have a unique answer set. This is for instance the case if Π has a total well-founded model M [vRS91], which can be computed efficiently. In particular, each stratified program Π has a total well-founded model. The semantic condition of a total well-founded model admits a limited use of unstratified negation, which is convenient for knowledge representation purposes, and in particular for expressing default properties.

Planning domains represent the universe of discourse for solving concrete planning problems, which are defined next.

Definition 3.8. *A* planning problem *$\mathcal{P} = \langle PD, q\rangle$ is a pair of a planning domain PD and a goal query q, where a* query *is an expression of the form*

$$g_1, \ldots, g_m, \texttt{not } g_{m+1}, \ldots, \texttt{not } g_n \ ?\ (i) \tag{3.4}$$

where $g_1, \ldots, g_n \in \mathcal{L}_{fl}$ are variable-free, $n \geq m \geq 0$, and $i \geq 0$ denotes the plan length.

3.1.2 Basic Semantics

For defining the semantics of $\mathcal{K}$ planning domains and planning problems, we start with the preliminary definition of the typed instantiation of a planning domain. This is similar to the grounding of a logic program, with the difference being that only correctly typed fluent and action literals are generated.

3.1.2.1 Typed Instantiation

Let substitutions and their application to syntactic objects be defined as usual (i.e. assignments of constants to variables which replace the variables throughout the objects).

Definition 3.9. *Let $PD = \langle \Pi, \langle D, R \rangle \rangle$ be a $\mathcal{K}$ planning domain, and let M be the (unique) answer set of background knowledge Π. Then, $p(x_1, \ldots, x_n)$ is a* legal action *(resp.* fluent*)* instance *of an action (resp. fluent) declaration $d \in D$ of the form (3.1) if there exists some ground substitution θ for $\sigma^{var}(d)$ such that $X_i\theta = x_i$, for $1 \leq i \leq n$ and $\{t_1\theta, \ldots, t_m\theta\} \subseteq M$. Any such θ is called a* witness substitution *for $p(x_1, \ldots, x_n)$.*

By $\mathcal{L}_{PD}$ we denote the set of all legal action instances, legal fluent instances (also referred to as positive legal fluent instances*) and classically negated legal fluent instances (*negative legal fluent instances*).*

Based on this, we now define the instantiation of a planning domain respecting type information as follows.

Definition 3.10. *For any planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, its* typed instantiation *is given by $PD\downarrow = \langle \Pi\downarrow, \langle D, R\downarrow \rangle \rangle$, where $\Pi\downarrow$ is the ground instantiation of Π (over σ^{con}, as defined in Section 2.1) and $R\downarrow = \{\theta(r) \mid r \in R, \theta \in \Theta_r\}$, where Θ_r is the set of all substitutions θ of the variables in r using σ^{con}, such that $\mathsf{lit}(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD}$*

In other words, in $PD\downarrow$ we replace Π and R by their ground versions, but keep of the latter only rules where the atoms of all fluent and action literals agree with their declarations. We say that a $PD = \langle \Pi, \langle D, R \rangle \rangle$ is *ground*, if Π and R are ground, and moreover that it is *well-typed*, if PD and $PD\downarrow$ coincide.

Analogously, for a planning problem $\mathcal{P} = \langle PD, q \rangle$ we define $\mathcal{P}\downarrow = \langle PD\downarrow, q \rangle$. We recall that q is ground by Definition 3.8.

3.1.2.2 States and Transitions

We are now prepared to define the semantics of a planning domain, which is given in terms of states and transitions between states.

Definition 3.11. *A* state *with respect to a planning domain PD is any consistent set $s \subseteq \mathcal{L}_{fl} \cap \mathcal{L}_{PD}$ of positive and negative legal fluent instances. A tuple $t = \langle s, A, s' \rangle$ where s, s' are states and $A \subseteq \mathcal{L}_{act} \cap \mathcal{L}_{PD}$ is a set of legal action instances in PD is called a* state transition.

Observe that a state does not necessarily contain either f or $\neg f$ for each legal instance f of a fluent. In fact, a state may even be empty ($s = \emptyset$). The empty state represents a "tabula rasa" state of knowledge about the fluent values in the planning domain. Furthermore, in this definition, state transitions are not constrained – this will be done in the definition of legal state transitions, which we develop now. To ease the intelligibility of the semantics, we proceed in analogy to the definition of answer sets in [GL91] in two steps. We first define the semantics for positive planning problems, i.e. planning problems without default negation, and then we define the semantics of general planning domains by a reduction to positive planning domains.

In what follows, we assume that $PD = \langle \Pi, \langle D, R \rangle \rangle$ is a ground planning domain which is well-typed, and that M is the unique answer set of Π. For any other PD, the respective concepts are defined through its typed instantiation $PD{\downarrow}$.

Definition 3.12. *A state s_0 is a* legal initial state *for a positive PD, if s_0 is the smallest (under inclusion) set such that $\mathsf{post}(c) \subseteq s_0 \cup M$ implies $\mathsf{h}(c) \subseteq s_0$, for all initial state constraints and static rules $c \in R$.*

Definition 3.13. *For a positive PD and a state s, a set $A \subseteq \mathcal{L}^+_{act}$ is called* executable action set *wrt. s, if for each $a \in A$ there exists an executability condition $e \in R$ such that $\mathsf{h}(e) = \{a\}$, $\mathsf{pre}^+(e) \cap \mathcal{L}_{fl,typ} \subseteq s \cup M$, $\mathsf{pre}^+(e) \cap \mathcal{L}^+_{act} \subseteq A$.*

Note that this definition allows for modeling dependent actions, i.e., actions which depend on the execution of other actions.

Definition 3.14. *Given a positive PD, a causation rule $r \in R$ is* satisfied *by a state s' wrt. a state transition $t = \langle s, A, s' \rangle$ if and only if either $\mathsf{h}(r) \subseteq s' \setminus \{\mathtt{false}\}$ or not all of (i)–(iii) hold: (i) $\mathsf{post}(r) \subseteq s' \cup M$, (ii) $\mathsf{pre}(r) \cap \mathcal{L}_{fl,typ} \subseteq s \cup M$, and (iii) $\mathsf{pre}(r) \cap \mathcal{L}_{act} \subseteq A$. A state transition $t = \langle s, A, s' \rangle$ is called* legal, *if A is an executable action set wrt. s and s' is the minimal consistent set that satisfies all causation rules in R except initial state constraints wrt. t.*

The above definitions are now generalized to a well-typed ground PD containing default negation by means of a reduction to a positive planning domain, which is similar in spirit to the Gelfond-Lifschitz reduct [GL91] (cf. Section 2.1).

Definition 3.15. *Let PD be a ground and well-typed planning domain, and let $t = \langle s, A, s' \rangle$ be a state transition. Then, the* reduction $PD^t = \langle \Pi, \langle D, R^t \rangle \rangle$ *of PD by t is the planning domain where R^t is obtained from R by deleting*

1. *every causal rule, executability condition, and initial state constraint $r \in R$ for which either $\mathsf{post}^-(r) \cap (s' \cup M) \neq \emptyset$ or $\mathsf{pre}^-(r) \cap (s \cup A \cup M) \neq \emptyset$ holds, and*
2. *all default literals* `not` L *($L \in \mathcal{L}$) from the remaining $r \in R$.*

Note that PD^t is positive and ground. Legal initial states, executable action sets, and legal state transitions are now defined as follows.

Definition 3.16. *Let PD be any planning domain. Then, a state s_0 is a* legal initial state, *if s_0 is a legal initial state for PD^t, where $t = \langle \emptyset, \emptyset, s_0 \rangle$; a set A is an* executable action set *in PD wrt. a state s, if A is executable wrt. s in PD^t with $t = \langle s, A, \emptyset \rangle$; and, a state transition $t = \langle s, A, s' \rangle$ is* legal *in PD, if it is legal in PD^t.*

Example 3.7. Reconsider the type declarations d_1 and d_2, causation rule r_1 and executability condition e_1 in Example 3.6. Suppose σ^{con} contains two constants `a` and `b`, and that the background knowledge Π has the following answer set: $M = \{-\mathtt{r(a,b)}, \mathtt{r(b,a)}, \mathtt{s(a,a)}, \mathtt{s(a,b)}, \mathtt{s(b,b)}\}$. Then, e.g. `f(a)` is a legal fluent instance of d_1,

```
f(X) requires -r(X,Y), s(Y,Y).
```

where $\theta = \{\mathtt{X} = \mathtt{a},\ \mathtt{Y} = \mathtt{b}\}$. Similarly, $\mathtt{ac(a,b)}$ is a legal action instance of declaration d_2,

$$\mathtt{ac(X,Y)\ requires\ s(X,Y).}$$

where $\theta = \{\mathtt{X} = \mathtt{a},\ \mathtt{Y} = \mathtt{b}\}$. Thus, $\mathtt{f(a)}$ and $\mathtt{ac(a,b)}$ belong to $\mathcal{L}_{PD}$. The empty set $s_0 = \emptyset$ is a legal initial state, and in fact the only one since there are no initial state constraints or static causation rules in PD, and thus also not in PD^t for every $t = \langle \emptyset, \emptyset, s_0 \rangle$. The action set $A = \{\mathtt{ac(a,b)}\}$ is executable wrt. s_0, since for $t = \langle s_0, A, \emptyset \rangle$, the reduct PD^t contains the executability condition

$$e_1' :\ \mathtt{executable\ ac(a,b)\ if\ s(a,b),\ a!=b.}$$

and both $\mathtt{s(a,b)}$ and $\mathtt{a!=b}$ are contained in $s_0 \cup M$. Thus, we can easily verify that $t = \langle s_0, A, s_1 \rangle$, where $A = \{\mathtt{ac(a,b)}\}$ and $s_1 = \{\mathtt{f(a)}\}$ is a legal state transition: PD^t contains a single causation rule

$$r_1' :\ \mathtt{caused\ f(a)\ if\ s(a,a)\ after\ ac(a,b).}$$

which results from r_1 for $\theta = \{\mathtt{X} = \mathtt{a},\ \mathtt{Y} = \mathtt{b}\}$. Clearly, s_1 satisfies this rule, as $\mathtt{h}(r_1') \subseteq s_1$, and s_1 is smallest, since $\mathtt{s(a,a)} \in M$ and $\mathtt{ac(a,b)} \in A$ holds. On the other hand, $t = \langle s_0, A', s_1 \rangle$, where $A' = \{\mathtt{ac(a,b)},\ \mathtt{ac(b,b)}\}$ is not a legal transition: while $\mathtt{ac(b,b)}$ is a legal action instance, there is no executability condition for it in $PD{\downarrow}^t$, and thus $\mathtt{ac(b,b)}$ is not executable in PD wrt. s_0. ◇

3.1.2.3 Plans

After having defined state transitions, we now formalize plans as suitable sequences of state transitions which lead from an initial state to some successor state which satisfies a given goal.

Definition 3.17. *Let PD be a planning domain. A sequence of state transitions $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{n-1}, A_n, s_n \rangle \rangle$, $n \geq 0$, is a* trajectory *for PD, if s_0 is a legal initial state of PD and all $\langle s_{i-1}, A_i, s_i \rangle$, $1 \leq i \leq n$, are legal state transitions of PD.*

Note that in particular, $T = \langle\rangle$ is empty if $n = 0$.

Definition 3.18. *Given a planning problem $\mathcal{P} = \langle PD, q \rangle$, where q has form (3.4), a sequence of action sets $\langle A_1, \ldots, A_i \rangle$, $i \geq 0$, is an* optimistic plan *for $\mathcal{P}$, if a trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ in PD exists such that T establishes the goal, i.e., $\{g_1, \ldots g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$.*

The notion of optimistic plan amounts to what in the literature is defined as "plan", "valid plan", or "weak plan" etc. The term "optimistic" should stress the credulous view underlying this definition, with respect to planning domains that provide only incomplete information about the initial state of affairs and/or bear nondeterminism in the action effects, i.e. alternative state transitions.

In such domains, the execution of an optimistic plan P is not a guarantee that the goal will be reached. We therefore resort to secure plans (alias conformant plans), which are defined as follows.

Definition 3.19. *An optimistic plan* $\langle A_1, \ldots, A_n \rangle$ *is a* secure plan, *if for every legal initial state* s_0 *and trajectory* $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$ *such that* $0 \leq j \leq n$, *it holds that (i) if* $j = n$ *then* T *establishes the goal, and (ii) if* $j < n$, *then* A_{j+1} *is executable in* s_j *wrt. PD and some legal transition* $\langle s_j, A_{j+1}, s_{j+1} \rangle$ *exists.*

Observe that plans admit in general the concurrent execution of actions at the same time. However, in many cases the concurrent execution of actions may not be desired (and explicitly prohibited, as discussed below), and attention focused to plans with one action at a time. More formally, we call a plan $\langle A_1, \ldots, A_n \rangle$ *sequential* (or *non-concurrent*), if $|A_j| \leq 1$, for all $1 \leq j \leq n$.

3.1.3 Enhanced Syntax

While the language presented in Section 3.1.1 is complete and allows for a succinct semantics definition, it can be enhanced wrt. user-friendliness. E.g. it is inconvenient to write `initially` in front of each initial state constraint, having an `initially`-section in which each rule is interpreted as an initial state constraint would be more desirable. In addition, some frequently occurring patterns can be identified for which macros will be defined for convenience and readability.

3.1.3.1 Partitions

The specification of a planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$ respectively planning problem $\mathcal{P} = \langle \langle \Pi, \langle D, R \rangle \rangle, q \rangle$ can be seen as being partitioned into

- the background knowledge Π,
- F_D, the fluent declarations in D,
- A_D, the action declarations in D,
- I_R, the initial state constraints in R,
- C_R, the causation rules and executability conditions in R, and
- the query (or goal) q.

In the sequel, we will denote a planning problem as follows:

`fluents:`	F_D
`actions:`	A_D
`always:`	C_R
`initially:`	I_R
`goal:`	q

where each construct in F_D, A_D, C_R, and I_R is terminated by ".". The background knowledge is assumed to be represented separately. This representation will also be called $\mathcal{K}$ *program* in the following. A sample program is provided in Figure 3.1 which will be explained below.

3.1.3.2 Macros

In the following, we will define several macros which allow for a concise representation of frequently used concepts. Let $\mathtt{a} \in \mathcal{L}^{+}_{act}$ denote an action atom, $\mathtt{f} \in \mathcal{L}_{fl}$ a fluent literal, `B` a (possibly empty) sequence $b_1, \ldots, b_k$, `not` $b_{k+1}, \ldots,$ `not` b_l where each $b_i \in \mathcal{L}_{fl,typ}, i = 1, \ldots, l$, and `A` a (possibly empty) sequence $a_1, \ldots, a_m$, `not` $a_{m+1}, \ldots,$ `not` a_n where each $a_j \in \mathcal{L}, j = 1, \ldots, n$.

Inertia In order to express frame axioms as discussed in Section 2.3 it is convenient to be able to declare fluents as inertial, which means that these fluents keep their truth values in a state transition, unless explicitly affected by an action or causation rule. Similar (but different) to $\mathcal{C}$, in order to allow for an easy representation of this kind of situation, we have enhanced the language by the shortcut

`inertial f if B after A.` $\Leftrightarrow$ `caused f if not` $\neg$`.f, B after f, A.`

Defaults A default value of a fluent in the planning domain can be expressed by the shortcut

`default f.` $\Leftrightarrow$ `caused f if not` $\neg$`.f.`

This default is in effect unless there is evidence to the opposite value of fluent `f`, given through some other causation rule.

Totality For reasoning under incomplete, but total knowledge we introduce

`total f if B after A.` $\Leftrightarrow$
`caused f if not` $-$`f, B after A.`
`caused` $-$`f if not f, B after A.`

where `f` must be positive.

State Integrity It is very common to formulate integrity constraints for states (possibly referring to the respective preceding state). To this end, we define the macro

`forbidden B after A.` $\Leftrightarrow$ `caused false if B after A.`

Nonexecutability Sometimes it is more intuitive to specify when some action is not executable, rather than when it is. To this end, we introduce

`nonexecutable a if B.` $\Leftrightarrow$ `caused false after a, B.`

Note that because of this definition, `nonexecutable` is stronger than `executable`, so in case of conflicts, `executable` is overridden by `nonexecutable`.

Non-concurrent Plans Finally, `noConcurrency.` disallows the simultaneous execution of actions. We define

`noConcurrency.` $\Leftrightarrow$ `caused false after` $\mathtt{a}_1$, $\mathtt{a}_2$.

where $\mathtt{a}_1$ and $\mathtt{a}_2$ range over all possible actions such that $\mathtt{a}_1, \mathtt{a}_2 \in \mathcal{L}_{PD} \cap \mathcal{L}_{act}$ and $\mathtt{a}_1 \neq \mathtt{a}_2$.

In all macros, "`if B`" (resp. "`after A`") can be omitted, if `B` (resp. `A`) is empty. We reserve the possibility of including further macros in future versions of $\mathcal{K}$.

```
(1)  actions :    crossTogether(X,Y) requires person(X), person(Y), X != Y.

(2)               cross(X) requires person(X).
(3)               takeLamp(X) requires person(X).

(4)  fluents :    at(X,S) requires person(X), side(S).
(5)               hasLamp(X) requires person(X).

(6)  always :     executable crossTogether(X,Y) if hasLamp(X), at(X,S), at(Y,S).
(7)               executable crossTogether(X,Y) if hasLamp(Y), at(X,S), at(Y,S).
(8)               executable cross(X) if hasLamp(X).
(9)               executable takeLamp(X) if hasLamp(Y), at(X,S), at(Y,S).
(10)              nonexecutable takeLamp(X) if hasLamp(X).

(11)              caused at(X,S1) after crossTogether(X,Y), at(X,S), otherSide(S,S1).
(12)              caused at(Y,S1) after crossTogether(X,Y), at(Y,S), otherSide(S,S1).
(13)              caused -at(X,S) after crossTogether(X,Y), at(X,S).
(14)              caused -at(Y,S) after crossTogether(X,Y), at(Y,S).

(15)              caused at(X,S1) after cross(X), at(X,S), otherSide(S,S1).
(16)              caused -at(X,S) after cross(X), at(X,S).

(17)              caused hasLamp(X) after takeLamp(X).
(18)              caused -hasLamp(X) after takeLamp(Y), hasLamp(X), X != Y.

(19)              inertial at(X,S).
(20)              inertial hasLamp(X).

(21)              noConcurrency.

(22)  initially : caused at(X,here).
(23)              caused hasLamp(joe).

(24)  goal :      at(joe,across), at(jack,across),
(25)              at(william,across), at(averell,across)? (5)
```

Figure 3.1: Basic $\mathcal{K}$ encoding of the Bridge crossing problem

3.1.4 Solving the Simple Bridge Crossing Example

Now we are ready to discuss our first full $\mathcal{K}$ program and discuss the concepts of $\mathcal{K}$ in detail returning to the Bridge Crossing Problem. A sample $\mathcal{K}$ encoding of the Bridge Crossing Problem is shown in Figure 3.1.

Optimistic Planning We assume that the four involved persons are specified in a logic program Π_{BCP} as defined in Example 3.1. Let $\mathcal{P}_{BCP}$ be the planning problem defined by Π_{BCP} and the $\mathcal{K}$ program in lines 1–25 of Figure 3.1.

In lines 1 to 5 of Figure 3.1 the respective actions and fluents are declared as described in Example 3.2. Subsequently, line 6 to 21 contain the executability conditions and causation rules in R_{BCP}:

In particular, lines 6 to 8 fix that crossing is only allowed with a lamp and crossing in

two is only allowed if both persons involved are at the same side of the bridge and one holds the lamp.

Lines 9 and 10 which encode executability of taking the lamp exemplify the alternative negative formulation of executability stating that a person can not take the lamp if he/she holds it. Alternatively, we could substitute these two lines semantically equivalent by:

$$\texttt{executable takeLamp(X) if hasLamp(Y), at(X,S), at(Y,S), X != Y.}$$

In $\mathcal{K}$ one can often choose between several encodings of a problem which we believe to be one of the strengths of our language.

Lines 11 to 16 encode the intuitive effects of crossing the bridge in two or alone, by toggling the resp. values of fluent `at`. Similarly, lines 17 to 18 encode the effect of taking the lamp.

Note that, by lines 19 and 20 only positive values of fluents `at` and `hasLamp` are declared inertial. This is one of the key features of $\mathcal{K}$ action descriptions, best described by: "Only model what you (need to) know, forget about the rest". As we deal with states of knowledge and transitions in $\mathcal{K}$ represent transitions between such states, we only have to keep track of relevant information: Strong negation in lines 13,14 and 16 is only used to switch off inertia where this negative information materializes only in the successor state but is not carried over to subsequent states. As easily seen, classical negation is not needed anywhere else in the action description apart from inertia. Take for instance a state s such that $\texttt{at(joe,across)} \in s$. For any legal state transition $\langle s, \{\texttt{cross(joe)}\}, s'\rangle$, by line 16 $-\texttt{at(joe,across)} \in s'$ holds, whereas any subsequent transition $\langle s', A, s''\rangle$ in this example will lead to a state s'' such that $-\texttt{at(joe,across)} \notin s''$, i.e. this fact will be "forgotten". On the other hand, $\mathcal{K}$ is well capable of encoding all contingencies, i.e. world states, explicitly, if needed, which will be further discussed in Sections 10.1.0.4 and 6.3.

The line 21 enforces sequential planning as discussed above in Section 3.1.3.2. Finally, the initial situation is described in lines 22 and 23. Line 22 states that *all* persons are initially `here` which can be explained by Definition 3.9: In fact, the rule in line 22 is semantically equivalent to

$$\texttt{initially caused at(X,here) if person(X), side(here).}$$

wrt. the declaration of fluent `at` in line 4.

For finding an optimistic plan of our problem, we assume that `joe` initially carries the lamp (line 23).

So, the only legal initial state for the program in Figure 3.1 is:

$$s_0 = \{\texttt{at(joe,here)}, \texttt{at(jack,here)}, \texttt{at(william,here)}, \texttt{at(averell,here)}, \texttt{hasLamp(joe)}\}$$

Note that, we do not need to explicitly represent negative information. For instance, we do not state that `averell` is *not* `across` initially, but just leave the value of fluent `at(averell,across)` open because of our knowledge state view (which will be discussed in detail in Chapter 6).

Our goal is to bring all four persons across in 5 steps, expressed in $\mathcal{K}$ by line 24.

For the planning problem $\mathcal{P}_{BCP}$ defined by Π_{BCP} and the $\mathcal{K}$ program in Figure 3.1 there are simple optimistic plans: `joe` always carries the lamp and brings all others across. One such plan is:

$P_1 = \langle$ {crossTogether(joe, jack)}, {cross(joe)}, {crossTogether(joe, william)},
{cross(joe)}, {crossTogether(joe, averell)} $\rangle$

Secure Planning and Encoding Contingencies Strictly speaking, in the description of Problem 3.1 there was no clue justifying our assumption that joe initially has the lamp. Furthermore, let us assume that all but Joe are quite clumsy, and potentially lose the lamp when crossing alone. So, we are confronted with the following problems beyond classical planning:

1. We initially know that one of the four has a lamp but we actually do not know who holds it.

2. Anyone but Joe might nondeterministically lose the lamp upon crossing alone.

So, our scenario involves incomplete information on the initial state as well as nondeterministic actions. We will now show how to model this situation in $\mathcal{K}$. Our encoding now can be modified to reflect the incomplete knowledge on who initially has the lamp by dropping line 23 and substituting it with the following rules in the `initially`-section:

```
total hasLamp(X).
forbidden hasLamp(X), hasLamp(Y), X != Y.
forbidden -hasLamp(joe), -hasLamp(jack),
          -hasLamp(william), -hasLamp(averell).
```

These rules say that either person might have the lamp, at most one person has the lamp and that it is forbidden that none of them holds the lamp, respectively. The latter could also be expressed more concisely by adding a new fluent, e.g. `anybodyHasLamp`, plus the static causation rules:

```
caused anybodyHasLamp if hasLamp(X).
forbidden if not anybodyHasLamp.
```

We emphasize that the modifications discussed here show useful techniques how to enforce state integrity in $\mathcal{K}$ which are not bound to this particular example.

The resulting legal initial states are exactly those where one of the four has the lamp, i.e.

$s_{0,1}$ = {at(joe, here), at(jack, here), at(william, here), at(averell, here), hasLamp(joe)}
$s_{0,2}$ = {at(joe, here), at(jack, here), at(william, here), at(averell, here), hasLamp(jack)}
$s_{0,3}$ = {at(joe, here), at(jack, here), at(william, here), at(averell, here), hasLamp(william)}
$s_{0,4}$ = {at(joe, here), at(jack, here), at(william, here), at(averell, here), hasLamp(averell)}

As for the nondeterministic effect of crossing alone, we add the following in the `always`-section of our program:

```
total hasLamp(X) after cross(X), X != joe.
```

Additionally, we have to modify executability of action `takeLamp` in order to deal with all contingencies in the initial state: We therefore drop line 10 such that now `takeLamp(X)` is also executable in case `X` already holds the lamp.

Let us denote the modified planning problem, i.e. the problem obtained from dropping lines 10 and 23 from the program in Figure 3.1 and adding the following lines 26–31 instead, as $\mathcal{P}_{BCPsec}$:

```
(26) fluents :     anybodyHasLamp.
(27) initially :   total hasLamp(X).
(28)               forbidden hasLamp(X), hasLamp(Y), X != Y.
(29)               caused anybodyHasLamp if hasLamp(X).
(30)               forbidden if not anybodyHasLamp.
(31) always :      total hasLamp(X) after cross(X), X != joe.
```

However, this reformulation still does not allow for secure plans of length 5. For finding the shortest secure plan we can incrementally adapt plan length i in goal:

$$\texttt{goal}: \texttt{at(joe,across)}, \texttt{at(jack,across)}, \texttt{at(william,across)}, \texttt{at(averell,across)} \; ? \; (i)$$

until a secure plan is found. Actually, here we find secure plans of length 6 where again Joe brings all others across again but he has to take the lamp in the first step in order to guarantee a secure plan. One such secure plan is:

$$P_2 = \langle\, \{\texttt{takeLamp(joe)}\},\ \{\texttt{crossTogether(joe, jack)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, william)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, averell)}\}\ \rangle$$

(Strategies for finding shortest plan will be further discussed in Section 6.4.) Note that plan P_1 from above still remains one possible *optimistic* plan but it is not secure since it is not guaranteed that joe initially has the lamp.

3.2 Action Costs – Language $\mathcal{K}^c$

Using the core language $\mathcal{K}$, we can already express and solve some involved planning tasks, cf. [Pol01, EFL$^+$00a, EFL$^+$03b]. However, $\mathcal{K}$ alone offers no means for finding optimal plans under an objective cost function. In general, different criteria of plan optimality can be relevant, such as optimality wrt. action costs as shown in the next example, which is a slight elaboration of the Bridge Crossing Problem, and a well-known brain teasing riddle:

Problem 3.2. [**Quick Bridge Crossing Problem**] *The persons in the bridge crossing scenario need different times to cross the bridge, namely 1, 2, 5, and 10 minutes, respectively. Walking in two implies moving at the slower rate of both. Is it possible that all four persons get across within 17 minutes?*

On first thought this is infeasible, since the seemingly optimal (optimistic) plan P_1 from above where `joe`, who is the fastest, keeps the lamp and leads all the others across takes 19 minutes altogether. Surprisingly, as we will see, the optimal solution indeed only takes 17 minutes.

In order to allow for an elegant and convenient encoding of such optimization problems, we extend $\mathcal{K}$ to the language $\mathcal{K}^c$ in which one can assign costs to actions.

3.2.1 Syntax of $\mathcal{K}^c$

$\mathcal{K}^c$ extends action declarations as in $\mathcal{K}$ with costs as follows.

Definition 3.20. *An* action declaration d *in* $\mathcal{K}^c$ *is of the form:*

$$p(X_1, \ldots, X_n) \texttt{ requires } t_1, \ldots, t_m \texttt{ costs } C \texttt{ where } c_1, \ldots, c_k. \tag{3.5}$$

where

(1) $p \in \sigma^{act}$,

(2) $X_1, \ldots, X_n \in \sigma^{var}$ *where* $n \geq 0$ *is the arity of* p,

(3) $t_1, \ldots, t_m$, $c_1, \ldots, c_k$ *are from* $\mathcal{L}_{typ}$ *such that every* X_i *occurs in* $t_1, \ldots, t_m$,

(4) C *is either an integer constant, a variable from the set of all variables occurring in* $t_1, \ldots, t_m$, $c_1, \ldots, c_k$ *(denoted by* $\sigma^{var}(d)$*), or the distinguished variable* `time`,

(5) $\sigma^{var}(d) \subseteq \sigma^{var} \cup \{\texttt{time}\}$, *and*

(6) `time` *does not occur in* $t_1, \ldots t_m$.

If $m = 0$, the keyword '`requires`' is omitted; if $k = 0$, the keyword '`where`' is omitted and '`costs` C' is optional. Here, (1) and (2) state that parameters to an action must be variables, and not fixed values. Informally, (3) means that all parameters of an action must be "typed" in the `requires` part. Condition (4) asserts that the cost is locally defined or given by the stage of the plan, which is referenced through the global variable `time`. Conditions (5) and (6) ensure that all variables are known and that type information of action parameters is static, i.e., does not depend on time.

Fluent declarations, as well as planning domains and planning problems in $\mathcal{K}^c$ are defined as in $\mathcal{K}$.

Example 3.8. For example, in the elaborated Bridge Crossing Problem, the declaration of `cross(X)` can be extended as follows: Suppose a predicate `walk(Person,Minutes)` in the background knowledge indicates that `Person` needs `Minutes` to walk across. Then, we may simply declare

`cross(X) requires person(X) costs WX where walk(X,WX).`

◇

3.2.2 Semantics of $\mathcal{K}^c$

Semantically, $\mathcal{K}^c$ extends $\mathcal{K}$ by the cost values of actions at points in time. Recall that in any plan $P = \langle A_1, \ldots, A_l \rangle$, at step $1 \leq i \leq l$, the actions in A_i are executed to reach time point i.

First, we slightly modify Definition 3.9 wrt. legal action instances:

Definition 3.21. *Let* $PD = \langle \Pi, \langle D, R \rangle \rangle$ *be a* $\mathcal{K}^c$ *planning domain, and let* M *be the (unique) answer set of* Π. *A ground action* $p(x_1, \ldots, x_n)$ *is a* legal action instance *of an action declaration* $d \in D$ *of the form (3.5) if there exists some ground substitution* θ *for* $\sigma^{var}(d) \cup \{\texttt{time}\}$ *such that* $X_i\theta = x_i$, *for* $1 \leq i \leq n$ *and* $\{t_1\theta, \ldots, t_m\theta\} \subseteq M$.

Witness substitutions and $\mathcal{L}_{PD}$ are defined as above. Action costs are now formalized as follows.

Definition 3.22. *Let $a = p(x_1, \ldots, x_n)$ be a legal action instance of a declaration d of the form (3.5), let $i \geq 1$ be a time point, and let θ be a witness substitution for a. such that* `time`$\theta = i$. *Then*

$$cost_\theta(p(x_1, \ldots, x_n)) = \begin{cases} 0, & \text{if the } \texttt{costs}\text{-part of } d \text{ is empty;} \\ val(C\theta), & \text{if } \{c_1\theta, \ldots, c_k\theta\} \subseteq M; \\ undefined & \text{otherwise.} \end{cases}$$

where M is the unique answer set of Π and $val : \sigma^{con} \to \mathbb{N}$ is defined as the integer value for integer constants and undefined for all non-integer constants.

By reference to the variable `time`, it is possible to define time-dependent action costs; we shall consider an example in Section 6.5. Using $cost_\theta$, we now introduce well-defined legal action instances and define action cost values as follows.

Definition 3.23. *A legal action instance $a = p(x_1, \ldots, x_n)$ is* well-defined, *if it holds that*

(i) for any time point $i \geq 1$, there is some witness substitution θ for a such that `time`$, \theta = i$ *and $cost_\theta(a)$ is defined* [12]*, and*

(ii) $cost_\theta(a) = cost_{\theta'}(a)$ holds for any two witness substitutions θ, θ' which coincide on `time` *and have defined costs.*

For any well-defined a, its unique cost at time point $i \geq 1$ is given by $cost_i(a) = cost_\theta(a)$ where θ is as in (i).

In this definition, condition (i) ensures that some cost value exists, which must be an integer, and condition (ii) ensures that this value is unique, i.e., any two different witness substitutions θ and θ' for a evaluate the `costs`-part to the same integer cost value.

In particular, here our claim that the answer set M of Π has to be unique comes into play in order to guarantee a unique cost value. However, this claim alone is not sufficient but we remark that checking well-definedness can be expressed easily as a planning task in $\mathcal{K}$, and also by a logic program; this will be considered later on in Section 4.3.7.

An action declaration d is *well-defined*, if all its legal instances are well-defined. This will be fulfilled if, in database terms, the variables $X_1, \ldots, X_n$ together with `time` in (3.5) functionally determine (cf. for instance [AHV95]) the value of C. In our framework, the semantics of a $\mathcal{K}^c$ planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$ (resp. a planning problem $\mathcal{P} = \langle PD, q \rangle$) is only well-defined for well-defined action declarations in PD. In the rest of this work, we assume well-definedness of $\mathcal{K}^c$ unless stated otherwise.

Using $cost_i$, we now define costs of and plans.

Definition 3.24. *Let $\mathcal{P} = \langle PD, Q\,?\,(l) \rangle$ be a planning problem. Then, for any plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$, its cost is defined as*

$$cost_{\mathcal{P}}(P) = \sum_{j=1}^{l} \left(\sum_{a \in A_j} cost_j(a) \right).$$

[12] This implies that, in order to write well-defined declarations, one may use only variables bound to an integer value for the costs of an action.

A plan P is optimal *for $\mathcal{P}$, if $cost_{\mathcal{P}}(P) \leq cost_{\mathcal{P}}(P')$ for each plan P' for $\mathcal{P}$, i.e., P has least cost among all plans for $\mathcal{P}$. The cost of a planning problem $\mathcal{P}$, denoted $cost^*_{\mathcal{P}}$, is given by $cost^*_{\mathcal{P}} = cost_{\mathcal{P}}(P^*)$, where P^* is an optimal plan for $\mathcal{P}$.*

In particular, $cost_{\mathcal{P}}(P) = 0$ if $P = \langle\rangle$, i.e., the plan is void. Note that $cost^*_{\mathcal{P}}$ is only defined if a plan for $\mathcal{P}$ exists.[13]

Usually one only can estimate some *upper bound* of the plan length, but does not know the exact length of an optimal plan. Although we have only defined optimality for a fixed plan length l, we will see in Section 6.4 that by appropriate encodings this can be extended to optimality for plans with length *at most* l.

Besides optimal plans, also plans with bounded costs are of interest, which motivates the following definition.

Definition 3.25. *A plan P for a planning problem $\mathcal{P}$ is* admissible *wrt. cost c, if $cost_{\mathcal{P}}(P) \leq c$.*

Admissible plans impose a weaker condition on the plan quality than optimal plans. They are particularly relevant if optimal costs are not a crucial issue, as long as the cost stays within a given limit, and if optimal plans are difficult to compute. We might face questions like "Can I make it to the airport within one hour?", "Do I have enough change to buy a coffee?" etc. which amount to admissible planning problems. As we shall see in Section 3.3, computing admissible plans is complexity-wise easier than computing optimal plans.

3.2.3 An Optimal Solution for the Quick Bridge Crossing Problem

To model the Quick Bridge Crossing Problem in $\mathcal{K}^c$, we first extend the background knowledge Π_{BCP} with the predicate `walk` as already mentioned in Example 3.8 by adding facts

```
walk(joe, 1). walk(jack, 2). walk(william, 5). walk(averell, 10).
```

We denote the extended background knowledge as Π_{QBCP}.

Finally, all we have to do, to get a $\mathcal{K}^c$ formulation for the optimal planning instance of the Quick Bridge Crossing problem is to slightly modify the action declarations for crossing alone or in two in lines 1 and 2 of Figure 3.1. We denote the resulting $\mathcal{K}^c$ planning problem where obtained from Π_{QBCP} and substituting lines 1 and 2 of Figure 3.1 by the following lines 32–34 as $\mathcal{P}_{QBCP}$:

```
(32) actions :  cross(X) requires walk(X, WX) costs WX.
(33)            crossTogether(X, Y) requires walk(X, WX), walk(Y, WY), X != Y,
(34)                        WX <= WY costs WY.
```

These two declarations model exactly the intended costs, i.e., crossing alone costs the time defined in in the `walk` predicate Π_{QBCP} and crossing in two costs the time which the slower of both crossing persons needs.

Remark 3.1. *Note that this formulation differs slightly from Example 3.8 and from the original $\mathcal{K}^c$ encoding of this example which appeared in [EFL+03c]. These declarations are more concise than the ones suggested in [EFL+03c] where we formalized crossing as follows:*

[13] In the following, subscripts will be dropped when clear from the context.

```
cross(X) requires person(X) costs WX where walk(X,WX).
crossTogether(X,Y) requires person(X), person(Y), X < Y
                   costs Wmax where walk(X,WX),walk(Y,WY),max(WX,WY,Wmax).
```

where $\texttt{max}(\cdot,\cdot,\cdot)$ *was a type predicate defined in the background knowledge. The present encoding (a) does not require additional* `where` *clauses, (b) does not require the additional* `max` *predicate, and (c) prunes the number of legal action instances for action* `crossTogether`*: By* `WX <= WY` *in line 34 we fix the order for predicate* $\texttt{crossTogether}(\texttt{X},\texttt{Y})$ *such that* `X` *always is the faster of both.*

Using the modified planning problem $\mathcal{P}_{QBCP}$, the 5-step plan P_1 reported on page 39 has cost 19. Actually, it is optimal for plan length $l = 5$. However, when we abandon the first intuition that the fastest person, `joe`, always has the lamp and consider the problem under varying plan length, then we can find the following 7-step plan:

$P_3 = \langle$ {crossTogether(joe, jack)}, {cross(joe)}, {takeLamp(william)},
{crossTogether(william, averell)}, {takeLamp(jack)}, {cross(jack)},
{crossTogether(joe, jack)} $\rangle$

Here, $cost_{\mathcal{P}}(P_3) = 17$, and thus P_3 is admissible with respect to cost 17. This means that the Quick Bridge Crossing Problem has a positive answer. In fact, P_3 has least cost over all plans of length $l = 7$, and is thus an optimal 7-step plan. Moreover, P_3 has also least cost over all plans that emerge if we consider all plan lengths. Thus, P_3 is an optimal solution for the Quick Bridge Crossing Problem under arbitrary plan length.

Let $\mathcal{P}_x(l)$ denote the planning problem $\mathcal{P}_x$ for plan length l. Then we can write, $cost^*_{\mathcal{P}_{QBCP}(5)} = 19$, resp. $cost^*_{\mathcal{P}_{QBCP}(7)} = 17$

Secure Optimal Planning Now, let us consider the nondeterministic variant $\mathcal{P}_{BCPsec}$ (see p. 41) again: By substituting the action declarations for crossing again with the version in lines 32–34 above, we obtain a further elaboration of the problem, denoted as $\mathcal{P}_{QBCPsec}$. However, for $\mathcal{P}_{QBCPsec}$ we still can not find a better *secure* plan than P_2 shown on page 41 since any cross not involving Joe might fail. Obviously, $cost_{\mathcal{P}_{QBCPsec}(6)}(P_2) = 19$, and this plan is optimal, i.e. $cost^*_{\mathcal{P}_{QBCPsec}(6)} = 19$.

3.3 Complexity Analysis

We now turn to the computational complexity of planning in our language $\mathcal{K}$ and its extension $\mathcal{K}^c$. In this section, we present the results of a detailed study of major planning issues in the propositional case. Results for the case of general planning problems (with variables) may be obtained by applying suitable complexity upgrading techniques (cf. [GLV99]). In the general case, by well-known complexity results on logic programming, cf. [DEGV01], already evaluating the background knowledge is EXPTIME-hard, and the problems are thus provably intractable.

We call a planning domain *PD* (resp. planning problem $\mathcal{P}$) *propositional*, if all predicates in it have arity 0, and thus it contains no variables.

As for the results we will start off with main results for the core language $\mathcal{K}$ and derivation of these results. We will thereafter extend the results for admissible and optimal planning with action costs in $\mathcal{K}^c$.

3.3.1 Main Problems Studied

In our analysis, we consider the following three problems:

Optimistic Planning Decide, given a propositional planning problem $\mathcal{P} = \langle PD, q \rangle$, whether some optimistic plan exists, resp. find such a plan.

Security Checking Decide, given an optimistic plan $P = \langle A_1, \ldots, A_n \rangle$ for a propositional planning problem $\langle PD, q \rangle$, whether P is secure.

Secure Planning Decide, given a propositional planning problem $\mathcal{P}$, whether some secure plan exists, resp. find such a plan.

We remark here that the formulation of security checking is, strictly speaking, a *promise problem*, since it is *asserted* that P is an optimistic plan, which can not be checked in polynomial time in general (and thus legal inputs can not be recognized easily). However, the complexity results that we derive below would remain the same, even if P were not known to be an optimistic plan.

We will consider the problems from above under the following two restrictions:

1. **General vs. proper planning domains** Because of their underlying stable semantics, which is well-known intractable [MT91], causation rules in domain descriptions can express computationally intractable relationships between fluents. In fact, determining whether for a state s and a set of executable actions A in s some legal transition $\langle s, A, s' \rangle$ to any successor state s' exists in a planning domain PD is intractable in general, since it comprises checking whether a logic program has an answer set. For this reason, we pay special attention to the following subclass of planning domains.

   Definition 3.26. *We call a planning domain PD* proper *if, given any state s and any set of actions A, deciding whether some legal state transition $\langle s, A, s' \rangle$ exists is polynomial. A planning problem $\langle PD, q \rangle$ is* proper*, if PD is proper.*

   Proper planning domains are not plagued with intractability of deciding whether doing some actions will violate the dynamic domain axioms, even if they possibly have nondeterministic effects. In fact, we expect that in many scenarios, the domain is represented in a way such that if a set of actions qualifies for execution in a state, then performing these actions is guaranteed to reach a successor state. In such cases, the planning domain is trivially proper. This applies, for example, to the standard STRIPS formalism and many of its variants.

   Unfortunately, deciding whether a given planning domain is proper is intractable in general; we thus need syntactic restrictions for taking advantage of this (semantic) property in practice. For obtaining significant lower complexity bounds, we consider in our analysis a very simple class of proper planning domains.

Definition 3.27. *We call a planning domain $PD = \langle \Pi, AD \rangle$* plain*, if the background knowledge Π is empty, and AD satisfies the following conditions:*

1. *Executability conditions* `executable`... *refer only to fluents.*
2. *No default negation –neither explicit nor implicit through language extensions (such as inertia rules)– is used in the post-part of causation rules in the* `always`-*section.*
3. *Given that $\alpha_1, \ldots, \alpha_m$ are all ground actions, AD contains the rules*

   `nonexecutable` α_i `if` α_j. $\qquad 1 \leq i < j \leq m$
   `caused false after not` α_1, `not` α_2, ..., `not` α_m.

We call a planning problem $\mathcal{P} = \langle PD, q \rangle$ plain*, if PD is plain.*

The conditions ensure that every legal state transition $t = \langle s, A, s' \rangle$ must satisfy $|A| = 1$. Thus all optimistic and secure plans must be sequential.

As easily seen, in plain planning domains (which can be efficiently recognized), deciding whether for a state s and an action set A some legal state transition $t = \langle s, A, s' \rangle$ exists is polynomial, since this essentially reduces to evaluating a `not` -free logic program with constraints. Thus, plain planning domains are proper. Moreover, even deciding whether for a state s any legal state transition $t = \langle s, A, s' \rangle$ exists is polynomial, since the candidate space for suitable action sets A is small and efficiently computed. Furthermore, each legal state transition t in a plain planning domain PD is clearly determined, and thus PD is also deterministic. As discussed below, for many problems plain planning domains harbor already the full complexity of proper planning domains.

We remark that further, more expressive syntactic fragments of proper planning domains can be obtained by exploiting known results on logic programs which are guaranteed to have answer sets, such as stratified logic programs, or order-consistent and odd-cycle free logic programs [Fag94, Dun92]; the latter allow for expressing nondeterministic action effects. In particular, these results may be applied on the rules obtained from the dynamic causation rules by stripping off their *pre*-parts. We will consider more classes of proper planning domains, namely *false-committed*, *(mux-)stratified* and *serial* domains in more detail in Sections 4.3.4.1 and 4.3.4.2.

2. Fixed vs. arbitrary given plan length We analyze the impact of *fixing* the length i in the query $q = Goal\ ?\ (i)$ of $\langle PD, q \rangle$ to a constant. For *arbitrary* i, the length of an optimistic plan for $\langle PD, q \rangle$ can be exponential in the size of the string representing the number i (which, as usual, is represented in binary notation), and even exponential in the size of the string representing the whole input $\langle PD, q \rangle$. Indeed, it may be necessary to pass through an exponential number of different states until a state satisfying the goal is reached. For example, the initial state s_0 may describe the value $(0, \ldots, 0)$ of an n-bit counter, and the goal description might state that the counter has value $(1, \ldots, 1)$. Assuming an action repertoire which allows, in each state, to increment the value of the counter by 1, the shortest optimistic plan for this problems has $2^n - 1$ steps. (We leave the formalization of this problem in $\mathcal{K}$ as an illustrative exercise to the reader; a similar encoding can be found in Section 6.3.4.) This shows that storing

a complete optimistic plan in working memory requires exponential space in general. However, if i is fixed to a constant, then the representation size of an optimistic plan is linear in the size of $\langle PD, q\rangle$.

Analogously, as shown by Turner [Tur02], if i is bound by a polynomial in the size of $\langle PD, q\rangle$, then the representation size of an optimistic plan is polynomial.

We remark that general plan existence, leaving plan length *unspecified* becomes more tricky: For secure planning, the shortest secure plan might have double exponential length in the number of fluents: In worst case, we have to step through all belief states, i.e. all sets of states before reaching a goal. However, since we consider plan length i as a part of the input by definition of planning problems, the representation size of such a secure plan is still (single) exponential in the size of $\langle PD, q\rangle$.

3.3.2 Main Complexity Results

Our main results on the complexity of $\mathcal{K}$ are compactly summarized in Table 3.1, and can be explained as follows.

- As for Optimistic Planning, we can avoid exponential space for storing an optimistic plan $P = \langle A_1, \ldots, A_n\rangle$ by generating it *step by step*: we guess a legal initial state s_0, and subsequently, one by one, the legal transitions $\langle s_{i-1}, A_i, s_i\rangle$. Since storing one legal transition requires only polynomial workspace and NPSPACE = PSPACE, Optimistic Planning is in PSPACE. On the other hand, propositional STRIPS, which is PSPACE-complete [Byl94], can be easily reduced to planning in $\mathcal{K}$, where the resulting planning problem is plain and thus proper. For fixed plan length, the *whole* optimistic plan has linear size, and thus can be guessed and verified in polynomial time.

- In Security Checking, the optimistic plan $P = \langle A_0, \ldots, A_n\rangle$ to be checked is part of the input, so the binary representation of the plan length is not an issue here. If P is not secure, there must be a legal initial state s_0 and a trajectory executing the actions in $A_0, \ldots, A_i$ such that either the execution is stuck, i.e., no successor state s_i exists or the actions in A_i are not executable in s_i, or the goal is not fulfilled in the final state s_n. Such a trajectory can be guessed and verified in polynomial time with the help of an NP oracle; this places the problem in Π_2^P. The NP oracle is needed to cover the case where no successor state s_i exists, which reduces to checking whether a logic program has no answer set. In proper planning domains, existence of s_i can be

	plan length i in query q = *Goal* ? (i)	
planning domain *PD*	fixed (=constant)	arbitrary
general	NP / Π_2^P / Σ_3^P -complete	PSPACE / Π_2^P / NEXPTIME -complete
proper	NP / co-NP / Σ_2^P -complete	PSPACE / co-NP / NEXPTIME -complete

Table 3.1: Complexity Results for Optimistic Planning / Security Checking / Secure Planning in $\mathcal{K}$ (Propositional Case)

decided in polynomial time, which makes the use of an NP oracle obsolete and lowers the overall complexity from $\Pi_2^P = \text{co-NP}^{\text{NP}}$ to co-NP.

- In Secure Planning, the existence of a secure plan can be decided by composing algorithms for constructing optimistic plans and for security checking. Our membership proofs for deciding the existence of an optimistic plan actually (nondeterministically) construct such a plan, and thus we easily obtain upper bounds on the complexity of Secure Planning from the complexity of the combined algorithm, by using the security check as an oracle. In the case of arbitrary plan length, the use of a Π_2^P oracle can be eliminated by a more clever procedure, in which plan security is checked by inspecting all sets of states reachable after $0, 1, 2, \ldots$ steps of the plan. In order to represent sets of states exponential space is required. Even if the number of steps may be exponential, this does not lead to a further complexity blow up. Thus, Secure Planning is in NEXPTIME. On the other hand, even in plain planning domains, an exponential number of (exponentially long) candidate secure plans may exist, and the best we can do seems to be guessing a suitable one and verifying it. In fact, we will show NEXPTIME-hardness.

  Remarkably, if we modify the definition of $\mathcal{K}$ planning problems such that plan length is unspecified (i.e. we do not assume plan length to be part of the input, the maximum number of steps before loop though might even be double exponential, but as we only need to represent the current set of states reachable when stepwise guessing the plan, this places this general problem of secure plan existence with unspecified plan length in NEXPSPACE (=EXPSPACE). Haslum and Jonsson [HJ99] showed EXPSPACE-completeness for conformant planning with unspecified plan length in a closely related formalism. Interestingly, we can obtain a similar hardness proof from our NEXPTIME result above with minor modifications.

Effect of parallel actions The results in Table 3.1 address the case where parallel actions in plans are allowed. However, excluding parallel actions and considering only sequential plans does not change the picture drastically. In all cases, the complexity stays the same except for secure planning under fixed plan length, where Secure Planning is Π_2^P-complete in general and D^P-complete in proper planning domains (Theorem 3.9). Intuitively, this is explained by the fact that for a plan length fixed to a constant, the number of potential candidate plans is polynomially bounded in the input size of $\mathcal{P}$, and thus the guess of a proper secure candidate can be replaced by an exhaustive search, where it remains to check as a side issue the consistency of the domain (i.e. existence of some legal initial state), which is NP-complete in general (also for plain domains); see Theorem 3.9 below.

Effect of nondeterministic actions Our results also imply some conclusions on nondeterministic vs. deterministic planning domains. Interestingly, in proper planning domains, nondeterminism has no impact on the complexity for all problems considered, and we can conclude the same for Optimistic Planning as well as Secure Planning under arbitrary plan length. Furthermore, for proper planning problems even the combined restrictions of sequential plans and deterministic action outcomes do not decrease the complexity except for Secure Planning with fixed plan length, since the hardness results are obtained for plain planning problems, which guarantee these restrictions.

Implications for implementation The complexity results have important consequences for the implementation of $\mathcal{K}$ on top of existing computational logic systems, such as DLV, SMODELS [Nie99], propositional satisfiability (SAT) checkers, e.g. [MMZ+01, LA97, BS97, Zha97], or satisfiability solvers for Quantified Boolean Formulae (QBF-SAT checkers) [CGS98, Rin99b, FMS00]. Optimistic Planning under arbitrary plan length is not polynomially reducible to systems with capability of solving problems within the Polynomial Hierarchy, e.g. SAT checkers, CCALC, Smodels, or DLV, while it is feasible using QBF solvers. On the other hand, for fixed (and similarly, for polynomially bounded) plan length, Optimistic Planning can be polynomially expressed in all these systems. However, even in the case of fixed plan length and proper planning domains, Secure Planning is beyond the capability of systems having "only" NP expressiveness such as CCALC, Smodels, or SAT checkers, while it can be encoded in DLV (which has Σ_2^P expressiveness) and QBF solvers.

Even in the more restrictive plain planning domains, where Secure Planning is D^P-complete, the systems mentioned can not polynomially express Secure Planning in a single encoding. On the other hand, if we abandon properness, then also DLV is incapable of encoding Secure Planning (whose complexity increases to Σ_3^P-completeness). Nonetheless, Secure Planning is feasible in DLV using a two step approach as in the CPLAN [Giu00] system, where optimistic plans are generated as secure candidate plans and then checked for security; this check is polynomially expressible in DLV.

Secure planning under arbitrary plan length is provably intractable, even in plain domains. Since NEXPTIME strictly contains PSPACE, there is no polynomial time transformation to QBF solvers or other popular computational logic systems with expressiveness limited to PSPACE, such as traditional STRIPS planning.

Here, further restrictions are needed to lower complexity to PSPACE, such as a polynomial bound on the plan length in the input query (cf. [Tur02].

3.3.3 Derivation of Results

In this section, we show how the results discussed above are derived.

In the proofs of the lower bounds, the constructed planning problems $\mathcal{P} = \langle\langle \Pi, \langle D, R\rangle\rangle, q\rangle$ will always have empty background knowledge Π. Furthermore, the action and fluent declarations F_D and A_D, respectively, will be as needed for the R-part, and are not explicitly mentioned. That is, we shall only explicitly address R and q, while $\Pi = \emptyset$ and D are implicitly understood.

The following lemma on checking initial states and legal state transitions is straightforward from well-known complexity results for logic programming (cf. [DEGV97]). As mentioned above, we view the background knowledge independently from the remaining planning domain informally as a set of facts by requiring that Π has a unique answer set. We disregard the actual complexity of computing this model here by assuming polynomial computability of this answer set; as mentioned above, this is for instance true if the well-founded model of Π is total, which is guaranteed for stratified logic programs and other (syntactic) classes of programs.

Lemma 3.1. *Given a state s_0 (resp. a state transition $t = \langle s, A, s'\rangle$) and a propositional planning domain $PD = \langle \Pi, \langle D, R\rangle\rangle$, such that the unique answer set M of the logic program*

Π *can be computed in polynomial time, checking whether* s_0 *is a legal initial state (resp.,* t *is a legal state transition) is possible in polynomial time.*

of Lemma 3.1. Given M, the set of legal action and (positive and negative) fluent instances $\mathcal{L}_{PD}$ is easily computable in polynomial time, as well as the reduction PD^t. Checking whether s_0 is a legal initial state for $PD^{\langle\emptyset,\emptyset,s_0\rangle}$ amounts to checking whether s_0 is the least fix-point of a set of positive propositional rules, which is well-known polynomial. Overall, this means that checking whether s_0 is a legal initial state of PD is polynomial. From M, t, and PD^t, it can be easily checked in polynomial time whether A is executable wrt. s and, furthermore, whether s' is the minimal consistent set that satisfies all causation rules wrt. $s \cup A \cup M$ by computing the least fix-point of a set of positive rules and verifying constraints on it. Thus, checking whether t is a legal state transition is also polynomial in the propositional case. □

Corollary 3.2. *Given a sequence of state transitions* $T = \langle t_1, \ldots, t_n\rangle$*, where* $t_i = \langle s_{i-1}, A_i, s_i\rangle$ *for* $i = 1, \ldots, n$*, and a propositional planning domain* $PD = \langle\Pi, \langle D, R\rangle\rangle$ *such that the unique answer set* M *of the logic program* Π *can be computed in polynomial time, checking whether* T *is legal with respect to* PD *is possible in polynomial time.*

3.3.3.1 Optimistic Planning

From the preparatory results, we thus obtain the following result on Optimistic Planning.

Theorem 3.3. *Deciding whether for a given propositional planning problem* $\mathcal{P} = \langle PD, q\rangle$ *an optimistic plan exists is (i)* NP*-complete, if the plan length in* q *is fixed to a constant, and (ii)* PSPACE*-complete in general. The hardness parts hold even for plain* $\mathcal{P}$*.*

Proof. (i). The problem is in NP, since a trajectory $T = \langle t_1, \ldots, t_i\rangle$ where $t_j = \langle s_{j-1}, A_j, s_j\rangle$ for $j = 1, \ldots, i$, such that s_i satisfies the goal G in $q = G\ ?(i)$ can be guessed and, by Corollary 3.2, verified in polynomial time if i is fixed.

NP-hardness for plain $\mathcal{P}$ is shown by a reduction from the satisfiability problem (SAT). Let $\phi = C_1 \wedge \cdots \wedge C_k$ be a CNF, i.e. a conjunction of clauses $C_i = L_{i,1} \vee \cdots \vee L_{i,m_i}$ where the $L_{i,j}$ are classical literals over propositional atoms $X = \{x_1, \ldots, x_n\}$. We declare these atoms and a further atom '0' as fluents in D, and put into the `initially`-section I_R of the planning domain $PD = \langle\emptyset, \langle D, R\rangle\rangle$ the following constraints:

`total` x_j.	for all $x_j \in X$
`forbidden` $\neg.L_{i,1}, \ldots, \neg.L_{i,m_i}$.	$1 \leq i \leq k$
`caused` 0.	

Here, the first constraint effects the choice of a truth value for each fluent x_j, the second excludes choices which violate clause C_i, and the third adds '0' as a marker to the initial state. Clearly, PD has a legal initial state iff ϕ is satisfiable. Thus, an optimistic plan P exists for $\mathcal{P} = \langle PD, 0\ ?\ (0)\rangle$ iff ϕ is satisfiable. As $\mathcal{P}$ can easily be constructed from ϕ, the result follows.

(ii). A proof of membership in PSPACE follows from the discussion of the main results above at the beginning of Section 3.3.2 (note Lemma 3.1). We remark that the problem can be solved by a deterministic algorithm in polynomial workspace as follows. Similar as

in [Byl94], design a deterministic algorithm REACH(s, s', ℓ) which decides, given states s and s' and an integer ℓ, whether a sequence $t_1,\ldots,t_\ell$ of legal transitions $t_i = \langle s_{i-1}, A_i, s_i\rangle$ exists, where $s = s_0$ and $s' = s_\ell$, by cycling trough all states s'' and recursively solving REACH$(s, s'', \lfloor \ell/2 \rfloor)$ and REACH$(s'', s', \lceil (\ell+1)/2 \rceil)$. Then, the existence of an optimistic plan of length ℓ can be decided cyclic through all pairs of states s, s' and testing whether s is a legal initial state, s' satisfies the goal in given in q, and REACH(s, s', ℓ) returns true. Since the recursion depth is $O(\log \ell)$, and each level of the recursion needs only polynomial space, Lemma 3.1 implies that this algorithm runs in polynomial space.

For the PSPACE-hardness part, we describe how propositional STRIPS planning as in [Byl94] can be reduced to planning in $\mathcal{K}$, where the planning domain PD is plain.

Recall that in propositional STRIPS, a state description s is a total consistent set of propositional literals, and an operator op has a precondition $pc(op)$, an add-list $add(op)$, and a delete-list $del(op)$, which all are lists of propositional literals. The operator op can be applied in s if $pc(op) \subseteq s$ holds, and its execution yields the state $s' = (s \setminus del(op)) \cup add(op)$ (where s' must be consistent). Otherwise, the application of op on s is undefined. A goal γ, which is a set of literals, can be reached from a state s, if there exists a sequence of operators $op_1, \ldots, op_\ell$, where $\ell \geq 0$, such that $s_i = op_i(s_{i-1})$, for $i = 1, \ldots, \ell$, where $s_0 = s$, and $\gamma \subseteq s_\ell$ holds. Any such sequence is called a STRIPS-*plan* (of length ℓ) for s, γ. Given s, γ, a collection of STRIPS operators $op_1, \ldots, op_n$, and an integer $\ell \geq 0$, the problem of deciding whether some STRIPS-plan of length at most ℓ exists is PSPACE-complete [Byl94]. As easily seen, this remains true if we ask for a plan of length exactly ℓ (just introduce a dummy operation with empty precondition and no effects).

Each STRIPS operator op_i is easily modeled as action in language $\mathcal{K}$ using the following statements in the `always`-section, i.e. the C_R part of R:

`executable` op_i `if` $pc(op_i)$.	
`caused` L `after` op_i.	for each $L \in add(op_i)$
`caused` L `after` op_i, L.	for each $L \notin add(op_i) \cup del(op_i)$

The last rule is an inertia rule for the literals not affected by op.

The initial state s of a STRIPS planning problem can be easily represented using the following constraints in the `initially`-section, i.e. the I_R part of R:

`caused` L. for all $L \in s$

Finally, C_R contains the mandatory rules for unique action execution in a plain planning domain:

`nonexecutable` op_i `if` op_j.	$1 \leq i < j \leq n$
`caused false after not` op_1, `not` op_2, ..., `not` op_n.	

It is easy to see that for the planning problem $\mathcal{P} = \langle PD, q\rangle$ where $PD = \langle \emptyset, AD\rangle$ and $q = \gamma\ ?\ (\ell)$, some optimistic plan exists iff a STRIPS-plan of length ℓ for s, γ exists. Since $\mathcal{P}$ is constructible from the STRIPS instance in polynomial time, this proves the PSPACE-hardness part. □

By minor adaptions of the above proof ideas, we can also show the following:

Corollary 3.4. *For a propositional planning problem $\mathcal{P}$, computing an optimistic plan is* NPMV*-complete if the plan length in q is fixed.*

While membership is a direct consequence of the considerations above, hardness can be shown by a modification of the proof of Theorem 3.3 such that the plan itself corresponds to a valid truth assignment of a propositional formula. We refer to a similar construction showing NPMV-hardness in the proof of part (i) of Theorem 3.12 below.

On the other hand, if the plan length is not fixed, due to the possibly exponential length printing the solution alone might take exponential time, and thus the computation of such a plan is provably intractable.

3.3.3.2 Secure Planning

Secure Planning appears to be harder; already recognizing a secure plan is difficult.

Theorem 3.5. *Given a propositional planning problem $\mathcal{P} = \langle PD, q\rangle$ and an optimistic plan P for $\mathcal{P}$, deciding whether P is secure is (i) Π_2^P-complete in general and (ii) co-*NP*-complete, if $\mathcal{P}$ is proper.*[14] *Hardness in (i) and (ii) holds even for fixed plan length in q and sequential P, and if $\mathcal{P}$ in (ii) is moreover plain.*

Proof. The plan $P = \langle A_1, \ldots, A_i\rangle$ for $\mathcal{P}$ is not secure, if a trajectory $T = \langle t_1, \ldots, t_\ell\rangle$, where $t_j = \langle s_{j-1}, A_j, s_j\rangle$, for $j = 1, \ldots, \ell$ exists, such that either (a) $\ell = i$ and s_i does not satisfy the goal in q, or (b) $\ell < i$ and for no state s, the tuple $\langle s_\ell,\ A_{\ell+1}, s\rangle$ is a legal transition. A trajectory T of any length ℓ can, by Corollary 3.2, be guessed and verified in polynomial time. Condition (a) can be easily checked. Condition (b) can be checked by a call to an NP oracle in polynomial time. It follows that checking security is in co-$\mathrm{NP}^{\mathrm{NP}} = \Pi_2^P$ in general. If $\mathcal{P}$ is proper, condition (b) can be checked in polynomial time, and thus the problem is in co-NP. This shows the membership parts.

Π_2^P-hardness in case (i) is shown by a reduction from deciding whether a QBF $\Phi = \forall X \exists Y \phi$ is true, where X, Y are disjoint sets of variables and $\phi = C_1 \wedge \ldots \wedge C_k$ is a CNF over $X \cup Y$. It is well-known that this problem is Π_2^P-complete, cf. [Pap94]. Without loss of generality, we assume that ϕ is satisfied if all atoms in $X \cup Y$ are set to true.

We declare the atoms in $X \cup Y$ as fluents in D, and further add the propositional fluents `state0` and `state1`. The `initially`-section I_R for $AD = \langle D, R\rangle$ has the following constraints:

`total` x_j.	for all $x_j \in X$
`caused state0`.	

The `always`-section C_R of R contains the following rules. Suppose that $L_{i,1}, \ldots L_{i,n_i}$ are all literals over atoms from X which occur in C_i, and similarly that $K_{i,1}, \ldots K_{i,m_i}$ are all literals over atoms from Y that occur in C_i.

`total` y_j `after state0`.	for all $y_j \in Y$
`forbidden` $\neg.K_{i,1}, \ldots, \neg.K_{i,m_i}$ `after state0`, $\neg.L_{i,1}, \ldots, \neg.L_{i,n_i}$.	$1 \leq i \leq k$
`caused state1 after state0`.	

[14] We are grateful to Hudson Turner for pointing out that in a draft of [EFL+00a], a co-NP-upper bound as reported there obtains only if deciding executability of an action (leading to a new legal state) is in P, and that the complexity in the general case may be one level higher up in PH. In fact, we were mainly interested in such domains, which are covered by our notion of proper domains.

These rules generate $2^{|X|}$ legal initial states $s_0^1, \ldots, s_0^{2^{|X|}}$ w.r.t. $\langle \emptyset, AD \rangle$, which correspond 1-1 to the truth assignments to the atoms in X. Each such s_0^i contains precisely one of x_j and $-x_j$, for all $x_j \in X$, and the fluent `state0`. The totalization rule for y_j effects that each legal state s_1 following the initial state contains exactly one of y_j and $-y_j$. That is, s_1 must encode a truth assignment for Y. The `forbidden` statements check that the assignment to $X \cup Y$, given jointly by s_0^i and s_1, satisfies all clauses of ϕ. Furthermore, `state1` must be contained in s_1 by the last rule.

Let us introduce an action α, which is always executable. Then, the assumption on Φ implies that $T = \langle \langle s_0, A_1, s_1 \rangle \rangle$, where $s_0 = X \cup \{\texttt{state0}\}$, $A_1 = \{\alpha\}$, and $s_1 = X \cup Y \cup \{\texttt{state1}\}$, is a trajectory w.r.t. $PD = \langle \emptyset, AD \rangle$, and thus $P = \langle A_1 \rangle$ is an optimistic plan for the planning problem $\mathcal{P} = \langle PD, q \rangle$ where $q =$ `state1 ? (1)`. It is not hard to see that P is secure iff Φ is true. Since $\langle PD, q \rangle$ is easily constructed from Φ, this proves the hardness part of (i). The hardness part of (ii) is established by a variant of the reduction; we disregard Y (i.e., $Y = \emptyset$), and modify the rules as follows: `false` (after macro expansion) is replaced by `state1`, and the rule with effect `state1` is dropped. Note that the resulting planning domain is plain. Then, the plan $P = \langle A_1 \rangle$ is secure iff $\forall X \neg \phi$ is true, i.e., the CNF ϕ is unsatisfiable, which is co-NP-hard to check. □

For Secure Planning with fixed plan length, we obtain the following result.

Theorem 3.6. *Deciding whether a given propositional planning problem $\mathcal{P} = \langle PD, q \rangle$ has a secure plan is (i) Σ_3^P-complete, if the plan length in q is fixed, (ii) Σ_2^P-complete, if the plan length in q is fixed and $\mathcal{P}$ is proper. Hardness in (ii) holds even for deterministic and plain PD.*

Proof. Membership (i) and (ii): A trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$ of fixed length i that induces an optimistic plan $P = \langle A_1, \ldots, A_i \rangle$ can be guessed and verified in polynomial time (cf. Corollary 3.2), and by Theorem 3.5, checking whether P is secure is possible with a call to an oracle for Π_2^P in case (i) and for co-NP in case (ii). Hence, it follows that the problem is in Σ_3^P in case (i) and in Σ_2^P in case (ii).

For the hardness part of (i), we transform deciding the validity of a QBF $\Phi = \exists Z \forall X \exists Y \phi$, where X, Y, Z are disjoint sets of variables and $\phi = C_1 \ldots C_k$ is a CNF over $X \cup Y \cup Z$, which is Σ_3^P-complete [Pap94], into this problem. The transformation extends the reduction in the proof of Theorem 3.5.

We introduce, for each atom $z_i \in Z$, a fluent z_i and an action $\texttt{set}_{z_i}$ in D. The `initially`-section, i.e. the I_R part of R, contains the following constraints:

`total` x_j. for all $x_j \in X$
`caused state0`.

C_R contains the following rules. Suppose that $L_{i,1}, \ldots L_{i,n_i}$ are all literals over atoms from X that occur in C_i, and similarly that $K_{i,1}, \ldots K_{i,m_i}$ are all literals over atoms from $Y \cup Z$ that occur in C_i.

`executable` $\mathtt{set}_{z_i}$.	for all $z_i \in Z$
`caused` z_i `after state0,` $\mathtt{set}_{z_i}$.	for all $z_i \in Z$
`caused` $-z_i$ `after state0, not` $\mathtt{set}_{z_i}$.	for all $z_i \in Z$
`caused state1 after state0.`	
`total` y_j `after state0.`	for all $y_j \in Y$
`forbidden` $\neg.K_{i,1}, \ldots, \neg.K_{i,m_i}$ `after state0,` $\neg.L_{i,1}, \ldots, \neg.L_{i,n_i}$.	$1 \leq i \leq k$

Given these action descriptions, there are $2^{|X|}$ many legal initial states $s_0^1, \ldots, s_0^{2^{|X|}}$ for the emerging planning domain $PD = \langle \emptyset, AD \rangle$, which correspond 1-1 to the possible truth assignments to the variables in X and contain `state0`. Executing in these states s_0^i some actions A means assigning a subset of Z the value true. Every state s_1^i reached from s_0^i by a legal transition must, for each atom $\alpha \in Z \cup Y$, either contain α or $-\alpha$, where for the atoms in Z this choice is determined by A. Furthermore, s_1^i must contain the fluent `state1`.

It is not hard to see that an optimistic plan of the form $P = \langle A_1 \rangle$ (where $A_1 \subseteq \{\mathtt{set}_{z_i} \mid z_i \in Z\}$) for the goal `state1` exists w.r.t. $PD = \langle \emptyset, AD \rangle$ iff there is an assignment to all variables in $X \cup Y \cup Z$ such that the formula ϕ is satisfied. Furthermore, P is secure iff A_1 represents an assignment to the variables in Z such that, regardless of which assignment to the variables in X is chosen (which corresponds to the legal initial states s_0^i), there is some assignment to the variables in Y (i.e., there is at least some state s_1^i reachable from s_0^i, by doing A_1), such that all clauses of ϕ are satisfied; any such s_1^i contains `state1`. In other words, P is secure iff Φ is true.

Since PD is constructible from Φ in polynomial time, it follows that deciding whether a secure plan exists for $\mathcal{P} = \langle PD, q \rangle$, where $q =$ `state1 ? (1)`, is Σ_3^P-hard. This proves part (i).

For the hardness part of (ii), we modify the construction for part (i) by assuming that $Y = \emptyset$, and

- replace `false` in rule heads (after macro expansion) by `state1`;
- remove the rule for `state1` and the `total`-rules for y_j).

The resulting planning domain PD' is proper: since no causation rule in C_R contains default negation, for each transition $t = \langle s, A, s_1 \rangle$, the reduct ${PD'}^t$ coincides with ${PD'}^{\langle s, A, \emptyset \rangle}$, and thus existence of a a legal transition $\langle s, A, s_1 \rangle$ can be determined in polynomial time. Furthermore, $\langle s, A, s_1 \rangle$ is determined, and thus PD' is also deterministic. We have again $2^{|X|}$ initial states s_0^i, which correspond to the truth assignments to X. An optimistic plan for the goal `state1` of the form $P = \langle A_1 \rangle$, where $A_1 \subseteq \{\mathtt{set}_{z_i} \mid z_i \in Z\}$, corresponds to an assignment to $Z \cup X$ such that ϕ evaluates to *false*. The plan P is secure iff every assignment to X, extended by the assignment to Z encoded by A_1, makes ϕ false.

It follows that a secure plan for $\mathcal{P} = \langle PD', q \rangle$, where $q =$ `state1 ? (1)`, exists iff the QBF $\exists Z \forall X \neg \phi$ is true. Evaluating a QBF of this form is Σ_2^P-hard (recall that ϕ is in CNF). Since $\mathcal{P}$ is constructible in polynomial time, this proves Σ_2^P-hardness for part (ii). □

Again, the guess and check algorithms sketched in the membership proof above can be directly used for computing the respective plans, resulting in a transducer having secure fixed length plans as output. This results in the following corollary:

Corollary 3.7. *Given a propositional planning problem $\mathcal{P} = \langle PD, q\rangle$ computing a secure plan is (i) Σ_3^PMV-complete, if the plan length in q is fixed to a constant, (ii) Σ_2^PMV-complete, if the plan length in q is fixed to a constant and $\mathcal{P}$ is proper.*

As for hardness, we can use the QBF reduction from above, where the respective hard problem for Σ_3^PMV (or Σ_2^PMV, respectively) is computing an assignment for Z for the QBF $\exists Z \forall X \exists Y \phi$ (or $\exists Z \forall X \phi$, respectively).

Next, we consider Secure Planning under (i) arbitrary given plan length and (ii) without given plan length. Note that in $\mathcal{K}$ plan length is always considered to be given. However the following theorem shows, that things get significantly harder in case of (ii), i.e. if we modify the definition of $\mathcal{K}$ planning problems with this respect.

As mentioned above, we can build a secure plan step by step only if we know all states that are reachable after the steps $A_1, \ldots, A_i$ so far when the next step A_{i+1} is generated. Either we store these states explicitly, which needs exponential space in general, or we store the steps $A_1, \ldots, A_i$ (from which these states can be recovered) which also needs exponential space in the representation size of $\langle PD, q\rangle$. In any case, such a nondeterministic algorithm for generating a secure plan needs exponential time. The next result shows that NEXPTIME actually captures the complexity of deciding the existence of a secure plan.

Note that, if we consider plan length i is unspecified in the input, i.e. we want to know whether a secure plan of *any* length exists, the plan length for reaching a particular goal state may intuitively be 2-EXP (double exponential) in the representation size of $\langle PD, q\rangle$, as pinpointed above.

In fact, for any case we can show that in general there is no better solution than storing these sets of states explicitly, which needs exponential space in general, i.e.

Theorem 3.8. *Deciding whether a given propositional planning problem $\mathcal{P} = \langle PD, q\rangle$ has a secure plan is (i)* NEXPTIME*-complete for arbitrary given plan length i and (ii)* EXPSPACE*-complete if we do not consider plan length as part of the input. Hardness holds in both cases even for plain (and thus deterministic) $\mathcal{P}$.*

Proof. As for the membership part of (i), the size of a string representing a secure plan $P = \langle A_1, \ldots, A_i\rangle$ of length i for the query $q = Goal\ ?\ (i)$ is at most $O(i \cdot |PD|)$, which is single exponential in the sizes $|PD|$ and $\log i$ of the strings for PD and i, respectively. Hence, this string has size single exponential in the size of $\mathcal{P}$. We can thus guess and verify a secure plan P for $\mathcal{P}$ in (single) exponential time as follows:

1. Compute the set $\mathcal{S}_0$ of all legal initial states. If $\mathcal{S}_0 = \emptyset$, then P is not secure (in fact, no secure plan exists).
2. Otherwise, for each $j = 1, \ldots, i$, compute for each $s \in \mathcal{S}_{j-1}$ the set $\mathcal{S}_j(s) = \{s' \mid \langle s, A_j, s'\rangle$ is a legal transition$\}$, and halt if some $\mathcal{S}_j(s)$ is empty; otherwise, set $\mathcal{S}_j = \bigcup_{s \in \mathcal{S}_{j-1}} \mathcal{S}_j(s)$.
3. Finally, check whether the goal is satisfied in every $s \in \mathcal{S}_i$, and accept iff this is true.

Therefore, the problem is in NEXPTIME for given plan length. [15] As for (ii), if we assume that plan length is unspecified in the input, the above computation can still be done by

[15] i.e., the computation sketched here needs at most $2^n * 2^n = 2^{2n}$ polynomial checks whether $\langle s, A_j, s'\rangle$ is a legal transition.

guessing the A_j stepwise instead of guessing the whole plan at once requiring exponential space to store the current set of states S_j at each step j. Furthermore we need a counter for the plan length, to detect possible loops which are guaranteed after double exponential time. This counter (in binary representation) also needs exponential space. This places the general problem in NEXPSPACE=EXPSPACE.

The hardness part for (i) is shown by a generic Turing machine (TM) encoding. That is, given a nondeterministic TM M which accepts a language $\mathcal{L}_\mathcal{M}$ in exponential time and an input word w, we show how to construct a plain planning problem $\mathcal{P} = \langle PD, q\rangle$ in polynomial time which has a secure plan iff M accepts w. Roughly, the states in the set $\mathcal{S}_0$ of legal initial states encode the tape cells of M and their initial contents; the actions in a secure plan represent the moves of the machine, which change the cell contents, and lead to acceptance of w. While the idea is clear, the technical realization bears some subtleties.

The reduction is as follows. Without loss of generality, M halts on w in less than 2^{n^k} many steps, where $n = |w|$ is the length of the input and $k \geq 0$ is some fixed integer (independent of n), and M has a unique accepting state. We modify M such that it loops in this state once it is reached. The cells $C_0, C_1 \ldots, C_N$, where $N = 2^{n^k} - 1$, of the work tape of M (only those are relevant) are represented in different legal states of the planning domain. Initially, the cells $C_0, \ldots, C_{|w|-1}$ contain the symbols $w_0, w_1 \ldots, w_{|w|-1}$ of the input word w, and all other cells $C_{|w|}, \ldots, C_N$ are blank.

The computation of M on w is modeled by a secure plan $P = \langle A_1, \ldots, A_N\rangle$, in which each A_j contains a single action α_{τ_j} which models the transition of M from the current configuration of the machine to the next one. A configuration of M, given by the contents of the work tape, the position of the read-write (rw) head, and the current state of the machine, is described by legal knowledge states s_i, $0 \leq i \leq N$, such that s_i contains the symbol σ currently stored in C_i, the current position h of the rw-head, and the current state q of M; all this information is encoded using fluents.

The information to which cell C_i a legal knowledge state corresponds is given by literals $\pm i_1, \ldots, \pm i_{n^k}$, which represent the integer $i \in [0, N]$ in binary encoding, where i_j (resp. $-i_j$) means that the j-th bit of i is 1 (resp. 0). The position of the rw-head, $h \in [0, N]$, is represented similarly using further literals $\pm h_1, \ldots, \pm h_{n^k}$. Each symbol σ in the tape alphabet Σ of M is represented by a fluent p_σ. Similarly, each state q in the set Q of states of M is represented by a fluent p_q; in each legal knowledge state, exactly one p_σ and one p_q is contained. There are 2^{n^k} legal initial knowledge states, which uniquely describe the initial configuration of M, in which the rw-head of M is placed over C_0, M is in its initial state (say, q_1), and the work tape contains the input w.

The legal initial knowledge states s are generated using constraints which "guess" a value for each bit of i, initialize the contents of C_i with the right symbol p_σ, include $-h_j$ for all $j = 1, \ldots, n^k$ (i.e., set $h = 0$), and include q_1. More precisely, the `initially`-section, i.e. I_R of R, in $AD = \langle D, R\rangle$ is as follows:

`total` i_j.	for all $j = 1, \ldots, n^k$
`caused` $-h_j$.	for all $j = 1, \ldots, n^k$ % set $h = 0$
`caused` p_{w_0} `if` $-i_1, -i_2, \ldots, -i_{n^k}$.	% work tape position 0
`caused` p_{w_1} `if` $i_1, -i_2, \ldots, -i_{n^k}$.	% work tape position 1
$\vdots$	$\vdots$
`caused` $p_{w_{\lvert w\rvert-1}}$ `if` "code of $\lvert w\rvert - 1$".	% work tape position $\lvert w\rvert - 1$
`caused` $p_\sqcup$ `if not` $p_{\sigma_1}, \ldots,$`not` p_{σ_m}.	% rest of tape is blank
`caused` q_1.	% initial state is q_1

Here, the tape alphabet Σ is assumed to be $\Sigma = \{\sqcup, \sigma_1, \sigma_2, \ldots, \sigma_m\}$, where $\sqcup$ is the blank symbol.

The transition function of M is given by tuples $\tau = \langle \sigma, q, \sigma', d, q' \rangle$, which reads as follows: if M is in state q and reads the symbol σ at the current rw-head position h (i.e., C_h contains σ), then M writes σ' at the position h (i.e. into C_h), moves the rw-head to position $h + d$, where $d = \pm 1$, and changes to state q'. (Without loss of generality, we omit here modeling that the rw-head might remain in the same position.)

Such a possible transition τ is modeled using rules which describe how to change a current knowledge state s, which corresponds to the tape cell C_i, to reflect C_i in the new configuration of M. Informally, its constituents are manipulated as follows.

work tape contents For the case that $h = i$, i.e., the rw-head is at position i, a rule includes p_σ into the state. Otherwise, i.e., the rw-head is not at h, an inertia rule includes p_σ, where σ is the old contents of C_i, to the new knowledge state.

rw-head position The change of the rw-head position by ± 1, is incorporated by replacing h with $h \pm 1$. This is possible using a few rules, which simply realize an increment resp. decrement of the counter h. We assume at this point that M is well-behaved, i.e., does not move left of C_0.

state A rule includes $p_{q'}$ for the resulting state q' of M into the new knowledge state.

To implement this, we introduce for each possible transition $\tau = \langle \sigma, q, \sigma', d, q' \rangle$ of M an action α_τ, whose executability is stated in C_R as follows:

`executable` α_τ `if` p_q, p_σ, *h_atPosition_i*.
`executable` α_τ `if not` *h_atPosition_i*.

Here *h_atPosition_i* is a *fluent* atom, which indicates whether the rw-head position h is the index i of the cell C_i represented by the knowledge state.

Furthermore, several groups of rules are put in the `always`-section, i.e. C_R of R. The first group serves for determining the value of *h_atPosition_i*, using auxiliary fluents $e_1, \ldots, e_{n^k}$:

`caused` e_j `if` h_j, i_j.	for all $j = 1, \ldots, n^k$
`caused` e_j `if` $-h_j$, $-i_j$.	for all $j = 1, \ldots, n^k$
`caused` *h_atPosition_i* `if` $e_1, \ldots, e_{n^k}$.	

The execution of α_τ effects a change in the state and the contents of C_i:

caused $p_{\sigma'}$ after α_τ, $h\_atPosition\_i$.
caused p_σ after α_τ, p_σ, not $h\_atPosition\_i$. for all $\sigma \in \Sigma$
caused $p_{q'}$ after α_τ.

Depending on the value of d, different rules are added for realizing the move of the rw-head. Recall that, given the binary representation $x011\cdots1$ of an integer z, the binary representation of $z+1$ is $x100\cdots0$. The rules for $d=1$ are as follows.

caused h_1 after α_τ, $-h_1$.
caused h_2 after α_τ, $-h_2$, h_1.
caused $-h_1$ after α_τ, $-h_2$, h_1.
$\vdots$
caused h_{n^k} after α_τ, $-h_{n^k}$, $h_{n^k-1},\ldots,h_1$.
caused $-h_{n^k-1}$ after α_τ, $-h_{n^k}$, $h_{n^k-1},\ldots,h_1$.
$\cdots$
caused $-h_1$ after α_τ, $-h_{n^k}$, $h_{n^k-1},\ldots,h_1$.
caused h_ℓ after α_τ, h_ℓ, $-h_j$. where $1 \le j < \ell \le n^k$
caused $-h_\ell$ after α_τ, $-h_\ell$, $-h_j$. where $1 \le j < \ell \le n^k$

The last two rules serve for carrying the leading bits of i, which are not affected by the increment, over to the new knowledge state. (This could also be realized in a simpler way using `inertial` statements; however, recall that such rules are not allowed in plain domains.)

The rules for $d=-1$ are similar, with the roles of 0 and 1 interchanged:

caused $-h_1$ after α_τ, h_1.
caused $-h_2$ after α_τ, h_2, $-h_1$.
caused h_1 after α_τ, h_2, $-h_1$.
$\vdots$
caused $-h_{n^k}$ after α_τ, h_{n^k}, $-h_{n^k-1},\ldots,-h_1$.
caused h_{n^k-1} after α_τ, h_{n^k}, $-h_{n^k-1},\ldots,-h_1$.
$\cdots$
caused h_1 after α_τ, h_{n^k}, $-h_{n^k-1},\ldots,-h_1$.
caused h_ℓ after α_τ, h_ℓ, h_j. where $1 \le j < \ell \le n^k$
caused $-h_\ell$ after α_τ, $-h_\ell$, h_j. where $1 \le j < \ell \le n^k$

Further rules are added to C_R for carrying the cell index i over to the next knowledge state:

caused i_j after i_j. for all $j = 1,\ldots,n^k$
caused $-i_j$ after $-i_j$. for all $j = 1,\ldots,n^k$

Finally, the mandatory rules of a plain planning domain enforcing the execution of one and only one action in each transition are added to C_R.

As easily checked, all rules that we have introduced satisfy the syntactic restrictions for plain planning domains.

Without loss of generality, suppose now that $q_m \in Q$ is the unique accepting state of M. Then, a secure plan $P = \langle A_1,\ldots,A_\ell\rangle$ of length ℓ reaching the goal q_m corresponds to the

fact that M will, starting from the initial configuration, be in the accepting state q_m after executing the transitions $\tau_1, \ldots, \tau_\ell$, where $A_j = \{\alpha_{\tau_j}\}$, for $j = 1, \ldots, \ell$. By our assumption on M, we know that M can reach some accepting configuration within N steps iff it can reach an accepting configuration in exactly N steps. Thus, we have that M accepts the input w iff there exists some secure plan of length N for the goal q_m in the planning domain $PD = \langle \emptyset, AD \rangle$ where AD is from above. In other words, M accepts w within N steps iff the proper propositional planning problem $\mathcal{P} = \langle PD, q_m \ ? \ (N) \rangle$ has a secure plan.

As easily seen, $\mathcal{P}$ can be constructed in polynomial time from M and w. This proves NEXPTIME-hardness of deciding the existence of an exponential length secure plan, even under the restriction to plain planning problems and concludes the proof of (i).

EXPSPACE-hardness of (ii) follows immediately from the proof of (i) and from the fact that NEXPSPACE=EXPSPACE: we can take the proof of (i) as is with minor modifications; If we simply drop the restriction that M halts on w in less than 2^{n^k} many steps, the encoding above models a NEXPSPACE Turing machine by our assumption of an exponentially restricted worktape $C_0, C_1 \ldots, C_N$, for which deciding acceptance on w is EXPSPACE-hard. □

Finally, secure planning has lower complexity if the plan length is fixed and concurrent actions are not allowed.

Theorem 3.9. *Deciding whether a given propositional planning problem $\mathcal{P} = \langle PD, q \rangle$ has a secure sequential plan is (i) Π_2^P-complete, if q is fixed, and (ii) D^P-complete, if q is fixed and $\mathcal{P}$ is proper. The hardness part of (ii) holds even for plain $\mathcal{P}$.*

Proof. If the plan length i in the query $q = Goal \ ? \ (i)$ is fixed, the number of candidate sequential secure plans, given by $(a+1)^i$, where a is the number of actions in PD, is bounded by a polynomial.

A candidate $P = \langle A_1, \ldots, A_n \rangle$ is not a secure plan, if (a) no initial state s_0 exists, or (b) like in the proof of Theorem 3.5, a trajectory $T = \langle t_1, \ldots, t_\ell \rangle$, where $t_j = \langle s_{j-1}, A_j, s_j \rangle$, for $j = 1, \ldots, \ell$ exists, such that either (b.1) $\ell = i$ and s_i does not satisfy the goal in q, or (b.2) $\ell < i$ and for no state s, the tuple $\langle s_\ell, A_{\ell+1}, s \rangle$ is a legal transition. The test for (a) is in co-NP, while the test for (b) is in Σ_2^P in general and in NP if $\mathcal{P}$ is proper (cf. proof of Theorem 3.5). Note that (a) is identical for all candidates.

Thus, the existence of a sequential secure plan can be decided by the conjunction of a problem in NP and a disjunction of polynomially many instances of a problem in Π_2^P in case (i) and in co-NP in case (ii); since NP $\subseteq \Pi_2^P$ and both Π_2^P and co-NP are closed under polynomial disjunctions and conjunctions of instances (i.e., a logical disjunction resp. conjunction of instances can be polynomially transformed into an equivalent single instance), it follows that the problem is in Π_2^P in case (i) and in D^P in case (ii).

Π_2^P-hardness for case (i) follows from the reduction in the proof of Theorem 3.5. There, a secure, sequential plan exists for the query $1 \ ? \ (1)$ iff the plan $P = \langle \{\alpha\} \rangle$ is secure.

D^P-hardness for case (ii) is shown by a reduction from deciding, given CNFs $\phi = \bigwedge_{i=1}^{n} L_{i,1} \vee L_{i,2} \vee L_{i,3}$ and $\psi = \bigwedge_{j=1}^{m} K_{j,1} \vee K_{j,2} \vee K_{j,3}$ over disjoint sets of atoms X and Y, respectively, whether ϕ is satisfiable and ψ is unsatisfiable.

The `initially`-section, i.e. I_R of R contains the following constraints:

`total` x_j.	for all $x_j \in X$
`caused` $L_{i,1}$ `if` $\neg.L_{i,2}$, $\neg.L_{i,3}$.	for all $i = 1, \ldots, n$
`total` y_j.	for all $y_j \in Y$
`caused` f `if` $\neg.K_{i,1}$, $\neg.K_{i,2}$, $\neg.K_{i,3}$.	for all $i = 1, \ldots, m$

Obviously, these rules satisfy the conditions for a plain planning domain. Then, for the query $q = f\ ?\ (0)$, the only candidate for a sequential secure plan is the empty plan $P = \langle\rangle$. As easily seen, P is a secure plan for q iff ϕ is satisfiable (which is equivalent to the existence of some legal initial state) and ψ is unsatisfiable (which means that f is true in each initial state). This proves the hardness part of (ii). □

We conclude this section with remarking that the constructions in the proofs of the hardness parts of Theorem 3.5, items (i) and (ii) of Theorem 3.6, and item (i) of Theorem 3.9 involve planning problems that have length fixed to 1. For plan length fixed to 0, these problems have lower complexity (co-NP-completeness for the problems in Theorem 3.5 and D^P-completeness for the other problems).

3.3.4 Complexity of Planning with Action Costs

We will now extend our results to the language $\mathcal{K}^c$ where action costs come into play. Here, we will focus on the following questions:

Checking Well-Definedness: Decide whether a given action description is well-defined wrt. a given planning domain *PD*, resp. whether a given planning domain *PD* is well-defined.

Admissible Planning: Decide whether for a propositional planning problem $\mathcal{P}$ an admissible (optimistic/secure) plan exists wrt. a given cost value c, resp. find such a plan.

Optimal Planning: Find an optimal (optimistic/secure) plan for a given planning problem. [16]

As opposed to the previous section where we also investigated arbitrary plan length, in this section, we confine the discussion to the case of planning problems $\mathcal{P} = \langle PD, Q\ ?\ (l)\rangle$ which look for *polynomial length plans*, i.e. problems where the plan length l is fixed, resp. bounded by some polynomial in the size of the input.

3.3.4.1 Checking Well-Definedness

We start by considering checking well-definedness. For this problem, it is interesting to investigate the non-ground case, assuming that the background knowledge is already evaluated. This way we can assess the intrinsic difficulty of this task obtaining the following result.

[16] For optimal planning, *plan existence* is not a problem of interest since it obviously simply amounts to checking well-definedness together with plan existence for the cost-free case.

Theorem 3.10 (Complexity of checking well-definedness). *Given a $\mathcal{K}^c$ planning domain $PD = \langle \Pi, \langle D, R\rangle\rangle$ and the unique model M of Π, checking (i) well-definedness of a given action declaration d of the form (3.5) wrt. PD and (ii) well-definedness of PD are both Π_2^P-complete.*

Proof. Membership: As for (i), d is violated if it has a non-empty `costs`-part and a legal action instance $a = p(x_1, \ldots, x_n)$ such that either (1) there exist witness substitutions θ and θ' for a such that $\texttt{time}\theta = \texttt{time}\theta'$, $cost_\theta(a) = val(C\theta)$ and $cost_{\theta'}(a) = val(C\theta')$, and $val(C\theta) \neq val(C\theta')$, or (2) there is no witness substitution θ for a such that $cost_\theta(a) = val(C\theta)$ is an integer. Such an a can be guessed and checked, via a witness substitution, in polynomial time, and along with a also θ and θ' as in (1); note that, by definition, all variables must be substituted by constants from the background knowledge (including numbers), and so must be values for `time` if it occurs in $c_1, \ldots, c_k$. Given a, we can decide (2) with the help of an NP oracle. In summary, disproving well-definedness of d is nondeterministically possible in polynomial time with an NP oracle. Hence, checking well-definedness of d is in co-$\Sigma_2^P = \Pi_2^P$. The membership part of (ii) follows from (i), since well-definedness of PD reduces to well-definedness of all action declarations in it, and Π_2^P is closed under conjunctions.
Hardness: We show hardness for (i) by a reduction from deciding whether a Quantified Boolean Formula (QBF)

$$Q = \forall X \exists Y. c_1 \wedge \cdots \wedge c_k$$

where each $c_i = L_{i,1} \vee \cdots \vee L_{i,\ell_i}$, $i = 1, \ldots, k$, is a disjunction of literals $L_{i,j}$ on the atoms $X = x_1, \ldots, x_n$ and $Y = x_{n+1} \ldots, x_m$, is true. Without loss of generality, we may assume that each c_i contains three (not necessarily distinct) literals, which are either all positive or all negative.

We construct a planning domain PD and d as follows. The background knowledge, Π, is

```
bool(0). bool(1).
pos(1,0,0). pos(0,1,0). pos(0,0,1). pos(1,1,0). pos(1,0,1). pos(0,1,1). pos(1,1,1).
neg(0,0,0). neg(1,0,0). neg(0,1,0). neg(0,0,1). neg(1,1,0). neg(1,0,1). neg(0,1,1).
```

Here, `bool` declares the truth values 0 and 1. The facts $\texttt{pos}(\texttt{X}_1, \texttt{X}_2, \texttt{X}_3)$ and $\texttt{neg}(\texttt{X}_1, \texttt{X}_2, \texttt{X}_3)$ state those truth assignments to X_1, X_2, and X_3 such that the positive clause $X_1 \vee X_2 \vee X_3$ resp. the negative clause $\neg X_1 \vee \neg X_2 \vee \neg X_3$ is satisfied.

The rest of the planning domain PD consists of the single action declaration d of the form

$$\texttt{p}(\texttt{V}_1, ..., \texttt{V}_\texttt{n})\ \texttt{requires}\ \texttt{bool}(\texttt{V}_1), ...,\ \texttt{bool}(\texttt{V}_\texttt{n})\ \texttt{costs}\ 0\ \texttt{where}\ c_1^*, ..., c_k^*.$$

where

$$c_i^* = \begin{cases} \texttt{pos}(\texttt{V}_{\texttt{i},1}, \texttt{V}_{\texttt{i},2}, \texttt{V}_{\texttt{i},3}), & \text{if } c_i = x_{i,1} \vee x_{i,2} \vee x_{i,3}, \\ \texttt{neg}(\texttt{V}_{\texttt{i},1}, \texttt{V}_{\texttt{i},2}, \texttt{V}_{\texttt{i},3}), & \text{if } c_i = \neg x_{i,1} \vee \neg x_{i,2} \vee \neg x_{i,3}, \end{cases} \qquad i = 1, \ldots, k.$$

For example, the clause $c = x_1 \vee x_3 \vee x_6$ is mapped to $c^* = \texttt{pos}(\texttt{V}_1, \texttt{V}_3, \texttt{V}_6)$. It is easy to see that each legal action instance $a = p(b_1, \ldots, b_n)$ of d corresponds 1-1 to the truth assignment σ_a of X given by $\sigma_a(x_i) = b_i$, for $i = 1, \ldots, n$. Furthermore, a has a cost value defined (which is 0) iff the formula $\exists Y(c_1\sigma_a \wedge \cdots \wedge c_k\sigma_a)$ is true. Thus, d is well-defined wrt. PD iff Q is true. Since PD and d are efficiently constructible, this proves Π_2^P-hardness. □

Observe that in the ground case, checking well-definedness is much easier. Since no substitutions need to be guessed, the test in the proof of Theorem 3.10 is polynomial. Thus, by our assumption on the efficient evaluation of the background program, we obtain:

Corollary 3.11. *In the ground (propositional) case, checking well-definedness of an action description d wrt. a $\mathcal{K}^c$ planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, resp. of PD as a whole, is possible in polynomial time.*

We remark that checking well-definedness can be expressed as a planning task in $\mathcal{K}$, and also by a logic program; this will be considered in Section 4.3.7.

3.3.4.2 Admissible Planning

We now turn to computing admissible plans.

Theorem 3.12 (Complexity of admissible planning). *For polynomial plan lengths, deciding whether a given (well-defined) propositional planning problem $\langle PD, q \rangle$ has (i) some optimistic admissible plan wrt. to a given integer b is* NP*-complete, and finding such a plan is complete for* NPMV*, (ii) deciding whether $\langle PD, q \rangle$ has some secure admissible plan wrt. to a given integer b is Σ_3^P-complete, and computing such a plan is Σ_3^P*MV*-complete. Hardness holds in both cases for fixed plan length.*

Proof. Membership (i): The problems are in NP resp. NPMV, since if l is polynomial in the size of $\mathcal{P}$, any optimistic plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$ with a supporting trajectory $T = \langle t_1, \ldots, t_i \rangle$ for P can be guessed and, by Theorem 3.3, verified in polynomial time. Furthermore, $cost_{\mathcal{P}}(P) \leq b$ can be efficiently checked, since $cost_{\mathcal{P}}(P)$ is easily computed (all costs are constants).
Hardness (i): $\mathcal{K}$ is a fragment of $\mathcal{K}^c$, and each $\mathcal{K}$ planning problem can be viewed as the problem of deciding the existence of resp. finding an admissible plan wrt. cost 0. As was previously shown in Theorem 3.3, deciding existence of an optimistic plan for a given $\mathcal{K}$ planning problem is NP-hard for fixed plan length l; hence, it is also NP-hard for $\mathcal{K}^c$.

Whereas we have only sketched how to show hardness for actually finding such a plan in $\mathcal{K}$ (cf. Corollary 3.4), we will now give the full proof for admissible planning in $\mathcal{K}^c$: We show that finding an admissible optimistic plan is hard for NPMV by a reduction from the well-known SAT problem, cf. [Pap94], whose instances are CNFs $\phi = c_1 \wedge \cdots \wedge c_k$ of clauses $c_i = L_{i,1} \vee \cdots \vee L_{i,m_i}$, where each $L_{i,j}$ is a classical literal over propositional atoms $X = \{x_1, \ldots, x_n\}$.

Consider the following planning domain PD_ϕ for ϕ:

```
fluents:    x_1. ... x_n.  state0. state1.
actions:    c_1 costs 1. ... c_k costs 1.
            set_{x_1}. ... set_{x_n}.
initially:  total x_1. ... total x_n.
            caused state0.
always:     caused state1 after state0.
            executable c_1 if ¬.L_{1,1}, ..., ¬.L_{1,m_1}.
            forbidden after ¬.L_{1,1}, ..., ¬.L_{1,m_1}, not c_1.
            ...
```

```
executable c_k if ¬.L_{k,1}, ..., ¬.L_{k,m_k}.
forbidden after ¬.L_{k,1}, ..., ¬.L_{k,m_k}, not c_k.
executable set_{x_1} if x_1.  forbidden after x_1, not set_{x_1}.
...
executable set_{x_n} if x_n.  forbidden after x_n, not set_{x_n}.
```

The fluents x_i and `state0` and the `total` statements in the `initially`-section encode the candidate truth assignments. The subsequent statements force c_j to be executed iff the corresponding clause is violated by the truth assignment encoded in the initial state. The final pairs of `executable` and `forbidden` statements force actions $\texttt{set}_{x_i}$ to be executed iff the corresponding fluents x_i hold. This is because it is necessary to directly extract the computed truth assignments from the plan, since we are dealing with a function class. The fluent `state1` identifies the state at time 1.

Consider now the planning problem $\mathcal{P}_\phi = \langle PD_\phi, \texttt{state1?(1)}\rangle$. Clearly, each optimistic plan P for $\mathcal{P}$ corresponds to a truth assignment σ_P of X and vice versa, and $cost_{\mathcal{P}_\phi}(P)$ is the number of clauses violated by σ_P. Thus, the admissible optimistic plans for $\mathcal{P}_\phi$ wrt. cost 0 correspond 1-1 to the satisfying assignments of ϕ. Clearly, constructing $\mathcal{P}_\phi$ from ϕ is efficiently possible, as is constructing a satisfying truth assignment σ from a corresponding plan P (because of the actions $\texttt{set}_{x_i}$). This concludes the hardness proof.

Membership (ii): Since the security of each optimistic plan admissible wrt. cost k can be checked, by Theorem 3.5, with a call to a Π_2^P-oracle, membership in Σ_3^P resp. in Σ_3^PMV follows by analogous considerations as in (i) (where no oracle was needed).

Hardness (ii): For the decision variant, Σ_3^P-hardness is again immediately inherited from the Σ_3^P-completeness of deciding the existence of a secure plan of a problem in the language $\mathcal{K}$, with hardness even for fixed plan length (cf. Theorem 3.6 above). For the plan computation variant, we give a reduction from the following Σ_3^PMV-complete problem: An instance I is an open QBF

$$Q[Z] = \forall X \exists Y \Phi[X, Y, Z]$$

where $X = x_1, \ldots, x_l$, $Y = y_1, \ldots, y_m$, and $Z = z_1, \ldots, z_n$, respectively, and $\Phi[X, Y, Z]$ is (w.l.o.g.) a 3CNF formula over X, Y, and Z. The solutions $S(I)$ are all truth assignments over Z for which $Q[Z]$ is satisfied.

Suppose that $\Phi[X, Y, Z] = c_1 \wedge \ldots \wedge c_k$ where $c_i = c_{i,1} \vee c_{i,2} \vee c_{i,3}$. Now consider the following planning domain $PD_{Q[Z]}$ for $Q[Z]$, which is a variant of the planning domain given in the proof of Theorem 3.6.

```
fluents:   x_1. ... x_l. y_1. ... y_m. z_1. ... z_n.  state0. state1.
actions:   set_{z_1} costs 0. ... set_{z_n} costs 0.
initially: total x_1. ... total x_l.
           caused state0.
always:    caused state1 after state0.
           executable set_{z_1}. executable set_{z_2}.... executable set_{z_n}.
           caused x_1 after x_1.  caused −x_1 after −x_1.
           ...
           caused x_l after x_l.  caused −x_l after −x_l.
           total y_1 after state0. ... total y_m after state0.
```

```
caused z_1 after set_{z_1}.  caused −z_1 after not set_{z_1}.
...
caused z_n after set_{z_n}.  caused −z_n after not set_{z_n}.
forbidden ¬.C_{1,1}, ¬.C_{1,2}, ¬.C_{1,3} after state0.
...
forbidden ¬.C_{k,1}, ¬.C_{k,2}, ¬.C_{k,3} after state0.
```

There are $2^{|X|}$ many legal initial states $s^1, \ldots, s^{2^{|X|}}$ for $PD_{Q[Z]}$, which correspond 1-1 to the possible truth assignments to X and all these initial states contain `state0`. Starting from any initial state s^i, executing a set of actions represents a truth assignment to the variables in Z. Since all actions are always executable, there are $2^{|Z|}$ executable action sets $A_1, \ldots, A_{2^{|Z|}}$, which represent all truth assignments to Z.

For each pair s^i and A_j there exist $2^{|Y|}$ many successor state candidates $s^{i,1}, \ldots, s^{i,2^{|Y|}}$, which contain fluents according to the truth assignment to X represented by s^i, fluents according to the truth assignment to Z represented by A_j, and fluents according to a truth assignment to Y, and the fluent `state1`. Of these candidate states, only those satisfying all clauses in $\Phi[X, Y, Z]$ are legal, by virtue of the `forbidden` statements.

It is not hard to see that an optimistic plan of the form $P = \langle A_1 \rangle$ (where $A_1 \subseteq \{\mathtt{set}_{z_i} \mid z_i \in Z\}$) for the goal `state1` exists wrt. $PD_{Q[Z]}$ iff there is an assignment to all variables in $X \cup Y \cup Z$ such that the formula $\Phi[X, Y, Z]$ is satisfied. Furthermore, P is secure iff A_1 represents an assignment to the variables in Z such that, regardless of which assignment to the variables in X is chosen (corresponding to a legal initial state s^i), there is some assignment to the variables in Y such that all clauses of $\Phi[X, Y, Z]$ are satisfied (i.e., there is at least one state $s^{i,k}$ reachable from s^i by executing A_1); any such $s^{i,k}$ contains `state1`. In other words, P is secure iff $\Phi[X, Y, Z]$ is true. Thus, the admissible secure plans of $PD_{Q[Z]}$ wrt. cost 0, correspond 1-1 with the assignments to Z for which $Q[Z]$ is true.

Since $PD_{Q[Z]}$ is constructible from $\Phi[X, Y, Z]$ in polynomial time, it follows that computing a secure plan for $\mathcal{P} = \langle PD_{Q[Z]}, q \rangle$, where $q = \mathtt{state1}\,?\,(1)$, is Σ_3^PMV-hard. □

3.3.4.3 Optimal Planning

We finally address the complexity of computing optimal plans.

Theorem 3.13 (Complexity of optimal planning). *For polynomial plan lengths, (i) computing an optimal optimistic plan for $\langle PD, Q\,?\,(l) \rangle$ in $\mathcal{K}^c$ is $\mathrm{F}\Delta_2^P$-complete, and (ii) computing an optimal secure plan for $\langle PD, Q\,?\,(l) \rangle$ in $\mathcal{K}^c$ is $\mathrm{F}\Delta_4^P$-complete. Hardness holds in both cases even if the plan length l is fixed.*

Proof. Membership (i): Concerning membership, by performing a binary search on the range $[0, max]$ (where max is an upper bound on the plan costs for a plan of polynomial length l given by l times the sum of all action costs) we can find out the least integer v such that any optimistic plan P for $\mathcal{P}$ which is admissible wrt. cost v exists (if any optimistic plan exists); clearly, we have $cost_{\mathcal{P}}(P) = v$ and $cost^*_{\mathcal{P}} = v$, and thus any such plan P is optimal. Since max is single exponential in the representation size of $\mathcal{P}$, the binary search, and thus computing $cost^*_{\mathcal{P}}$, is, by Theorem 3.12, feasible in polynomial time with an NP oracle. Subsequently, we can construct an optimistic plan P such that $cost_{\mathcal{P}}(P) = cost^*_{\mathcal{P}}$

by extending a partial plan $P_i = \langle A_1, \ldots, A_i \rangle$, $i = 0, \ldots, l-1$ step by step as follows. Let $A = \{a_1, \ldots, a_m\}$ be the set of all legal action instances. We initialize $B_{i+1} := A$ and ask the oracle whether P_i can be completed to an optimistic plan $P = \langle A_1, \ldots, A_l \rangle$ admissible wrt. $cost^*_{\mathcal{P}}$ such that $A_{i+1} \subseteq (B_{i+1} \setminus \{a_1\})$. If the answer is yes, then we update $B_{i+1} := B_{i+1} \setminus \{a_1\}$, else we leave B_{i+1} unchanged. We then repeat this test for a_j, $j = 2, 3, \ldots, m$; the resulting B_{i+1} is an action set such that $P_{i+1} = \langle A_1, \ldots, A_i, A_{i+1} \rangle$ where $A_{i+1} = B_{i+1}$ can be completed to an optimistic plan admissible wrt. $cost^*_{\mathcal{P}}$. Thus, A_{i+1} is polynomial-time constructible with an NP oracle.
In summary, we can construct an optimal optimistic plan in polynomial time with an NP oracle. Thus, the problem is in $\mathrm{F}\Delta_2^P$.
Hardness (i): We show hardness for plan length $l = 1$ by a reduction from problem MAX WEIGHT SAT [Pap94], where an instance is a SAT instance $\phi = c_1 \wedge \cdots \wedge c_k$ as in the proof of Theorem 3.12.(i), plus positive integer weights w_i, where $i = 1, \ldots, k$. Then, $S(I)$ contains those truth assignments σ of X for which $w_{sat}(\sigma) = \sum_{i\,:\,c_i\sigma = \mathrm{true}} w_i$ is maximal.

To that end, we take the planning domain PD_ϕ as in the proof of Theorem 3.12 and modify the cost of c_i to w_i, for $i = 1, \ldots, k$, thus constructing a new planning domain PD_I. Consider now the planning problem $\mathcal{P}_I = \langle PD_I, \mathtt{state1?(1)} \rangle$. Since the actions c_j are the only actions with nonzero cost, any plan (corresponding to a truth assignment σ) will be associated with the sum of weights of violated clauses, $w_{vio}(\sigma) = (\sum_{i=1}^k w_i) - w_{sat}(\sigma)$. Since $\sum_{i=1}^k w_i$ is constant for I, minimizing $w_{vio}(\sigma)$ is equivalent to maximizing $w_{sat}(\sigma)$. Hence, there is a one-to-one correspondence between optimal optimistic plans of $\mathcal{P}_I$ (for which $w_{vio}(\sigma)$ is minimal) and maximal truth assignments for I. Furthermore, computing $\mathcal{P}_I$ from I and extracting a MAX-WEIGHT SAT solution from an optimal plan P is efficiently possible. This proves $\mathrm{F}\Delta_2^P$-hardness.
Membership (ii): The proof is similar to the membership proof of (i), but uses an oracle which asks for completion of a partial secure plan $P_i = \langle A_1, \ldots, A_i \rangle$ to a secure plan $P = \langle A_1, \ldots, A_l \rangle$ such that $A_{i+1} \subseteq (B_{i+1} \setminus \{a_j\})$ and P is admissible wrt. $cost^*_{\mathcal{P}}$, rather than of a partial optimistic plan. This oracle is, as easily seen, in Σ_3^P. Thus, computing an optimal secure plan is in $\mathrm{F}\Delta_4^P$.
Hardness (ii): We show hardness by a reduction from the following problem, which is $\mathrm{F}\Delta_4^P$-complete (cf. [Kre92]): Given an open QBF $Q[Z] = \forall X \exists Y \Phi[X, Y, Z]$ like in the proof of Theorem 3.12.(ii), compute the lexicographically first truth assignment of Z for which $Q[Z]$ is satisfied.

This can be accomplished by changing the cost of each action $\mathtt{set}_{z_i}$ in $PD_{Q[Z]}$ from 0 to 2^{n-i}, $i = 1, \ldots, n$. Let $PD'[Q[Z]]$ be the resulting planning domain. Since the cost of $\mathtt{set}_{z_i}$ (i.e., assigning z_i the value true) is greater than the sum of the costs of all $\mathtt{set}_{z_j}$ for $i + 1 \leq j \leq n$, an optimal secure plan for the planning problem $\langle PD'[Q[Z]], \mathtt{state1?(1)} \rangle$ amounts to the lexicographically first truth assignment for Z such that $Q[Z]$ is satisfied. Thus, $\mathrm{F}\Delta_4^P$-hardness of the problem follows. □

Chapter 4

Transformations to Answer Set Programming

The goal of this chapter is to introduce transformations from planning problems in $\mathcal{K}^c$ into logic programs such that the answer sets of these programs correspond to solutions, i.e. plans, of the problem at hand. In Section 3.3 we have already discussed that some planning problems, such as secure planning and secure optimal planning do in general not allow for such direct translations, even under fixed constant plan length. Whereas problems on the second level of the polynomial hierarchy can be expressed as answer set programs by use of disjunction, secure planning is located on the third level of the PH, even for plan length fixed to a constant, unless the domain at hand is proper. We will on the one hand introduce interleaved computations where direct encodings into a single program are too expensive or not feasible but also identify some syntactic subclasses of proper $\mathcal{K}$ planning problems where secure planning becomes easier.

The remainder of this chapter is organized as follows. First, we will recall general methods of problem solving in ASP by means of the so called "guess and check" paradigm [EFLP00], which is intuitively splitting a problem into (i) guessing possible solutions first and (ii) then checking these candidate solutions by appropriate constraints and rules. Many answer set programs reflect this structure and can be divided into guessing and checking rules, respectively. In the literature, this paradigm is also denoted as "generate and test" sometimes [Lif02].

We will furthermore introduce a general approach for integrating separate "guess" and "check" programs into a single disjunctive logic program which solves the overall problem by using meta-interpretation techniques. We will show the applicability of our method by means of two well-studied problems in Answer Set Programming, namely, QBFs with one quantifier alternation and "Strategic Companies", a problem from the business domain.

After that we will gradually apply the introduced ASP methods to planning in $\mathcal{K}^c$, providing translations for optimistic planning, optimistic optimal planning, secure checking, secure planning and secure optimal planning, respectively. Here, we will discuss direct encodings in a single program as well as interleaved computations by separate "guess" and "check" programs. The provided methods serve as a basis for the $\texttt{DLV}^{\mathcal{K}}$ planning system

which we present in Chapter 5.

4.1 General Methods for Problem Solving in ASP

Before discussing particular translations from planning problems in our formalism to logic programs, we will discuss general methods of problem solving in ASP by means of the so called "guess and check" paradigm. For instance, ASP allows for encoding hard problems on the second level of the polynomial hierarchy by use of disjunctive logic programming where solutions can be guessed and verified in polynomial time by means of an (co-)NP oracle. As pinpointed in Section 3.3 some relevant problems concerning planning lie within this class, e.g. checking plan security, or secure plan generation for proper planning domains with fixed plan length. On the other hand, by well-known complexity results (cf. Theorem 2.2), problems in this class are the "hardest" problems solvable with disjunctive logic programming.[17]

The idea behind the general discussion here and in the following section is that the methods presented can successfully be applied in translations from $\mathcal{K}$ and $\mathcal{K}^c$ planning problems to logic programming which we will show in Section 4.3 below.

4.1.1 Guess and Check - The NP Case

A general method for solving NP problems using answer set programming is given by the so called "guess and check" paradigm [EFLP00, LPF+02]: First a (possibly disjunctive) program is used to guess a set of candidate solutions, and then rules and constraints are added which eliminate unwanted solutions. DLPs allow for the formulation of problems in NP in a very intuitive way (e.g. solutions of 3-colorability, deterministic planning, etc.) if checking is easy (polynomial), such as checking whether no adjacent nodes have the same color, a course of deterministic actions reaches a certain goal, etc. For instance, given a graph as a set of facts of the form `node(`x`).` and `edge(`x,y`).` we can write a simple DLP which guesses and checks all possible 3-colorings as follows:

```
col(red,X)  v  col(green,X)  v  col(blue,X) :- node(X).   } Guess
:- edge(X,Y), col(C,X), col(C,Y).                         } Check
```

As easily seen, this program is head-cycle-free. Many NP-hard problems allow for such concise HEDLP encodings within the framework of Answer Set Programming (cf. Proposition 2.3). In particular, we will discuss such intuitive HEDLP encodings for solving optimistic planning problems with fixed plan length in Section 4.3.1, or checking plan security for certain classes of proper planning domains in Sections 4.3.4.1 and 4.3.4.2 below.

However, we also might want to express a problem which is complementary to some NP problem, and thus belongs to the class co-NP; it is widely believed that in general, not all such problems are in NP and hence not always a polynomial-size "certificate" checkable in polynomial time exists. One such problem is the property that a graph is *not* 3-colorable. Such properties p can be expressed by a HEDLP Π_p, where the property holds iff Π_p has

[17] We note that additional features for optimal answer set computation such as weak constraints in `DLV` further increase expressiveness to Δ_3^P, cf. [BLR00].

no answer set at all. Checks in co-NP typically occur as subproblems within more complex problems which have complexity higher than NP. An example for such a check is checking plan security in proper planning domains (cf. Section 3.3).

4.1.2 Guess and Check - The Σ_2^P Case

Encoding "guess and check" problems where the check itself is in co-NP but not known to be polynomial or in NP is not always obvious. For instance, secure plan generation for proper planning domains with fixed plan length as stated above involves such a co-NP check.

Similarly, checking plan security in general can be seen as such a problem, where a plan $P = \langle A_1, \ldots, A_n \rangle$ is *insecure* if we can guess a (partial) trajectory t which either (a) reaches a non-goal sate in step n (this check is polynomial) or (b) reaches a state s_j, $j < n$, such that no legal transition $\langle s_j, A_{j+1}, s_{j+1} \rangle$ exists (co-NP check).

Problems involving such a hard check can also be found in other areas like solving Quantified Boolean Formulae (QBFs) with one quantifier alternation (cf. Section 4.2.5.1), finding a set of strategic companies (cf. Section 4.2.5.2), computation of minimal update answer sets (cf. [EFST02]), etc.

This justifies the effort of spending some time on finding general strategies for solving such kinds of problems within the Answer Set Programming framework.

A simple, commonly used workaround is to write two programs:

(i) a normal LP or HEDLP Π_{guess} which guesses solution candidates, and

(ii) a HEDLP (equivalently, normal LP) Π_{check} which encodes the co-NP check,

and to proceed as follows:

1. compute, one by one, candidate solutions $S_1, S_2, \ldots$ as answer sets of Π_{guess} and
2. pipe each S_i as input to Π_{check};
3. output S_i if $\Pi_{check} \cup S_i$ has no answer set.

However, by the computational power of full disjunctive logic programs (Σ_2^P, cf. Section 2.2.2), we know that such problems can also be expressed by a single EDLP, Π_{solve}. In the following, we will show a generic method which can be used to automatically combine Π_{guess} and Π_{check} into such a single program.

In general, it is not clear how to combine Π_{guess} and Π_{check} into a *single* program Π_{solve} which solves the overall problem. Simply taking the union $\Pi_{guess} \cup \Pi_{check}$ does not work, and rewriting is needed. Theoretical results [EGM97, LPF+02] informally give strong evidence that for problems with Σ_2^P-complexity, it is required that Π_{check} (given as a normal logic program or a head-cycle-free disjunctive logic program) is rewritten into a disjunctive logic program Π'_{check}, where Π'_{check} "emulates" the inconsistency check for Π'_{check} as a minimal model check, which is co-NP-complete for disjunctive programs. The goal of such a rewriting is that the answer sets of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$ yield the solutions of the problem. This becomes even more complicated, as we will see, by the fact that Π'_{check} can not crucially rely on the use of negation.

These difficulties can make rewriting Π_{check} to Π'_{check} a formidable and challenging task. The question thus rises how to construct Π'_{check} from Π_{check} in general, and preferably in an automatic way.

4.2 Transforming co-NP Answer Set Checks to Stratified Disjunctive Programs

In this section, we will provide a generic rewriting method from Π_{check} to Π'_{check} by using a meta-interpreter approach. Thereafter, we will show how this method can be used to achieve the integrated encoding Π_{solve}.

In particular, we will proceed as follows:

1. We provide a polynomial-time transformation $tr(\Pi)$ from propositional head-cycle-free (extended) disjunctive logic programs (HEDLPs) Π to (general) disjunctive logic programs (DLPs), such that the following conditions hold:

   T1 Each answer set S' of the transformed program $tr(\Pi)$ corresponds to an answer set S of Π, such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ for some predicate $\texttt{inS}(\cdot)$.

   T2 If the original program has no answer sets, then $tr(\Pi)$ has exactly one designated answer set Ω, which is easily recognizable.

   T3 The transformation is of the form $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ is a factual representation of Π and Π_{meta} is a fixed meta-interpreter.

   T4 $tr(\Pi)$ is *modular* at the syntactic level, i.e., $tr(\Pi) = \bigcup_{r \in \Pi} tr(r)$ holds. Moreover, $tr(\Pi)$ returns a stratified DLP which uses negation only in its "deterministic" part, i.e. only concerning predicates which are deterministically defined by the representation of Π.

   We also describe optimizations and a transformation to positive DLPs, and show that in a precise sense, modular transformations to positive programs do not exist.

2. We show how to use $tr(\cdot)$ for integrating separate guess and check programs Π_{guess} and Π_{check} into a single DLP Π_{solve} such that the answer sets of Π_{solve} yield the solutions of the overall problem.

The results from this section further promote and advance the guess and check programming paradigm for ASP, and fill a gap by providing an automated construction for integrating guess and check programs. Note that integrated encodings may be direct subject to automated program optimization, which considers both the guess and check parts as well as their interaction in answer set engines like DLV or GNT; this is not possible for separate programs. Furthermore, our results complement recent results about meta-interpretation techniques in ASP, cf. [MR01, DST01, EFLP03].

4.2.1 Meta-Interpreter Transformation

As mentioned above, a rewriting of a given program Π_{check} to a program Π'_{check} for integrating the guess and check parts into a single program is not easy to accomplish in general. The problem is that the working of the answer set semantics is not easy to be emulated in Π'_{check}, since essentially we lack negation in Π'_{check}: Upon a "guess" S for an answer set of $\Pi_{solve} = \Pi_{guess} \cup \Pi'_{check}$, the reduct Π^S_{solve} is not-free. Recall that we want to consider only such guesses for S, representing an answer set of Π_{guess} such that Π_{check} has no answer set.

However, contrary to Π_{check}, there is no possibility to consider varying guesses for the value of negated atoms in Π'_{check} in combination with one guess for the negated atoms in Π_{guess} – all we have is a one in one combination. Thus, we will "mark" inconsistency of Π_{check} by some special answer set of Π'_{check}, which we will call Ω. On the other hand, for "emulating" negation we will have to make use of disjunction in Π'_{check} such that the non-existence of an answer set of Π'_{check} will be encoded as a minimal model check for Ω.

This leads us to consider an approach in which the program Π'_{check} is constructed by the use of meta-interpretation techniques [MR01, DST01, EFLP03]. There, the idea is that a program Π is represented by a set of facts, $F(\Pi)$, which is input to a fixed program Π_{meta}, the meta-interpreter, such that the answer sets of $\Pi_{meta} \cup F(\Pi)$ correspond to the answer sets of Π. Note that the meta-interpreters available are normal logic programs, and can not be used for our purposes for the reasons explained above. We thus have to construct a novel meta-interpreter which is essentially `not`-free but contains (non-head-cycle-free) disjunction.

To this end, we exploit the characterization of answer sets given for HEDLPs by Ben-Eliyahu and Dechter (Theorem 2.1 on page 10). More precisely, we will use Theorem 2.1 as a basis for a transformation from a given HEDLP Π to a DLP $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$ such that $tr(\Pi)$ fulfills the properties **T1** – **T4**.

4.2.1.1 Input Representation $F(\Pi)$

As input for our meta-interpreter Π_{meta}, which we provide in the next subsection, we choose the following representation $F(\Pi)$ of a propositional program Π.

We assume that each rule r has a unique name $n(r)$; for convenience, in the following we simply identify r with $n(r)$. For any rule $r \in \Pi$, we set up in $F(\Pi)$ the facts

`lit(h,`l`,`r`).`	`atom(`l`,`$\lvert l \rvert$`).`	for each literal $l \in Head(r)$,
`lit(p,`l`,`r`).`		for each literal $l \in Body^+(r)$,
`lit(n,`l`,`r`).`		for each literal $l \in Body^-(r)$.

While the facts for predicate `lit` obviously encode the rules of Π, the facts for predicate `atom` indicate whether a literal is classically positive or negative. We only need this information for head literals; this will be further explained below. Note that $F(\Pi)$ is a slightly enriched representation of Π and can be generated in linear time wrt. the size of Π.

4.2.1.2 Meta-Interpreter Π_{meta}

We construct our meta-interpreter program Π_{meta}, which in essence is a positive disjunctive program, in a sequence of several steps. They center around checking whether a guess for an answer set $S \subseteq Lit(\Pi)$, encoded by a predicate `inS(`$\cdot$`)`, is an answer set of Π by testing the criteria of Theorem 2.1. The steps of the transformation cast the various conditions there into rules of Π_{meta}, and also provide auxiliary machinery where needed.

Step 1 We add the following preprocessing rules:

```
1:    rule(L,R) :- lit(h,L,R), not lit(p,L,R), not lit(n,L,R).
2:    ruleBefore(L,R) :- rule(L,R), rule(L,R1), R1 < R.
3:    ruleAfter(L,R) :- rule(L,R), rule(L,R1), R < R1.
```

```
4:    ruleBetween(L,R1,R2) :- rule(L,R1), rule(L,R2), rule(L,R3),
                              R1 < R3, R3 < R2.
5:    firstRule(L,R) :- rule(L,R), not ruleBefore(L,R).
6:    lastRule(L,R) :- rule(L,R), not ruleAfter(L,R).
7:    nextRule(L,R1,R2) :- rule(L,R1), rule(L,R2), R1 < R2,
                           not ruleBetween(L,R1,R2).
8:    before(HPN,L,R) :- lit(HPN,L,R), lit(HPN,L1,R), L1 < L.
9:    after(HPN,L,R) :- lit(HPN,L,R), lit(HPN,L1,R), L < L1.
10:   between(HPN,L,L2,R) :- lit(HPN,L,R), lit(HPN,L1,R),
                             lit(HPN,L2,R), L<L1, L1<L2.
11:   next(HPN,L,L1,R) :- lit(HPN,L,R), lit(HPN,L1,R), L < L1,
                         not between(HPN,L,L1,R).
12:   first(HPN,L,R) :- lit(HPN,L,R), not before(HPN,L,R).
13:   last(HPN,L,R) :- lit(HPN,L,R), not after(HPN,L,R).
14:   hlit(L) :- rule(L,R).
```

Lines 1 to 7 fix an enumeration of the rules in Π from which a literal l may be derived, assuming a given order `<` on rule names (e.g. in `DLV`, the built-in lexicographic order; `<` can also be easily generated using guessing rules). Note that under the answer set semantics, we need only to consider rules where the literal l to prove does not occur in the body: A rule r cannot prove a literal l in its head if it is positively or negatively cyclic; for the positive case another rule r would be necessary to prove the body, making r redundant for the derivation of l, whereas, if l occurs negatively in the body, the r is not in the reduct wrt. to any set S containing l, and thus r can not contribute to prove l either.
Lines 8 to 13 fix enumerations of $Head(r)$, $Body^+(r)$ and $Body^-(r)$ for each rule. The final line 14 collects all literals that can be derived from rule heads. Note that the rules on lines 1–14 plus $F(\Pi)$ form a stratified program, which has a single answer set (cf. [Prz89, Prz91]).

Step 2 Next, we add rules which "guess" a candidate answer set $S \subseteq Lit(\Pi)$ and a total ordering `phi` on S corresponding with the function ϕ in condition 2 of Theorem 2.1. We will explain this correspondence in more detail below (cf. proof of Theorem 4.7).

```
15:   inS(L)  v  ninS(L) :- hlit(L).
16:   ninS(L) :- lit(pn,L,R), not hlit(L).          } for each pn ∈ {p,n}
17:   notok :- inS(L), inS(NL), L !=NL, atom(L,A), atom(NL,A).
18:   phi(L,L1)  v  phi(L1,L) :- inS(L), inS(L1), L < L1.
19:   phi(L,L2) :- phi(L,L1),phi(L1,L2).
```

Line 15 focuses the guess of S to literals occurring in some relevant rule head in Π; only these can belong to an answer set S, but no others (line 16). Line 17 then checks whether S is consistent, deriving a new distinct atom `notok` otherwise.

We see that the input facts for `atom` were only needed in order to check consistency. We could skip line 17 and the respective input facts if we prohibit classical negation in the input program Π.

Finally, line 18 guesses a strict total order `phi` on `inS` where line 19 guarantees transitivity; note that minimality of answer sets prevents that `phi` is cyclic, i.e., that `phi(L,L)` holds.

In the subsequent steps, we will check whether S and `phi` violate the conditions of Theorem 2.1 by deriving the distinct atom `notok` (considered in Step 5 below) in case, indicating that S is not an answer set or `phi` does not represent a proper function ϕ.

Step 3 Corresponding to condition 1 in Theorem 2.1, `notok` is derived whenever there is an unsatisfied rule by the following program part:

```
20:  allInSUpto(p,Min,R) :- inS(Min), first(p,Min,R).
21:  allInSUpto(p,L1,R) :- inS(L1), allInSUpto(p,L,R),
                           next(p,L,L1,R).
22:  allInS(p,R) :- allInSUpto(p,Max,R),last(p,Max,R).
23:  allNinSUpto(hn,Min,R) :- ninS(Min), first(hn,Min,R).
24:  allNinSUpto(hn,L1,R) :- ninS(L1), allNinSUpto(hn,L,R),
                             next(hn,L,L1,R).
25:  allNinS(hn,R) :- allNinSUpto(hn,Max,R), last(hn,Max,R).
26:  hasHead(R) :- lit(h,L,R).
27:  hasPBody(R) :- lit(p,L,R).
28:  hasNBody(R) :- lit(n,L,R).
29:  allNinS(h,R) :- lit(HPN,L,R), not hasHead(R).
30:  allInS(p,R) :- lit(HPN,L,R), not hasPBody(R).
31:  allNinS(n,R) :- lit(HPN,L,R), not hasNBody(R).
32:  notok :- allNinS(h,R), allInS(p,R), allNinS(n,R), lit(HPN,L,R).
```

(lines 23–25: for each $hn \in$ {h,n})

These rules compute by iteration over $Body^+(r)$ (resp. $Head(r)$, $Body^-(r)$) for each rule r, whether for all positive body (resp. head and default negated body) literals in rule r `inS` holds (resp. `ninS` holds) (lines 20 to 25). Here, empty heads (resp. bodies) are interpreted as unsatisfied (resp. satisfied), cf. lines 26 to 31. The final rule 32 fires exactly if one of the original rules from Π is unsatisfied.

Step 4 We derive `notok` whenever there is a literal $l \in S$ which is not provable by any rule r wrt. `phi`. This corresponds to checking condition 2 from Theorem 2.1.

```
33:  failsToProve(L,R) :- rule(L,R), lit(p,L1,R), ninS(L1).
34:  failsToProve(L,R) :- rule(L,R), lit(n,L1,R), inS(L1).
35:  failsToProve(L,R) :- rule(L,R), rule(L1,R), inS(L1), L1 !=L, inS(L).
36:  failsToProve(L,R) :- rule(L,R), lit(p,L1,R), phi(L1,L).
37:  allFailUpto(L,R) :- failsToProve(L,R), firstRule(L,R).
38:  allFailUpto(L,R1) :- failsToProve(L,R1), allFailUpto(L,R),
                          nextRule(L,R,R1).
39:  notok :- allFailUpto(L,R), lastRule(L,R), inS(L).
```

Lines 33 and 34 check whether condition 2.(a) or (b) are violated, i.e. some rule can only prove a literal if its body is satisfied. Condition 2.(d) is checked in line 35, i.e. r fails to prove l if there is some $l' \neq l$ such that $l' \in Head(r) \cap S$. Violations of condition 2.(e) are checked in line 36. Finally, lines 37 to 39 derive `notok` if all rules fail to prove some literal $l \in S$. This is checked by iterating over all relevant rules with $l \in Head(r)$ using the order from Step 1. Thus, condition 2.(c) is implicitly checked by this iteration.

Step 5 Whenever `notok` is derived, indicating a wrong guess, then we apply a saturation technique as in [EGM97, LRS01] to some other predicates, such that a canonical set Ω results. This set turns out to be an answer set iff no guess for S and ϕ works out, i.e., Π has no answer set. In particular, we saturate the predicates `inS`, `ninS`, and `phi` by the following rules:

```
40:  phi(L,L1) :- notok, hlit(L), hlit(L1).
41:  inS(L)    :- notok, hlit(L).
42:  ninS(L)   :- notok, hlit(L).
```

Intuitively, by these rules, any answer set containing `notok` is "blown up" to an answer set Ω containing all possible guesses for `inS`, `ninS`, and `phi`.

4.2.1.3 Answer Set Correspondence

Let $tr(\Pi) = F(\Pi) \cup \Pi_{meta}$, where $F(\Pi)$ is the input representation of Section 4.2.1.1 and Π_{meta} is the meta-interpreter from Section 4.2.1.2.

Clearly, $tr(\Pi)$ satisfies property **T3**, and as easily checked, $tr(\Pi)$ is modular. Moreover, strong negation does not occur in $tr(\Pi)$ and default negation only stratified. The latter is not applied to literals depending on disjunction; it thus occurs only in the deterministic part of $tr(\Pi)$, which means **T4** holds.

To establish **T1** and **T2**, we define the literal set Ω as the following uniquely determined set of ground literals.

Definition 4.1. *Let Π^i_{meta} be the set of rules in Π_{meta} established in Step $i \in \{1, \ldots, 5\}$. For any program Π, let $\Pi_\Omega = F(\Pi) \cup \bigcup_{i \in \{1,3,4,5\}} \Pi^i_{meta} \cup \{\texttt{notok.}\}$. Then, Ω is defined as the answer set of Π_Ω.*

Lemma 4.1. *Ω is well-defined and uniquely determined by Π.*

Proof. Follows immediately from the fact that Π_Ω is a stratified normal logic program without $\neg$ and constraints, which as well-known has a single (consistent) answer set (cf. [Prz89, Prz91]). □

Theorem 4.2. *For a given HEDLP Π the following holds for $tr(\Pi)$:*

1. *$tr(\Pi)$ always has some answer set, and $S' \subseteq \Omega$ for every answer set S' of $tr(\Pi)$.*
2. *S is an answer set of $\Pi \Leftrightarrow$ there exists an answer set S' of $tr(\Pi)$ such that $S = \{l \mid \texttt{inS}(l) \in S'\}$ and $\texttt{notok} \notin S'$.*
3. *Π has no answer set $\Leftrightarrow$ $tr(\Pi)$ has the unique answer set Ω.*

Proof. 1. The first part follows immediately from the fact that $tr(\Pi)$ has no constraints, no strong negation, and default negation is stratified; this guarantees the existence of at least one answer set S of $tr(\Pi)$ [Prz91]. Moreover, $S' \subseteq \Omega$ must hold for every answer set: after removing $\{\texttt{notok.}\}$ from Π_Ω and adding Π^2_{meta}, we obtain $tr(\Pi)$. Note that any rule in Π^2_{meta} fires wrt. S' only if all literals in its head are in Ω, and `inS`, `ninS`, and `phi` are elsewhere not referenced recursively through negation or disjunction. Therefore, increasing

S' locally to the value of Ω on `inS`, `ninS`, `phi`, and `notok`, and closing off thus increases it globally to Ω, which means $S' \subseteq \Omega$.
2. $\Rightarrow$ Assume that S is an answer set of Π. Clearly, then S is a consistent set of literals which has a corresponding set $S'' = \{\texttt{inS}(l) \mid l \in S\} \cup \{\texttt{ninS}(l) \mid l \in Lit(\Pi) \setminus S\}$ being one possible guess by the rules in lines 15 to 17 of Π_{meta}. Let now $\phi : Lit(\Pi) \to \mathbb{N}$ be the function from Theorem 2.1 for answer set S: Without loss of generality, we may assume two restrictions on this function ϕ:

- $\phi(l) = 0$ for all $l \in Lit(\Pi) \setminus S$ and $\phi(l) > 0$ for all $l \in S$.
- $\phi(l) \neq \phi(l')$ for all $l, l' \in S$.

Then, the function ϕ can be mapped to a total order over S `phi` such that

$$\texttt{phi}(l, l') \Leftrightarrow \phi(l) > \phi(l') > 0.$$

This relation `phi` fixes exactly one possible guess by the lines 18 and 19 of Π_{meta}.

Note that it is sufficient to define `phi` only over literals in S: Violations of condition 2.(e) have only to be checked for rules with $Body^+(r) \subseteq S$, as otherwise condition 2.(a) already fails. Obviously, condition 2.(e) of Theorem 2.1 is violated wrt. ϕ iff (a) `phi(Y,X)` holds for some `X` in the head of a rule with `Y` in its positive body or (b) if `X` itself occurs in its positive body. While (a) is checked in lines 36, (b) is implicit by definition of predicate `rule` (line 1) which says that a literal can not prove itself.

Given S'' and `phi` from above, we can now verify by our assumption that S is an answer set and by the conditions of Theorem 2.1 that (a) `notok` can never be derived in $tr(\Pi)$ and (b) S'' and `phi` uniquely determine an answer set S' of $tr(\Pi)$ of the form we want to prove. Here (a), can be argued by construction of Steps 3 and 4 of $tr(\Pi)$, where `notok` will only be derived if some rule is unsatisfied (Step 3) or there is a literal in S (i.e. S'') which fails to be proved by all other rules (Step 4). On the other hand, as for (b), uniqueness of S' follows from the fact that by fixing S'' and `phi`, the rules in Step 2 are "frozen" to a particular guess, and that the rest of the program is free of disjunction and unstratified default negation.

$\Leftarrow$ We can prove this by similar considerations: Assume that S' is an answer set of $tr(\Pi)$ not containing `notok`. Then by the guess of `phi` in Step 5 a function $\phi : Lit(\Pi) \to \mathbb{N}$ can be constructed by the implied total order of `phi` as follows: We number all literals $l \in S = \{l \mid \texttt{inS}(l) \in S'\}$ according to that order from 1 to $|S|$ and fix $\phi(l) = 0$ for all other literals in $Lit(\Pi)$. Again, by construction of Steps 3 to 5 and the assumption that $\texttt{notok} \notin S'$, we can see that S and the function ϕ constructed necessarily have to fulfill all the conditions of Theorem 2.1; in particular, line 17 guarantees consistency. Hence S is an answer set of Π.
3. $\Leftarrow$ Assume that Π has an answer set. Then, by the already proved Part 2 of the Theorem, we know that there exists an answer set $S' \subseteq \Omega$ of $tr(\Pi)$ such that $\texttt{notok} \notin S'$. By minimality of answer sets, Ω can not be an answer set of $tr(\Pi)$.

$\Rightarrow$ By Part 1 of Theorem 4.2, we know that $tr(\Pi)$ always has an answer set $S' \subseteq \Omega$. Assume that there is an answer set $S' \subsetneq \Omega$. We distinguish 2 cases: (a) $\texttt{notok} \notin S'$ and (b) $\texttt{notok} \in S'$. In case (a), proving Part 2 of this proposition, we have already shown that Π has an answer set; this is a contradiction. On the other hand, in case (b) the

final "saturation" rules in Step 5 "blow up" any answer set containing `notok` to Ω, which contradicts the assumption $S' \subsetneq \Omega$. □

The following proposition is not difficult to establish.

Proposition 4.3. *Given a ground program* Π*, the transformation* $tr(\Pi)$*, as well as the ground instantiation of* $tr(\Pi)$*, is computable in* LOGSPACE *(thus in polynomial time).*

Proof. Let $l = |Lit(\Pi)|$ be the number of distinct literals and $r = |\Pi|$ the number of rules in the original program Π. Then $n = l * r$ is (an upper bound of) the size of Π. A relevant part of the ground instantiation[18] of transformation $tr(\Pi)$ consists of

- $O(n)$ rules of constant size from the input representation $F(\Pi)$,
- $O(l^3 r + l r^3)$ rules of constant size from Step 1 [19],
- $O(l^3 + lr)$ rules of constant size from Step 2,
- $O(l^2 r)$ rules of constant size from Step 3,
- $O(l^2 r + l r^2)$ rules of constant size from Step 4, and
- $O(l^2)$ rules of constant size from Step 5.

These ground rules can obviously be generated efficiently in LOGSPACE by a naive grounding method using counters which iterate over the variables designating literals and rules of Π, respectively. Obviously, all steps of the translation $tr(\Pi)$ as well as their ground instantiations can be constructed by a fixed number of counters iterating over $Lit(\Pi)$ and the rules of Π where each of these counters (in binary representation) only needs logarithmic space.

Intelligent, efficient grounding methods such as used in `DLV` [ELM+98a, FLMP99, LPF+02] usually generate an even smaller ground instantiation. □

As noticed above, the transformation $tr(\Pi)$ uses default negation only stratified and in a deterministic part of the program[20]; we can easily eliminate it by computing the complement of each predicate accessed through `not` in the transformation and providing it in $F(\Pi)$ as facts; we then obtain a positive program. (The built-in predicates $<$ and $!=$ can be eliminated similarly if desired.) However, such a modified transformation is not modular. As shown next, this is not incidentally.

Proposition 4.4. *There is no modular transformation* $tr'(\Pi)$ *from HEDLPs to DLPs satisfying* **T1**, **T2** *and* **T3** *such that* $tr'(\Pi)$ *is a positive program.*

[18] Note that it is obviously not necessary to ground all variables over the whole Herbrand Universe, since we do not have to ground variables designating rules by constants designating literals and vice versa.

[19] Here, $l^3 r$ emerges from the instantiations of the rule in line 10 of Π_{meta}: This rule has three variables `L`, `L1`, `L2` which range over all literals in $Lit(\Pi)$, and one variable `R` which ranges over the rules in Π. A naive grounding, taking all possible substitutions, results in exactly $l^3 r$ ground rules. On the other hand, lr^3 emerges from the rule in line 4 which has one variable `L` which ranges over all literals in $Lit(\Pi)$, and three variables `R1`, `R2`, `R3` which range over the rules in Π. For the other steps, similar considerations apply, where the rules with the maximum number of distinct variables over r and l cause the highest impact on the size of the ground instantiation.

[20] Intuitively, default negation is only used to deterministically extend the information from the input representation $F(\Pi)$.

Proof. We assume that such a transformation exists and derive a contradiction. Let $\Pi_1 = \{$ `a :- not b.` $\}$ and $\Pi_2 = \Pi_1 \cup \{$`b.`$\}$. Then, $tr'(\Pi_2)$ has some answer set S_2. Since $tr'(\cdot)$ is modular, $tr'(\Pi_1) \subseteq tr'(\Pi_2)$ holds and thus S_2 satisfies each rule in $tr'(\Pi_1)$. Hence, S_2 contains some answer set S_1 of $tr'(\Pi_1)$ (by our assumption that $tr'(\cdot)$ is positive and by monotonicity of positive logic programs). $tr'(\Pi_1)$. Hence, S_2 contains some answer set S_1. By **T1**, we have that `inS(a)` $\in S_1$ must hold, and hence `inS(a)` $\in S_2$ as well. By **T1** again, it follows that Π_2 has an answer set S such that `a` $\in S$. But the single answer set of Π_2 is $\{$`b`$\}$, which is a contradiction. □

We remark that Proposition 4.4 remains true if **T1** is generalized such that the answer set S of Π corresponding to S' is given by $S = \{l \mid S' \models \Psi(l)\}$, where $\Psi(x)$ is a monotone query (e.g., computed by a normal positive program without constraints). Moreover, if a successor predicate `next(X,Y)` and predicates `first(X)` and `last(X)` for the constants are available (i.e., the universe of constants and rule names in Π is finite), then computing the negation of the non-input predicates accessed through `not` is feasible by a positive normal program, since such programs capture polynomial time computability by well-known results on the expressive power of Datalog [Pap85]; thus, negation of input predicates in $F(\Pi)$ is sufficient in this case.

Note that properties **T1** – **T4** for $tr(\cdot)$ are somehow orthogonal to the notion of polynomial faithful modular (PFM) transformations as introduced by Janhunen [Jan00, Jan01]. We recall the definition of PFM transformations:

Definition 4.2 (cf. [Jan00, Jan01]). *Given two classes of logic programs C and C' that are closed under union and the respective semantic operators Sem_C and $Sem_{C'}$, a translation function $Tr : C \to C'$ is*

- ***polynomial*** *if for all logic programs $\Pi \in C$, the time required to compute $Tr(\Pi) \in C'$ is polynomial in the size of Π.*
- ***faithful*** *if (i) for all logic programs $\Pi \in C$, the Herbrand base $HB_\Pi \subseteq HB_{Tr(\Pi)}$ and (ii) the models/interpretations in $Sem_C(\Pi)$ and $Sem_{C'}(Tr(\Pi))$ are in a one-to-one correspondence and coincide up to HB_Π.*
- ***modular*** *if (i) for all logic programs $\Pi_1, \Pi_2 \in C$, the translation $Tr(\Pi_1 \cup \Pi_2) = Tr(\Pi_1) \cup Tr(\Pi_2)$ and (ii) $C' \subset C$ implies that the translation $Tr(\Pi') = \Pi'$ for all logic programs $\Pi' \in C'$.*

Our transformation $tr(\cdot)$ relates to PFM transformations as follows: On the one hand, our translation is clearly polynomial and modular. Further, the requirement that $HB_\Pi \subseteq HB_{tr(\Pi)}$ and that answer sets coincide on HB_Π (i.e., on $Lit(\Pi)$) could be fulfilled by adding rules l `:-` `inS(`l`).` for every $l \in Lit(\Pi)$. The number of such rules is clearly polynomial and they could be added during input generation.

On the other hand, our condition **T2** clearly contradicts faithfulness, since Ω never has a corresponding answer set of Π. Moreover, condition **T1** is weaker than the one-to-one correspondence between the answer sets of Π and $tr(\Pi)$ required for faithfulness: In fact, whenever Π has positive cycles there might be several possible guesses for the ordering ϕ for an answer set S of Π in Theorem 2.1 reflected by different answer sets of $tr(\Pi)$. We illustrate this by a short example:

Example 4.1. Let Π be the program consisting of the following four rules

$r1$: `a :- b.` $r2$: `b :- a.` $r3$: `a.` $r4$: `b.`

Π has a single answer set $S = \{\mathtt{a}, \mathtt{b}\}$, whereas $tr(\Pi)$ has two answer sets
$S_1 = \{\ldots, \mathtt{inS(a)}, \mathtt{inS(b)}, \mathtt{phi(a,b)}, \ldots\}$ and
$S_2 = \{\ldots, \mathtt{inS(a)}, \mathtt{inS(b)}, \mathtt{phi(b,a)}, \ldots\}$, which intuitively reflect that the order of applications of rules $r1$ and $r2$ does not matter here, although they are cyclic. ◇

4.2.2 Optimizations

The above meta-interpreter Π_{meta} can be improved in several respects. We discuss here some modifications which, though not necessarily reducing the size of the ground instantiation, intuitively prune the search of an answer set solver applied to $tr(\Pi)$.

Give up modularity (OPT$_{mod}$) If we sacrifice modularity (i.e. that $tr(\Pi) = \bigcup_{r\in\Pi} tr(r)$), and allow that Π_{meta} partly depends on the input, then we can circumvent the iterations in Step 3 and part of Step 1. Intuitively, instead of iterating over the heads and bodies of all rules in order to determine whether these rules are satisfied, we add a single rule in $tr(\Pi)$ for each original rule r in Π firing `notok` whenever r is unsatisfied. So, we replace the rules from Step 3 by

$$\begin{aligned}\mathtt{notok} \texttt{ :- } &\mathtt{ninS}(h_1),\ \ldots,\ \mathtt{ninS}(h_l),\ \mathtt{inS}(b_1),\ \ldots,\ \mathtt{inS}(b_m),\\ &\mathtt{ninS}(b_{m+1}),\ \ldots\ \mathtt{ninS}(b_n).\end{aligned} \tag{4.1}$$

for each rule r in Π of form (2.1). These rules can be efficiently generated in parallel to $F(\Pi)$. Lines 8 to 13 of Step 1 then become unnecessary and can be dropped.

We can even refine this further, in that for any normal rule $r \in \Pi$ with $Head(r) = \{h\}$, which has a satisfied body, we can force the guess of h: we replace (4.1) by

$$\mathtt{inS}(h)\texttt{:- } \mathtt{inS}(b_1),\ \ldots,\ \mathtt{inS}(b_m), \mathtt{ninS}(b_{m+1}),\ \ldots\ \mathtt{ninS}(b_n). \tag{4.2}$$

In this context, adding facts `lit(p,`l`,`c`).`, $l \in Body^+(c)$ for a constraint $c \in \Pi$ is unnecessary: Since c only serves to "discard" unwanted models but cannot prove any literal, clearly the rule (4.1) is sufficient.

However, note that dropping `lit(n,`l`,`c`).` from $F(\Pi)$ requires more care: Assume that $l \in \mathtt{b}^-(c)$ only occurs in negative bodies of constraints but nowhere else in Π. Then, `ninS(`l`)` can not be derived by any of the lines 15 or 16 in Π_{meta} once `lit(n,`l`,`c`).` is dropped. Hence, rule (4.1) for c cannot "fire" in $tr(\Pi)$ and a violation of the constraint would not be detected in case `lit(n,`l`,`c`).` is dropped. Such critical l can be removed by simple preprocessing, though, by removing all $l \in \mathtt{b}^-(c)$ which do not occur in any rule head in Π. On the other hand, all literals $l \in \mathtt{b}^-(c)$ which appear in some other (non-constraint) rule r are not critical, since facts `lit(`hpn`,`l`,`r`).` ($hpn \in \{\mathtt{h},\mathtt{p},\mathtt{n}\}$) from this other rule will ensure that either line 15 or line 16 in Π_{meta} is applicable and therefore, either `inS(`l`)` or `ninS(`l`)` will be derived. Thus, after elimination of critical literals in constraints beforehand, we can safely drop the factual representation of constraints completely from $F(\Pi)$ (including `lit(n,`l`,`c`).` for the remaining negative literals).

Restrict to potentially applicable rules ($\mathbf{OPT}_{pa}$) We only need to consider literals in heads of *potentially applicable* rules. These are all rules with empty bodies, and rules where any positive body literal – recursively – is the head of another potentially applicable rule. This suggests the following definition:

Definition 4.3. *We call a set R of ground rules* potentially applicable, *if there exists an enumeration $\langle r_i \rangle_{i \in I}$ of R such that $Body^+(r_i) \subseteq \bigcup_{j=1}^{i-1} Head(r_j)$.*

Proposition 4.5. *Let Π be any ground HEDLP. Then there exists a unique maximal set $R \subseteq \Pi$ of potentially applicable rules, denoted by* $\mathrm{PA}(\Pi)$.

The set $\mathrm{PA}(\Pi)$ can be easily computed by adding a rule:

```
pa(r) :- lit(h,b1,R1), pa(R1), ..., lit(h,bm,Rm), pa(Rm).
```

for any rule r of the form (2.1) in Π. In particular, if $m = 0$ we simply add the fact `pa(r).` Finally, we change line 1 in Π_{meta} to:

```
rule(L,R) :- lit(h,L,R), not lit(p,L,R), not lit(n,L,R), pa(R).
```

such that only "interesting" rules are considered.

Optimize guess of order ($\mathbf{OPT}_{dep}$) We only need to guess and check the order ϕ for literals L, L' if they cyclicly depend on one another through positive recursion wrt. a set of literals S, i.e., they appear in the heads of rules within the same strongly connected component of the program wrt. S.[21] These mutual dependencies wrt. S are easily computed:

```
dep(L,L1) :- lit(h,L,R),lit(p,L1,R),inS(L),inS(L1).
dep(L,L2) :- lit(h,L,R),lit(p,L1,R),dep(L1,L2),inS(L).
cyclic :- dep(L,L1), dep(L1,L).
```

The guessing rules for ϕ (line 18 and 19) are then be replaced by:

```
phi(L,L1)  v  phi(L,L1) :- dep(L,L1), dep(L1,L), L < L1,cyclic.
phi(L,L2) :- phi(L,L1),phi(L1,L2), cyclic.
```

Moreover, we add the new atom `cyclic` also to the body of any other rule where `phi` appears (lines 36 and 40) to check `phi` only in case Π has *any* cyclic dependencies wrt. S.

Remark 4.1. *We remark that in principle, the rule defining atom `cyclic`, as well as adding this atom to the body of other rules seems superfluous at first sight. It is indeed, from the viewpoint of declarative logic programming. However, this additional machinery has proven successful experimentally, using DLV. Here, depending on the problem instance, DLV can possibly recognize that there are no `cyclic` dependencies, and eliminates the respective rules beforehand. For instance in our QBF encodings (cf. Section 4.2.5.1) Π_{check} never leads to cyclic dependencies among literals, making the additional rules for guessing and checking `phi` redundant in this example.*

[21] A similar optimization is used in [BED94], where $\phi : Lit(\Pi) \mapsto \{1, \ldots, r\}$ is only defined for a range r bound by the longest acyclic path in any strongly connected component of the program.

Note that it depends much (i) on the solver and grounding procedure, and (ii) the structure of the program Π whether these optimizations result in a performance gain. This is especially the case for **(OPT$_{pa}$)**: In particular, potential applicability of rules is already checked by intelligent grounding methods such as implemented in DLV in some cases, and normally in a program (written by a human) we might assume that all rules are possibly applicable. We refer to a more detailed discussion on the effects of optimizations in Chapter 8.

4.2.3 Integrating Guess and co-NP Check Programs

As pinpointed in Section 4.1.2 above, many problems on the second level of the polynomial hierarchy have an intuitive "guess and check" characteristics and can be divided into two problems Π_{guess}, which guesses some solution, and Π_{check} which encodes the co-NP check, If we assume that both problems are encoded as HEDLPs, and Π_{check} takes the solutions of Π_{guess} as input, any such solution with Π_{check} having no answer set shall be accepted.

We will now show how to integrate two such programs into a single program Π_{solve} by means of the translation $tr(\cdot)$ to solve the overall problem.

First, let us assume that the set $Lit(\Pi_{guess})$ is a Splitting Set [LT94] of $\Pi_{guess} \cup \Pi_{check}$, i.e., no head literal from Π_{check} occurs in Π_{guess}. Then, each rule r in Π_{check} is of the form

$$h_1 \texttt{ v } \cdots \texttt{ v } h_l \texttt{ :- } bc_1, \ldots, bc_m, \texttt{ not } bc_{m+1}, \ldots, \texttt{ not } bc_n \\ bg_1, \ldots, bg_p, \texttt{ not } bg_{p+1}, \ldots, \texttt{ not } bg_q. \tag{4.3}$$

where the bg_i are the body literals of r defined in Π_{guess}. We write $\texttt{body}_{guess}(r)$ for $bg_1, \ldots, bg_p, \texttt{ not } bg_{p+1}, \ldots, \texttt{ not } bg_q$. We will now redefine $tr(\Pi)$ wrt. these rules in Π_{check} in order to obtain a new check program Π'_{check} as mentioned in the beginning of this section.

Program Π'_{check} The program Π'_{check} contains the following rules and constraints:

1. The facts $F(\Pi_{check})$ in a conditional version $F'(\Pi_{check})$: For each rule $r \in \Pi_{check}$ of form (4.3),

   $\texttt{lit(h,}l,r\texttt{)} \texttt{ :- } \texttt{body}_{guess}(r).$ $\texttt{atom(}l,|l|\texttt{).}$ for each $l \in Head(r)$;
   $\texttt{lit(p,}bc_i,r\texttt{)} \texttt{ :- } \texttt{body}_{guess}(r).$ for each $i \in \{1, \ldots, m\}$;
   $\texttt{lit(n,}bc_j,r\texttt{)} \texttt{ :- } \texttt{body}_{guess}(r).$ for each $j \in \{m+1, \ldots, n\}$.

   Intuitively, these rules generate literals similar to the original factual input representation $F(\cdot)$ in dependence of Π_{guess}, since only those rules are "active" in $F'(\Pi_{check})$ where $\texttt{body}_{guess}(r)$ is satisfied.

2. All rules in $tr(\Pi) \setminus F(\Pi)$

3. Finally, a constraint

   ```
   :- not notok.
   ```

   This will eliminate all answer sets S of Π_{guess} such that $\Pi_{check} \cup S$ has an answer set.

The union of Π_{guess} and Π'_{check} then amounts to the desired integrated encoding Π_{solve}, which is expressed by the following result.

Theorem 4.6. *Given Π_{guess} and Π_{check}, the answer sets S_{solve} of Π_{solve}=$\Pi_{guess} \cup \Pi'_{check}$ correspond one-to-one with those answer sets S of Π_{guess} where $\Pi_{check} \cup S$ has no answer set.*

Proof. This result can be obtained from Theorem 4.7 and the Splitting Set Theorem [LT94]. We consider the proof for the original transformation $tr(\cdot)$; optimizations **(OPT$_{mod}$)**, **(OPT$_{dep}$)**, and **(OPT$_{mod}$)** will be considered separately, below. In what follows, let Q be a program over literals in $HB_{\Pi_{guess} \cup \Pi_{check} \cup \Pi_{meta}}$ and S be a consistent set of literals. Then $Q[S]$ denotes the program obtained from Q by eliminating every rule r such that $\mathtt{body}_{guess}(r)$ is false in S, and by removing $\mathtt{body}_{guess}(r)$ from the remaining rules. Notice that $\Pi_{check} \cup S$ and $\Pi_{check}[S] \cup S$ have the same answer sets.

We can rewrite Π_{solve} as

$$\Pi_{solve} = (\Pi_{guess} \cup F'(\Pi_{check})) \cup \Pi_{meta} \cup \{ \texttt{:- not notok.} \}$$

where $F'(\Pi_{check})$ denotes the modified factual representation for Π_{check}, given in item 1. of the definition of Π'_{check} above. By hypothesis on $\Pi_{guess} \cup \Pi_{check}$, the set $Lit(\Pi_{guess})$ is a Splitting Set for Π_{solve}. As easily seen, also $Lit(\Pi_{guess} \cup F'(\Pi_{check}))$ is a Splitting Set for Π_{solve}, and $Lit(\Pi_{guess})$ is also a splitting for $\Pi_{guess} \cup F'(\Pi_{check})$. Moreover, each answer set S of Π_{guess} is in one-to-one correspondence with an answer set S' of $\Pi_{guess} \cup F'(\Pi_{check})$. Then, $S' \setminus S = F(\Pi_{check}[S]) \cup A_S$ such that $F(\Pi_{check}[S])$ is the factual representation of $\Pi_{check}[S]$ in the transformation $tr(\Pi_{check}[S])$, and $A_S = \{\mathtt{atom}(l, |l|). \mid l \in Head(\Pi_{check}) \setminus Head(\Pi_{check}[S])\}$[22]. A_S is an additional set of facts emerging by construction of $F'(\Pi_{check})$: We added facts $\mathtt{atom}(l, |l|).$ unconditionally, i.e. for all head literals of $r \in \Pi_{check}$, not only for those r where $\mathtt{body}_{guess}(r)$ was satisfied. Thus, there might be superfluous such facts which we denote by the set A_S.

Now let S_{solve} be any (consistent) answer set of Π_{solve}. From the Splitting Set Theorem [LT94], we conclude that S_{solve} can be written as $S_{solve} = S \cup S_{check} \cup A_S$ where S and $S_{check} \cup A_S$ are disjoint, S is an answer set of Π_{guess}, and $S_{check} \cup A_S$ is an answer set of the program $\Pi'_S = (\Pi_{solve} \setminus \Pi_{guess})[S]$. Since $F'(\Pi_{check})$ is the only part of $\Pi_{solve} \setminus \Pi_{guess}$ where literals from $Lit(\Pi_{guess})$ occur, we obtain

$$\begin{aligned} \Pi'_S &= F(\Pi_{check}[S]) \cup A_S \cup \Pi_{meta} \cup \{\texttt{:- not notok.}\}, \text{ i.e.,} \\ &= tr(\Pi_{check}[S]) \cup A_S \cup \{\texttt{:- not notok.}\}. \end{aligned}$$

The additional facts A_S can be viewed as independent part of any answer set of Π'_S, since the answer sets of Π'_S are the sets $T \cup Lit(A_S)$ where T is any answer set of $\Pi'_S \setminus A_S$; note that $T \cap Lit(A_S) = \emptyset$. Indeed, the only rule in Π'_S where the facts of A_S play a role, is line 17 of Π_{meta}. All ground instances of line 17 are of the following form:

$$\texttt{notok :- inS(}l\texttt{), inS(}nl\texttt{), } l \texttt{ != } nl\texttt{, atom(}l\texttt{,|}l\texttt{|), atom(}nl\texttt{,|}l\texttt{|).}$$

where l and nl denote two complementary literals. Let us assume that $\mathtt{atom}(l, |l|). \in A_S$ (or $\mathtt{atom}(nl, |l|). \in A_S$, respectively), i.e. intuitively that l (or nl) does not occur in the head of an "active" rule wrt. S. Then, in order for the above rule to fire, $\mathtt{inS}(l)$ and

[22] Here, for any program Π, we write $Head(\Pi) = \bigcup_{r \in \Pi} Head(r)$.

`inS(`nl`)` both have to be true. However, this can only be the case for literals l (or nl) occurring in a rule head of $\Pi_{check}[S]$ (backwards, by the rules in line 15, 14 and 1 of Π_{meta} and by definition of Π'_{check}), which contradicts our assumption that `atom(`l`,|`l`|).` $\in A_S$ or `atom(`nl`,|`l`|).` $\in A_S$. Therefore, the additional facts of A_S do not affect the rule in line 17 and consequently Π'_S has an answer set if and only if $\Pi'_S \setminus A_S$ has an answer set and these answer sets coincide on $Lit(\Pi'_S) \setminus Lit(A_S)$.

By Theorem 4.7, we know that (i) $tr(\Pi_{check}[S])$ always has an answer set, and (ii) $tr(\Pi_{check}[S])$ has any answer set containing `notok` if and only if $\Pi_{check}[S]$ has no answer set. However, the constraint `:- not notok.` only allows for answer sets of Π'_S containing `notok`. Hence, $\Pi'_S \setminus A_S$ has an answer set S_{check} if and only if $\Pi_{check}[S]$ has no answer set, or equivalently, $\Pi_{check} \cup S$ has no answer set.

Conversely, suppose S is an answer set of Π_{guess} such that $\Pi_{check} \cup S$ has no answer set; equivalently, $\Pi_{check}[S]$ has no answer set. By Theorem 4.7, we know that $tr(\Pi_{check}[S]) = F(\Pi_{check}[S]) \cup \Pi_{meta}$ has a unique answer set S_{check}, and S_{check} contains `notok`. Hence, also the program $Q_S = F(\Pi_{check}[S]) \cup \Pi_{meta} \cup \{$`:- not notok.`$\}$ has the unique answer set S_{check}. On the other hand, since S is an answer set of Π_{guess} and $Lit(\Pi_{guess})$ is a Splitting Set for Π_{solve}, for each answer set S'' of the program $\Pi'_S = (\Pi_{solve} \setminus \Pi_{guess})[S]$, we have that $S \cup S''$ is an answer set of Π_{solve}. However, $\Pi'_S = Q_S \cup A_S$; hence, $S'' = S_{check} \cup A_S$ must hold and $S_{solve} = S \cup S_{check} \cup A_S$ is the unique answer set of Π_{solve} which extends S. This proves the result. □

The optimizations $(\mathbf{OPT}_{pa})$ and $(\mathbf{OPT}_{dep})$ in Section 4.2.2 still apply. However, concerning $(\mathbf{OPT}_{mod})$, the following modifications are necessary:

(1) For any rule r, in analogy to in the input representation $F'(\Pi_{check})$, rules (4.1) and (4.2) have to be extended by adding $\mathtt{body}_{guess}$`(r)` as defined above.

(2) We mentioned above that the factual representation of literals in $Body(c)$ may be skipped for constraints c. This now only applies to literals in $Body^+(c)$; the rule `lit(n,`l, c`) :-` $\mathtt{body}_{guess}(c)$`.` for $l \in Body^-(c)$ may no longer be dropped in general, as shown by the following example.

Example 4.2. Let $\Pi_{guess} =$ { `g  v -g.` } and $\Pi_{check} =$ { $r1$`:  x :- g.`, $r2$`:  :- not x.` } The "input" representation of Π_{check} wrt. optimization $(\mathbf{OPT}_{mod})$, i.e., the variable part of Π'_{check}, now consists of:

```
lit(h,x,r1) :- g.  lit(n,x,r2).  inS(x) :- g.  notok :- ninS(x).
```

where the latter two rules correspond to the conditional versions of rules (4.2) and (4.1). If we now assume that we want to check answer set $S =$ { `-g`} of Π_{guess}, it is easy to see that Π_{check} has no answer set for S. Therefore, S should be represented by some answer set which can be found by our integrated encoding Π_{solve}. Now assume that `lit(n,x,`$r2$`).` is dropped and we proceed generating the integrated encoding as outlined above wrt. to $(\mathbf{OPT}_{mod})$. Since `g` $\notin S$ and we have dropped `lit(n,x,`$r2$`).`, the "input" representation of Π_{check} for S comprises only the final rule `notok :- ninS(x).` However, this rule can never fire because neither line 15 nor line 16 of Π_{meta} can ever derive `ninS(c)`. Therefore, also `notok` can not be derived and the integrated check fails. On the other hand, `lit(n,x,`$r2$`).` suffices to derive `ninS(x)` via line 16 of Π_{meta}, so `notok` can be derived and the integrated check works as intended. ◇

In certain cases, we can still drop $l \in \mathtt{b}^-(c)$. For example, if l occurs in the head of a rule r with $\mathtt{body}_{guess}(r) = \emptyset$: In this case `lit(h,l,r)` will always be added to the program Π_{solve} and either `inS(`l`)` or `ninS(`l`)` will be contained in any answer set of Π'_{check}.

Remark 4.2. *For the encoding of QBFs in Section 4.2.5.1, considerations upon negative literals in constraints in (OPT_{mod}) do not play a role because all literals in the constraints of QBF_{check} are positive. On the other hand, in the encoding of Strategic Companies from Section 4.2.5.2, depending on the concrete problem instance, SC_{check} contains critical constraints c, where* `not strat1(·)` *occurs, such that* `lit(n,"strat1(·)",c)` *must not be dropped here.*

Similarly, for the general translations of planning problems wrt. the integrated encodings introduced in Sections 4.3.3 and 4.3.5 there might be cases where we might not drop negative literals from constraints in the "input" representation offhand.

Remark 4.3. *Note that the translations defined here are only applicable for* ground *programs Π_{guess} and Π_{check}. So, when solving such problems given non-ground programs Π_{guess} and Π_{check}, respectively, we have to ground the problems* before *applying these translations. However, as grounding in general can be exponential, our transformations, though polynomial on ground programs can add a significant blowup with respect to the original (non-ground) program. Investigating how the presented transformations can partly be lifted to non-ground problems is part of future research.*

4.2.4 Integrating Guess and NP Check Programs

We remark that integrating a guess program Π_{guess} and a check program Π_{check}, which succeeds iff $\Pi_{check} \cup S$ has *some* answer set, is easy. Given that Π_{check} is a HEDLP again, this amounts to integrating a check which is in NP. After a rewriting to ensure the Splitting Set property (if needed), simply take $\Pi_{solve} = \Pi_{guess} \cup \Pi_{check}$; its answer sets correspond on the predicates in Π_{guess} to the desired solutions.

4.2.5 Applications

We now exemplify the use of our transformation for two Σ_2^P-complete problems, which are well-studied in Answer Set Programming. One is about Quantified Boolean Formulae (QBFs) with one quantifier alternation, another about a business problem on Strategic Companies [LPF+02]; both problems involve co-NP-complete solution checking.

Further examples from the area of planning will be given in the Sections 4.3.3 and 4.3.5. Remarkably, compared to the QBF and Strategic Companies problems from this Section, no previous ad hoc encodings existed for these planning problems but, as will be shown, our general method can be applied with minor modifications there as well.

Note that our method is applicable to *any* check encoded by inconsistency of a HEDLP; co-NP-hardness is not a prerequisite.

4.2.5.1 Quantified Boolean Formulae

We now exemplify the use of our translation for solving Quantified Boolean Formulae (QBFs) with one quantifier alternation.

Given a QBF $F = \exists x_1 \cdots \exists x_m \forall y_1 \cdots \forall y_n \, \Phi$, where $\Phi = c_1 \vee \cdots \vee c_k$ is a propositional formula over $x_1, \ldots, x_m, y_1, \ldots, y_n$ in disjunctive normal form, i.e. each $c_i = a_{i,1} \wedge \cdots \wedge a_{i,l_i}$ and $|a_{i,j}| \in \{x_1, \ldots, x_m, y_1 \ldots, y_n\}$, compute the assignments to the variables $x_1, \ldots, x_m$ which witness that F evaluates to true.

Intuitively, this problem can be solved by "guessing and checking" as follows:

(QBF_{guess}) Guess a truth assignment for the variables $x_1, \ldots, x_m$.

(QBF_{check}) Check whether this (fixed) assignment satisfies Φ for all assignments of variables $y_1, \ldots, y_n$.

Both parts can be encoded by very simple HEDLPs (or similarly by normal programs):

QBF_{guess} :

$x_1 \ \mathtt{v} \ -x_1. \quad \ldots \quad x_m \ \mathtt{v} \ -x_m.$

QBF_{check} :

$y_1 \ \mathtt{v} \ -y_1. \quad \ldots \quad y_n \ \mathtt{v} \ -y_n.$

$\mathtt{:-} \ a_{1,1}, \ldots, a_{1,l_1}.$

$\vdots$

$\mathtt{:-} \ a_{k,1}, \ldots, a_{k,l_k}.$

Obviously, both programs are head-cycle-free and for every answer set S of QBF_{guess}, representing an assignment to $x_1, \ldots, x_m$, the program $QBF_{check} \cup S$ has no answer set iff every assignment for $y_1, \ldots, y_n$ satisfies formula Φ, because of the constraints. Intuitively, the constraints in QBF_{check} only accept "bad" assignments to variables $y_1, \ldots, y_n$ which make no disjunct true. If, conversely, there is no such "bad" assignment, then the candidate guess S represents a valid assignment to $x_1, \ldots, x_m$.

By the method sketched above, we can automatically generate a single program Π_{solve} integrating the guess and check programs. For a full encoding of a sample QBF example we refer to Appendix A.1. Note that the customary (but tricky) saturation technique to solve this problem (cf. [EGM97, LPF+02]) is fully transparent to the non-expert, who might come up with the two programs above easily.

4.2.5.2 Strategic Companies

Another Σ_2^P-complete problem is the strategic companies problem from [CEG97]. Briefly, a holding owns companies, each of which produces some goods. Moreover, several companies may have jointly control over another company. Now, some companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding after the transaction. A company is *strategic*, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. Guessing a strategic set, and checking its minimality, respectively, can be done by the following two programs:

$STRATCOMP_{guess}$:

```
strat(X) v -strat(X) :- company(X).
```

$STRATCOMP_{check}$:

```
strat1(X) v -strat1(X) :- strat(X).
```

```
:- prod_by(X,Y,Z), not strat(Y),
   not strat(Z).
:- contr_by(W,X,Y,Z), not strat(W),
   strat(X), strat(Y), strat(Z).
```

```
:- prod_by(X,Y,Z), not strat1(Y),
   not strat1(Z).
:- contr_by(W,X,Y,Z), not strat1(W),
   strat1(X), strat1(Y), strat1(Z).
smaller :- -strat1(X).
:- not smaller.
```

Here `strat`(C) means that C is strategic, `prod_by`$(P, C1, C2)$ that product P is produced by companies $C1$ and $C2$, and `contr_by`$(C, C1, C2, C3)$ that C is jointly controlled by $C1, C2$ and $C3$; We have adopted from [CEG97] that each product is produced by at most two companies and each company is jointly controlled by at most three other companies. We assume facts `company`$(\cdot)$`.`, `prod_by`$(\cdot,\cdot,\cdot)$`.`, and `contr_by`$(\cdot,\cdot,\cdot,\cdot)$`.` to be defined in a separate program which can be considered as part of $STRATCOMP_{guess}$.

The two programs above intuitively encode guessing a set `strat` of companies which fulfills the production and control preserving constraints, such that no real subset `strat1` fulfills these constraints. While the ad hoc encodings from [EFLP00, LPF$^+$02] are not immediate (and require some thought), the above programs are very natural and easy to come up with.

Given a concrete problem instance by facts defining companies and the production and control relations as mentioned above, integration of these programs after grounding is again possible by our method in an automatic way.

4.3 From $\mathcal{K}$ to Logic Programming

In this section, we will show how planning problems defined so far, namely finding optimistic (optimal) plans and secure (optimal) plans as well as secure checking, can be solved by polynomial reductions to logic programming. To this end, we will apply the methods sketched above, where the theoretical results from Section 3.3 serve as a basic guideline which reductions are feasible.

4.3.1 Optimistic Planning – Translation $lp(\mathcal{P})$

First, we will review the translation of fixed length optimistic planning problems in the basic language $\mathcal{K}$ to logic programs as presented in [EFL$^+$03a] with some refinements. As pinpointed in Section 3.3, this task is possible by a polynomial translation from a $\mathcal{K}$ planning problem $\mathcal{P}$ into a HEDLP (or into a normal LP, respectively), since the problem lies in NPMV.

We assume a $\mathcal{K}$ planning problem $\mathcal{P}$, given by a background knowledge Π and a $\mathcal{K}$ program in the enhanced syntax described in Section 3.1.3, which we translate into a logic program $lp(\mathcal{P})$, whose answer sets represent the optimistic plans of $\mathcal{P}$. For the sake of our translation, we extend fluent and action literals by a timestamp parameter T such that an answer set M of the translated program $lp(\mathcal{P})$ corresponds to a successful trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$ of $\mathcal{P}$. More precisely, we define $lp(\mathcal{P})$ such that the following properties **LP1** – **LP4** are fulfilled:

LP1 The fluent literals in M having timestamp 0 represent a (legal) initial state s_0 of T.

LP2 The fluent literals in M having timestamp $i > 0$ represent a state s_i obtained after executing i many action sets (i.e., they represent the evolution after i steps).

LP3 The action literals in M having timestamp i represent the actions in A_{i+1} (i.e., those actions which are executed at step $i + 1$ of the plan).

LP4 The fluent literals in M having timestamp n represent a state s_n which establishes the goal.

In total, these properties say that trajectories encoded in the answer sets of $lp(\mathcal{P})$ establish the goal of the planning problem, and the underlying sequence of action sets is therefore an optimistic plan. In the following, we incrementally describe a transformation from a planning problem $\mathcal{P}$ to a logic program $lp(\mathcal{P})$ satisfying the properties **LP1** – **LP4**. We will illustrate this transformation on the Bridge Crossing planning problem from Section 3.1.4.

Step 0 (Macro Expansion): In a preliminary step, replace all macros in the $\mathcal{K}$ program by their definitions (cf. Appendix 3.1.3.2).

Example 4.3. In the encoding of Bridge problem, among others the macros

```
nonexecutable takeLamp(X) if hasLamp(X).
inertial at(X,S).
```

are replaced by

```
caused false after takeLamp(X),hasLamp(X).
at(X,S) if not -at(X,S) after at(X,S).
```

◇

Step 1 (Background Knowledge): The background knowledge Π is already given as a logic program; all the rules in Π can be directly included in $lp(\mathcal{P})$ without further modification.

Step 2 (Auxiliary Predicates): To represent time steps, we add the following facts to $lp(\mathcal{P})$

```
time(0).,...,time(i).
next(0,1).,...,next(i − 1,i).
```

where i is the plan length of the query $q = G?(i) \in \mathcal{P}$ at hand. The predicate `time` denotes all possible timestamps and the predicate `next` describes a successor relation over the timestamps in our program.

Note that we refrain from using built-in predicates of a particular logic programming engine here. In the $\texttt{DLV}^{\mathcal{K}}$ implementation, these auxiliary predicates are efficiently handled in a preprocessing step.

Example 4.4. For the original formulation of the Bridge Crossing problem $\mathcal{P}_{BCP}$ (see p. 38), where the goal query is given by

q = `at(joe,across), at(jack,across), at(william,across), at(averell,across)? (5)`

we add the following facts:

```
time(0). time(1). time(2). time(3). time(4). time(5).
next(0,1). next(1,2). next(2,3). next(3,4). next(4,5).
```

◇

Step 3 (Causation Rules): For each causation rule r

`caused` H `if` B `after` A.

in C_R, we include a rule r' in $lp(\mathcal{P})$ as follows:

$$Head(r') = \begin{cases} \emptyset, & \text{if } H = \texttt{false}, \\ f(\vec{t}, T_1), & \text{if } H = f(\vec{t}),\ f \in \sigma^{fl} \end{cases}$$

$Body(r')$ consists of the following literals:

- each (default negated) type literal in r, i.e., (`not`)$l \in A \cup B$ where $l \in \mathcal{L}_{typ}$;
- (`not`) $b(\vec{t}, T_1)$, where (`not`) $b(\vec{t}) \in B$ and $b(\vec{t}) \in \mathcal{L}_{fl}$;
- (`not`) $b(\vec{t}, T_0)$, where (`not`) $b(\vec{t}) \in A$ and $b(\vec{t}) \in \mathcal{L}_{dyn}$.
- for timing, we add
  - `time`(T_1), if A is empty;
  - `next`(T_0,T_1), otherwise.
- To respect typing declarations and to establish safety of r', for any fluent literal in H and any default negated fluent/action literal in $A \cup B$ we add typing information from the corresponding action/fluent declaration. That is, if $H = (\neg)p(\vec{t})$ or `not` $(\neg)p(\vec{t}) \in A \cup B$, $p(\vec{t}) \in \mathcal{L}_{dyn}$, where

  $$p(\vec{Y}) \texttt{ requires } t_1(\vec{Y}_1), \ldots, t_m(\vec{Y}_m)$$

  is an action/fluent declaration (standardized apart), and θ is a witness substitution such that $\vec{Y}\theta = \vec{t}$, then we add $t_1(\vec{Y}_1)\theta, \ldots, t_m(\vec{Y}_m)\theta$ to $Body(r')$. If p has multiple action/fluent declarations, each of them is considered separately, which gives rise to multiple typed versions of r'.

Here, T_0 and T_1 are new variables not occurring in $H \cup A \cup B$.

Example 4.5. In our encoding of the Bridge Crossing problem, the statements

```
caused -hasLamp(X) after takeLamp(Y), hasLamp(X), X != Y.
caused hasLamp(X) if not -hasLamp(X) after hasLamp(X).
```

(where the latter is an expanded `inertial` rule) lead to the following rules in $lp(\mathcal{P})$:

$$\texttt{hasLamp(X,T}_1\texttt{)} \texttt{ :- takeLamp(Y,T}_0\texttt{), hasLamp(X,T}_0\texttt{), person(X), next(T}_0\texttt{,T}_1\texttt{).}$$
$$\texttt{hasLamp(X,T}_1\texttt{)} \texttt{ :- not } \neg\texttt{hasLamp(X,T}_1\texttt{), hasLamp(X,T}_0\texttt{), person(X), next(T}_0\texttt{,T}_1\texttt{).}$$

Here, the timing atom $\texttt{next(T}_0\texttt{,T}_1\texttt{)}$, and the type information `person(X)` for the fluent `hasLamp(X)` in the H-part of each statement are added. Note that we do not have to add `person(Y)` in the translation of the first statement, as `takeLamp(Y)` does not occur default negated and therefore cannot harm safety. ◇

Step 4 (Executability Conditions): Next, we introduce rules and constraints which "guess" an executable action set at each time step. For each executability condition e of the form

$$\texttt{executable } a(\vec{t}) \texttt{ if } B.$$

in C_R, we introduce the following rules in $lp(\mathcal{P})$:

1. A rule e_1' of the form

   $$a(\vec{t},T_0) \texttt{ v } \neg a(\vec{t},T_0) \texttt{:- next}(T_0,T_1), Type.$$

   Here, $Body(e_1')$ is contains type information literals $Type$ for $a(\vec{t})$ obtained from the respective action declaration similar as in Step 3 (and like there we may obtain multiple rules in case of multiple declarations of the same action). These rules "guess" each action at the transition from time step T_0 to T_1.

2. A rule e_2' with

   $Head(e_2') = \texttt{ exec}_a(\vec{t},T_0)$

   $Body(e_2')$ consists of the following literals:

   - each (default negated) type literal in e, i.e., $(\texttt{not})l \in B$ where $l \in \mathcal{L}_{typ}$;
   - $(\texttt{not})\, b(\vec{t},T_0)$, where $(\texttt{not})\, b(\vec{t}) \in B$ and $b(\vec{t}) \in \mathcal{L}_{dyn}$;
   - $\texttt{next}(T_0,T_1)$
   - for typing and safety, type information literals for $a(\vec{t})$ and every default negated literal $\texttt{not } (\neg)p(\vec{t}) \in B$ such that $p(\vec{t}) \in \mathcal{L}_{dyn}$, similar as in Step 3 (which may again lead to multiple rules e_2').

3. Finally, a constraint e_3', which forbids that action $a(\vec{t})$ is guessed by e_1' without being executable:

   $$\texttt{:- } a(\vec{t},T_0), \texttt{ not exec}_a(\vec{t},T_0).$$

where T_0, T_1 are new variables not occurring in $\{a(\vec{t})\} \cup B$.

Simplified translation: Note that this translation differs slightly from the original translation in [EFL+03a]. There, we assumed that executability conditions are never cyclic, i.e., the directed graph built from all executability conditions in C_R such that each action

atom occurring in C_R is a node and there is an edge from a' to a iff there is an executability condition e in $\mathcal{P}$ such that in $a = \mathsf{h}(e)$ and $a' \in \mathsf{pre}^+(e)$, is acyclic. Under this assumption, we can optimize Step 4 by substituting the above rules e'_1–e'_3 with a single rule e' with $Head(e') = Head(e'_1)$ and $Body(e') = Body(e'_2)$.

This is only applicable for acyclic executability conditions, since in general the rule e' is not appropriate. In fact, minimality of answer sets can cause problems when action executability involves cyclic mutual dependencies as shown by the following example (which is a slight variation of our original Bridge Crossing example).

Example 4.6. In our running example, assume an additional action `carryLamp(X)` expressing that person `X` carries the lamp over the bridge. We want to express that crossing is only executable simultaneously with carrying the lamp and vice versa as follows:

```
executable carryLamp(X) if cross(X), hasLamp(X).
executable cross(X) if carryLamp(X).
```

(For simplicity, we ignore the action `crossTogether` and its causation rules). Following the semantics of $\mathcal{K}$, any executable action set A contains either both or none of the actions `cross`(x) and `carryLamp`(x) for any person x. The two executability conditions above introduce in $lp(\mathcal{P})$ the rules

```
carryLamp(X,T0) v -carryLamp(X,T0) :- person(X), next(T0,T1).
exec_carryLamp(X,T0):- person(X), cross(X,T0), hasLamp(X,T0), next(T0,T1).
:- carryLamp(X,T0), not exec_carryLamp(X,T0).

cross(X,T0) v -cross(X,T0) :- person(X) ,next(T0,T1).
exec_cross(X,T0):- person(X), carryLamp(X,T0), next(T0,T1).
:- cross(X,T0), not exec_cross(X,T0).
```

where type information `person(X)`, `location(L)` has been added. Taking only this clipping of the program, we can easily verify that the corresponding answer sets are exactly the guesses where for each person x and time point t either `cross`(x,t) *and* `carryLamp`(x,t) are true or none of both whenever x has the lamp. This perfectly reflects the semantics of executable actions sets.

On the other hand, the simplified translation above would result in the following two rules:

```
carryLamp(X,T0) v -carryLamp(X,T0) :- cross(X,T0), hasLamp(X,T0),
                                      person(X), next(T0,T1).
cross(X,T0) v -cross(X,T0) :- carryLamp(X,T0), person(X), next(T0,T1).
```

where none of the literals `cross`(x,t), `carryLamp`(x,t), –`cross`(x,t), or –`carryLamp`(x,t) is contained in any answer for a person x and time point t, due to minimality of answer sets. In general, the simplified translation is not applicable whenever (positive) cyclic dependencies among executability conditions occur. ◇

Step 5 (Initial State Constraints): Initial state constraints in $I_{R'}$ are transformed like static causation rules r (i.e., A is empty) in Step 3 but we use the constant `0` instead of the variable T_1 and omit the literal `time(0)`.

Example 4.7. The facts in I_R:

```
initially: caused at(X,here).
           caused hasLamp(joe).
```

become:

```
at(X,here,0) :- person(X).
hasLamp(joe,0):- person(joe).
```

◇

Step 6 (Goal Query): Finally, the query q:

goal : $g_1(\vec{t}_1), \ldots, g_m(\vec{t}_m)$, not $g_{m+1}(\vec{t}_{m+1}), \ldots,$ not $g_n(\vec{t}_n)$? (i).

is translated to:

goal_reached :- $g_1(\vec{t}_1, i), \ldots, g_m(\vec{t}_m, i)$, not $g_{m+1}(\vec{t}_{m+1}, i), \ldots,$ not $g_n(\vec{t}_n, i)$.
:- not goal_reached.

where goal_reached is a new predicate symbol. Intuitively, the final constraint invalidates any answer set of $lp(\mathcal{P})$ such that the goal is not reached in the final time step i.

Example 4.8. For our running example,

```
goal:      at(joe,across), at(jack,across), at(william,across),
           at(averell,across)? (5)
```

is translated to:

```
goal_reached :- at(joe,across,5), at(jack,across,5),
                at(william,across,5),at(averell,across,5).
:- not goal_reached.
```

◇

The complete transformation of the basic Bridge Crossing problem, $\mathcal{P}_{BCP}$ from Figure 3.1, after expansion of all macros (see Section 3.1.3.2), is shown in Appendix A.2.

4.3.1.1 Answer Set Correspondence

The following result formally states the desired correspondence between the solutions of a $\mathcal{K}$ planning problem $\mathcal{P}$ and the answer sets of the logic program $lp(\mathcal{P})$ obtained by following the procedure described above.

Theorem 4.7. *Let $\mathcal{P}$ be a planning problem given by a background knowledge Π and a $\mathcal{K}$ program, and let $lp(\mathcal{P})$ be the logic program generated by Steps 0–6 above. Define, for any consistent set of ground literals S, the sets $A_j^S = \{\ a(\vec{t}) \mid a(\vec{t}, j-1) \in S,\ a \in \sigma^{act}\}$ and $s_j^S = \{\ f(\vec{t}) \mid f(\vec{t}, j) \in S,\ f(\vec{t}) \in \mathcal{L}_{fl}\}$, for all $j \geq 0$. Then,*

(i) for each optimistic plan $P = \langle A_1, \ldots, A_i \rangle$ *of* $\mathcal{P}$ *and witnessing trajectory* $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$, *there exists some answer set* S *of* $lp(\mathcal{P})$ *such that* $A_j = A_j^S$ *for all* $j = 1, \ldots, i$ *and* $s_j = s_j^S$, *for all* $j = 0, \ldots, i$;

(ii) for each answer set S *of* $lp(\mathcal{P})$, *the sequence* $P = \langle A_1, \ldots, A_i \rangle$ *is a solution of* $\mathcal{P}$, *i.e. an optimistic plan, witnessed by the trajectory* $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{i-1}, A_i, s_i \rangle \rangle$, *where* $A_j = A_j^S$ *and* $s_k = s_k^S$ *for all* $j = 1, \ldots, i$ *and* $k = 0, \ldots, i$.

We remark that this correspondence reflects exactly the desired properties **LP1** – **LP4** from above.

Proof. The proof is again based on the well-known notion of splitting of a logic program as defined by Lifschitz and Turner [LT94], were we use the theorem on *Splitting Sequences* which extends the Splitting Set Theorem to a monotone sequence of Splitting Sets. We define the Splitting Sequence $U = \langle U_{BG}, U_0, \ldots, U_i, U_G \rangle = \langle BG, BG \cup S_0, \ldots, BG \cup S_0 \cup \ldots \cup S_i, BG \cup S_0 \cup \ldots \cup S_i \cup G \rangle$ of the ground program $P' = lp(\mathcal{P})\downarrow$ as follows:

- BG is the set of type literals and `time` and `next` literals occurring in P';
- S_j, $0 \leq j \leq i$, is the set of literals in P' of the form $f(\vec{t}, j)$, where $f \in \sigma^{fl}$, and of the forms $a(\vec{t}, j-1), \mathtt{exec}_a(\vec{t}, j-1)$, where $a \in \sigma^{act}$;
- $G = \{\mathtt{goal\_reached}\}$.

By the *Splitting Sequence Theorem* of [LT94], P' (and thus $lp(\mathcal{P})$) has some (consistent) answer set S iff $S = X_{BG} \cup X_0 \cup \ldots \cup X_i \cup X_G$ for some solution $X = \langle X_{BG}, X_0, \ldots, X_i, X_G \rangle$ of P' wrt. U. We note the following facts.

- $P_{BG} = b_{U_{BG}}(P')$ (as defined in [LT94], intuitively the program corresponding to U_{BG}) consists of the background program and of the facts defining `time` and `next`.
- $P_0 = e_{U_{BG}}(b_{U_0}(P') \setminus b_{U_{BG}}(P'), X_{BG})$ (as defined in [LT94], intuitively the program corresponding to U_0) consists of rules and constraints which are translations of initial state constraints and static rules (i.e., causation rules with an empty `after`-part), where the argument of `time` and the last argument of the head predicates has been instantiated with 0.
- $P_j = e_{U_{j-1}}(b_{U_j}(P') \setminus b_{U_{j-1}}(P'), X_{BG} \cup X_0 \cup \cdots X_{j-1})$, for $1 \leq j \leq i$ (intuitively, the program corresponding to U_j), consists of rules and constraints which are translations of causation rules and executability conditions in the `always`-section, in which the argument of `time` and the second argument of `next` is instantiated with j (thus, the last argument in head predicates of rules from causation rules and executability conditions is j and $j-1$, respectively).
- $P_G = e_{U_i}(b_{U_G}(P') \setminus b_{U_i}(P'), X_{BG} \cup X_0 \cup \cdots X_i)$ (intuitively the program corresponding to U_G) consists of the rule and the constraint which were generated by Step 6.
- $X_{BG} \cup X_0 \cup \ldots \cup X_i \cup X_G$ is a consistent set iff each of the sets $X_{BG}, X_0, \ldots, X_i, X_G$ is consistent, since there is no literal in any of the sets $BG, S_0, \ldots, S_i, G$ such that its complement is contained in any other of these sets.

We now prove (i) and (ii) of the theorem.

(i) We show that for each optimistic plan a corresponding answer set S of $lp(\mathcal{P})$ as described exists. By the Splitting Sequence Theorem, we must prove that a respective solution $X = \langle X_{BG}, X_0, \ldots, X_i, X_G \rangle$ of $lp(\mathcal{P})$ exists:

X_{BG}: As s_0 is a legal initial state, the background knowledge has a consistent answer set. Thus, by definition of P_{BG}, it clearly has a consistent answer set X_{BG}.

X_0: s_0 in the witnessing trajectory must be a legal initial state, so s_0 satisfies all rules in the `initially`-section and the rules in the `always`-section with empty `after`-part, under the answer set semantics if causal rules are read as logic programming rules. These rules are essentially identical (modulo the `time` literals and the timestamp arguments) to P_0, so X_0 exists and $s_0 = s_0^S$.

X_j: For $1 \leq j \leq i$, $\langle s_{j-1}, A_j, s_j \rangle$ must be a legal transition. We proceed inductively. A_j has to be an executable action set wrt. s_{j-1}, so each action $a \in A_j$ must occur in the head of an executability condition whose body is true wrt. s_{j-1}. There must be a corresponding rule in P_j constructed by Step 4 of the translation with head $\texttt{exec}_a(j-1)$ such that its body is true wrt. X_{j-1}. If we choose X_j such that $A_j^S = A_j$, then all the guessing rules from Step 4 are satisfied and none of the constraints from Step 4 is violated: For each rule in P_j which has an action literal l_a in the head such that l_a is not in A_j, we include its negation $\neg l_a$ in X_j in order to satisfy the remaining guessing rules.

Furthermore, s_j satisfies all causal rules from the `always`-section whose `after`-part is true wrt. s_{j-1} and A_j under the answer set semantics. From the correspondence of causal rules from the `always`-section and rules in P_j, we may thus conclude that P_j has an answer set X_j s.t. $s_j = s_j^S$ and $A_j = A_j^S$, as seen above.

X_G: s_i satisfies the goal of $\mathcal{P}$. Let $g_1, \ldots, g_m, \texttt{not}\ g_{m+1}, \ldots, \texttt{not}\ g_n?(i)$ be the goal of $\mathcal{P}$. Then $\{g_1, \ldots, g_m\} \subseteq s_i$ and $\{g_{m+1}, \ldots, g_n\} \cap s_i = \emptyset$ hold. Since $s_i = s_i^{X_i}$, the body of the rule generated in Step 6 is true and therefore $X_G = \{\texttt{goal\_reached}\}$ exists.

In total, we have shown that for each optimistic plan of $\mathcal{P}$ a corresponding answer set S of $lp(\mathcal{P})$ exists, which contains literals representing a witnessing trajectory.

(ii) We must prove that for each answer set S of $lp(\mathcal{P})$, a corresponding optimistic plan of $\mathcal{P}$ exists. By the Splitting Sequence Theorem, a solution $X = \langle X_{BG}, X_0, \ldots, X_i, X_G \rangle$ exists for $lp(\mathcal{P})$. Since X_0 is an answer set for the program corresponding to initial state constraints and static rules, a legal initial state $s_0 = s_0^S$ must exist as well.

For X_j, $1 \leq j \leq i$, we proceed inductively. All rules corresponding to executability conditions wrt. time $j-1$ must be satisfied, so for every literal l_a in X_j which corresponds to a positive action literal, a rule in P_j whose body is true wrt. X_{j-1} and which has a corresponding $\texttt{exec}_a$ in the head must exist in order to not violate the constraint from Step 4. By construction, an executability condition whose body is true wrt. s_{j-1} and whose head is the corresponding action literal, must exist in $\mathcal{P}$, so an executable action set $A_j = A_j^S$ exists wrt. s_{j-1}.

All rules in P_j corresponding to causal rules from the `always`-section of $\mathcal{P}$ must be satisfied by X_j and X_{j-1} (for literals translated from `after`-parts). So for each causal rule

in $\mathcal{P}$, either its `after`-part is false wrt. A_j and s_{j-1}, or the causal rule is satisfied by the state $s_j = s_j^S$.

Finally, since X_G exists, `goal_reached` must be true. Hence, the body of the rule generated in Step 6 must be true, and therefore s_i must establish the goal of $\mathcal{P}$.

In total, we have shown that for each answer set S of $lp(\mathcal{P})$ an optimistic plan of $\mathcal{P}$ exists, such that the witnessing trajectory can be constructed from S as described. □

4.3.1.2 Special Translation for Macros

Some macros allow for more concise translations before expansion in Step 0 of $lp(\mathcal{P})$. In particular this applies to the `total` macro and to `noConcurrency`.

Macro `total`: Instead of expanding a statement

`total` $f(\vec{t})$ `if` B `after` A.

(where $f \in \sigma^{fl}$) like in Section 3.1.3.2 and translating the resulting two causation rules separately, this statement can equivalently be translated to a single disjunctive rule r' with

$Head(r') = \{\ f(\vec{t}, T_1), -f(\vec{t}, T_1)\ \}$

where T_1 is a new variable. As for body $Body(r')$ and the safety of $Head(r')$ we proceed analogously to the translation of causation rules in Step 3 above. Note that in this special case, disjunction is semantically equivalent to the regular translation by the two rules from page 37, which involves non-stratified negation. This is the case because the implicit constraint $\text{:-}f(\vec{t}, T_1), -f(\vec{t}, T_1)$. (cf. Footnote 5 in Section 2.1) imposes a restriction similar to head-cycle-freeness, since any possible head-cycle would cause inconsistency: Head-cycle-freeness ensures that literals in the same rule head do not mutually depend on each other in any answer set, which is also enforced here, since a literal can not positively depend on its complement (in any consistent answer set).

As easily seen, the original macro translation amounts exactly to the rewriting for head-cycle-free disjunction pinpointed in Section 2.1.4.1 on page 10, just in the opposite direction.

Macro `noConcurrency`: Exploiting the recent extension of `DLV` by aggregates [DFI+03], we can find an alternative translation for the `noConcurrency` macro which, as we will see, allows for more flexibility than the simple expansion of the macro in Section 3.1.3.2. Here, we use the built-in `#count` aggregate in order to count the number of actions per time step `T`: For any action name $a \in \sigma^{act}$ with arity n, we add a rule

$\texttt{occurred}(a, \texttt{X}_1, \ldots, \texttt{X}_n, 0, \ldots, 0, \texttt{T})$:- $a(\texttt{X}_1, \ldots, \texttt{X}_n, \texttt{T})$.

where `occurred` is a new distinct predicate with arity $l + 2$, where l is the maximum arity of all actions in PD. We add $l - n$ parameters with constant "0" between $\texttt{X}_n$ and `T` if $n < l$. This is necessary in order to get uniform arity $l + 2$ for predicate `occurred`.

Finally, we add a single constraint involving the `DLV` `#count` aggregate, for limiting the actions per time:

:- $\texttt{next}(\texttt{T}_0, \texttt{T}_1)$, 0 <= #count{ $\texttt{occurred}(\texttt{A}, \texttt{X}_1, \ldots, \texttt{X}_l, \texttt{T}_0)$ } <= 1.

Note that this translation allows, with slight modification, for arbitrary upper and lower bounds x and y, respectively, on the number of actions per time instead of 0 and 1, beyond `noConcurrency`. In fact, this translation is used for the `-planminactions=`x and `-planmaxactions=`y command-line options of the DLV$^{\mathcal{K}}$ system presented in Chapter 5.

4.3.1.3 Alternative Translation for SMODELS

As the reader can easily verify, the above transformation $lp(\mathcal{P})$ employs disjunction only in Step 4 for translating executability conditions (we consider only the original translation for macro `total`). Furthermore, negated action atoms $\neg a(\vec{t}, T)$ occur only in the heads of the rules of $lp(\mathcal{P})$. Thus, the program is head-cycle-free, which is profitably exploited by the DLV engine underlying our implementation. The disjunction, which informally encodes a guess of whether the action $a(\vec{t})$ is executed or not at time T_0, may equivalently be replaced by disjunction-free guessing rules as shown in Section 2.1.4.1 (see p. 10). The adapted transformation can then also be used on engines for computing answer sets of normal programs, such as SMODELS.

4.3.2 Optimistic Optimal Planning – Translation $lp^w(\mathcal{P})$

From [BLR00] we know that, given a head-cycle-free (extended) disjunctive program with weak constraints (HEDLP^w) as defined in Section 2.1.5.1, deciding whether a query q is true in some optimal answer set is Δ_2^P-complete. The respective class for computing such an answer set is FΔ_2^P-complete. Together with the results from Section 3.3 this indicates that translations of optimal planning problems to head-cycle-free disjunctive logic programs with weak constraints or the language of SMODELS are feasible in polynomial time.

To this end, we extend the basic translation $lp(\cdot)$ for a $\mathcal{K}^c$ problem $\mathcal{P}$ to a HEDLP^w $lp^w(\mathcal{P})$ which computes optimistic optimal plans: $lp^w(\mathcal{P})$ includes all rules of $lp(\mathcal{P}_{nc})$, where $\mathcal{P}_{nc}$ is obtained from $\mathcal{P}$ by stripping off all cost parts. Furthermore, we add the following step:

Step 7 (Action Costs): For any action declaration d of form (3.5) with a non-empty `costs`-part, add:
(i) A new rule r_d of the form

$$\texttt{cost}_\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}, \texttt{C}\theta) \texttt{ :- } \begin{array}{l} \texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}),\ \texttt{t}_1, \ldots, \texttt{t}_\texttt{m}, \\ \texttt{c}_1\theta, \ldots, \texttt{c}_\texttt{k}\theta, \texttt{U} = \texttt{T} + 1. \end{array} \tag{4.4}$$

where $\texttt{cost}_\texttt{p}$ is a new symbol, `T` and `U` are new variables and $\theta = \{\texttt{time} \to \texttt{U}\}$. As an optimization, $\texttt{U} = \texttt{T} + 1$ is only present if `U` occurs elsewhere in r_d.[23]
(ii) A weak constraint wc_d of the form

$$:\sim\ \texttt{cost}_\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}, \texttt{C}\theta).\ [\texttt{C}\theta :] \tag{4.5}$$

Example 4.9. For example, the `cross` action from the Quick Bridge Crossing problem $\mathcal{P}_{QBCP}$ (see p. 44) is translated to

[23] In the current implementation we simply add `next(T, T1)` and replace all occurrences of "`time`" by `T1`.

$\texttt{cost}_{\texttt{cross}}(\texttt{X},\texttt{T},\texttt{WX})\texttt{:-}\ \texttt{cross}(\texttt{X},\texttt{T}),\ \texttt{person}(\texttt{X}),\ \texttt{walk}(\texttt{X},\texttt{WX}).$

$\texttt{:}\sim\ \texttt{cost}_{\texttt{cross}}(\texttt{X},\texttt{T},\texttt{WX}).\ [\texttt{WX}:]$

◇

For a complete translation of the Quick Bridge Crossing example, we refer to Appendix A.2.

4.3.2.1 Answer Set Correspondence

As we have shown, the answer sets of $lp(\mathcal{P})$ correspond to trajectories of optimistic plans for $\mathcal{P}$. The following theorem states a similar correspondence result for $lp^w(\mathcal{P})$ and optimal plans for $\mathcal{P}$.

A_j^S and s_j^S are defined as above. It is easy to see that Theorem 4.7 applies wrt. candidate answer sets of $lp^w(\mathcal{P})$ as well, and we can slightly extend it by costs:

Theorem 4.8 (Answer Set Correspondence). *Let $\mathcal{P} = \langle PD, q\rangle$ be a (well-defined) $\mathcal{K}^c$ planning problem, and let $lp^w(\mathcal{P})$ be the above program. Then,*

(i) for each optimistic plan $P = \langle A_1, \ldots, A_l\rangle$ of $\mathcal{P}$ and supporting trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{l-1}, A_l, s_l\rangle\rangle$ of P, there exists some answer set S of $lp^w(\mathcal{P})$ such that $A_j = A_j^S$ for all $j = 1, \ldots, l$, $s_j = s_j^S$, for all $j = 0, \ldots, l$ and $cost_{\mathcal{P}}(P) = \mathsf{cost}_{lp^w(\mathcal{P})}(S)$;

(ii) for each answer set S of $lp^w(\mathcal{P})$, the sequence $P = \langle A_1, \ldots, A_l\rangle$ is a solution of $\mathcal{P}$, i.e., an optimistic plan, witnessed by the trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{l-1}, A_l, s_l\rangle\rangle$ with $cost_{\mathcal{P}}(P) = \mathsf{cost}_{lp^w(\mathcal{P})}(S)$, where $A_j = A_j^S$ and $s_k = s_k^S$ for all $j = 1, \ldots, l$ and $k = 0, \ldots, l$.

The proof is based on the respective correspondence result for $\mathcal{K}$ in Theorem 4.7:

Proof. Disregarding weak constraints, we can split the program $lp^w(\mathcal{P})$ into a bottom part consisting of $lp(\mathcal{P}_{nc})$, where $\mathcal{P}_{nc}$ is $\mathcal{P}$ with the cost information stripped off, and a top part containing the remaining rules; we then derive the correspondence between optimistic plans for $\mathcal{P}$ and answer sets of $lp^w(\mathcal{P})$ from the similar correspondence result for $lp(\mathcal{P}_{nc})$.

By Theorem 4.7, any answer set S' of $lp(\mathcal{P}_{nc})$ corresponds to some trajectory T' of an optimistic plan P' for $\mathcal{P}_{nc}$ and vice versa.

In what follows, when talking about $lp(\mathcal{P}_{nc})$ and $lp^w(\mathcal{P})$, we mean the respective grounded logic programs. $lp^w(\mathcal{P})$ augments $lp(\mathcal{P}_{nc})$ by rules (4.4) and weak constraints (4.5). Let now $U = lit(lp(\mathcal{P}_{nc}))$ be the set of all literals occurring in $lp(\mathcal{P}_{nc})$. Clearly, U splits $lp^w(\mathcal{P})$ as defined in [LT94], where we disregard weak constraints in $lp^w(\mathcal{P})$, since the rules of form (4.4) introduce only new head literals. Consequently, we get $b_U(lp^w(\mathcal{P})) = lp(\mathcal{P}_{nc})$. Then, for any answer set S' of $lp(\mathcal{P}_{nc})$, each rule in $e_U(lp^w(\mathcal{P}) \setminus b_U(lp^w(\mathcal{P})), S')$ is of the form

$$cost_a(x_1, \ldots, x_n, t, c)\ \texttt{:-}\ Body.$$

From the fact that all these rules are positive, we can conclude that with respect to the split by U, any answer set S' of $lp(\mathcal{P}_{nc})$ induces a unique answer set $S \supseteq S'$ of $lp^w(\mathcal{P})$. Therefore, modulo costs, a correspondence between supporting trajectories T and candidate answer sets S as claimed follows directly from Theorem Theorem 4.7.

It remains to prove that $cost_{\mathcal{P}}(P) = \mathsf{cost}_{lp^w(\mathcal{P})}(S)$ holds for all candidate answer sets S corresponding to an optimistic plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$. By the correspondence shown above, any action $p(x_1, \ldots, x_n) \in A_j$ corresponds to exactly one ground atom $p(x_1, \ldots, x_n, j-1)$, $j \in \{1, \ldots, l\}$. Therefore, if $p(x_1, \ldots, x_n)$ is declared with a non-empty `costs`-part, by (4.4) and well-definedness, modulo $x_1, \ldots, x_n$, there is exactly one fact $cost_p(x_1, \ldots, x_n, j-1, c)$ in the model of $e_U(lp^w(\mathcal{P}) \setminus b_U(lp^w(\mathcal{P})), S)$.

Furthermore, by definition of (4.4), we have that $c = cost_j(p(x_1, \ldots, x_n))$, i.e., the cost of action instance $p(x_1, \ldots, x_n)$ at time j. Consequently, by the weak constraints of form (4.5) for the corresponding in $lp^w(\mathcal{P})$ for each action instance $p(x_1, \ldots, x_n)$ at time j, c is added to $\mathsf{cost}_{lp^w(\mathcal{P})}(S)$

Since all violation values adding to $\mathsf{cost}_{lp^w(\mathcal{P})}(S)$ stem from weak constraints (4.5), in total we have $\mathsf{cost}_{lp^w(\mathcal{P})}(S) = \sum_{j=1}^{l} \sum_{a \in A_j} cost_j(a) = cost_{\mathcal{P}}(P)$. This proves the result. □

From this result and the definitions of optimal cost plans and optimal answer sets, we conclude the following result:

Corollary 4.9 (Optimal answer set correspondence). *For any well-defined $\mathcal{K}^c$ planning problem $\mathcal{P} = \langle PD, Q\,?\,(l) \rangle$, the trajectories $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{l-1}, A_l, s_l \rangle \rangle$ of optimal plans P for $\mathcal{P}$ correspond to the optimal answer sets S of $lp^w(\mathcal{P})$, such that $A_j = A_j^S$ for all $j = 1, \ldots, l$ and $s_j = s_j^S$, for all $j = 0, \ldots, l$.*

Proof. For each $a \in A_j$, the weak constraint (4.5) causes a violation value of $cost_j(a)$. Furthermore, these are the only cost violations. Thus, a candidate answer set S is optimal if and only if $\mathsf{cost}_{lp^w(\mathcal{P})}(S) = cost_{\mathcal{P}}(P)$ is minimal, i.e., S corresponds to an optimal plan. □

A similar correspondence result also holds for admissible plans:

Corollary 4.10 (Answer set correspondence for admissible plans). *For any well-defined $\mathcal{K}^c$ planning problem $\mathcal{P} = \langle PD, Q\,?\,(l) \rangle$, the trajectories $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{l-1}, A_l, s_l \rangle \rangle$ of admissible plans P for $\mathcal{P}$ wrt. cost c correspond to the answer sets S of $lp^w(\mathcal{P})$ having $\mathsf{cost}_{lp^w(\mathcal{P})}(S) \leq c$, such that $A_j = A_j^S$ for all $j = 1, \ldots, l$ and $s_j = s_j^S$, for all $j = 0, \ldots, l$.*

4.3.2.2 Alternative Translation for SMODELS

Apart from the presented translation using weak constraints, one could also choose an alternative approach for the translation to answer set programming. SMODELS [SNS02] supports another extension to pure answer set programming allowing to minimize over sets of predicates (cf. Section 2.1.5.2) which (together with the rewriting of disjunctive rules for SMODELS outlined above) allows for optimal plan computation as well. To this end, we use SMODELS `minimize` statement in an alternative formulation of Step 7:

Step 7a: For action declarations with non-empty `costs`-parts, we add a new rule of form

$$\texttt{cost(p,X}_1\texttt{,}\ldots\texttt{,X}_\texttt{n}\texttt{,0,}\ldots\texttt{,0,T,C}\theta\texttt{) :- t}_1\texttt{,}\ldots\texttt{,t}_\texttt{m}\texttt{, c}_1\theta\texttt{,}\ldots\texttt{,c}_\texttt{k}\theta\texttt{,U = T + 1.} \qquad (4.6)$$

similar to Step 7 above, with two differences: (1) action name p is now a parameter, and (2) we add $l - \mathtt{n}$ parameters with constant "0" between $\mathtt{X_n}$ and T where l is the maximum arity of all actions in *PD*. This is necessary in order to get unique arity $l + 2$ for predicate cost. Furthermore, we add

$$\mathtt{occurs(p, X_1, \ldots, X_n, 0, \ldots, 0, T)} \text{ :- } \mathtt{p(X_1, \ldots, X_n, T), t_1, \ldots, t_m,}. \tag{4.7}$$

This second rule adds the same "0" parameters as for to achieve unique arity $l+1$ of the new predicate occurs. Using SMODELS syntax, we can now compute optimal plans by adding

$$\mathtt{minimize[occurs(A, X_1, ..., X_l, T) : cost(A, X_1, ..., X_l, T, C) = C]}.$$

4.3.3 Secure Checking in General – Translation $lp^{SC}(\mathcal{P}, P)$

Before discussing how to encode secure planning in logic programming, we first have to consider the subproblem of checking plan security. Since secure checking in general is already Π_2^P-complete (cf. Theorem 3.5) and secure planning even for fixed plan length is on the third level of the PH (cf. Theorem 3.6) we can only encode the former as a DLP, but not the latter. In this section, we will proceed as follows: After providing a general method for secure checking by means of logic programming based on the ideas from Section 4.2, we will identify some syntactic subclasses of proper $\mathcal{K}$ planning problems where easier co-NP checks can be applied. We will then show how to integrate these easier checks by the method from Section 4.2.

Given a $\mathcal{K}$ planning problem $\mathcal{P} = \langle PD, q \rangle$ and a sequence of action sets $P = \langle A_1, \ldots, A_n \rangle$, we now want to decide whether P is a secure plan by Answer Set Programming methods. The basic idea for solving the Π_2^P-complete problem of checking plan security is as follows: We will construct a program $lp^{SC}(\mathcal{P}, P)$ solving the complementary problem, deciding whether P is *insecure*, such that any answer set of $lp^{SC}(\mathcal{P}, P)$ corresponds to a witness for plan insecurity. On the other hand, whenever $lp^{SC}(\mathcal{P}, P)$ has no answer set at all, then P shall be secure.

The question whether P is insecure consists of two parts: Namely, P is insecure if there exists a (partial) trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$ such that either

(a) $j = n$ and the goal is not satisfied in s_j, or

(b) $j < n$ and no legal transition $\langle s_j, A_{j+1}, s_{j+1} \rangle$ exists.

So, we can split the problem into "guess" and "check" parts as follows:

Guess: guess a (partial) trajectory for plan P.

Check: check conditions (a) and (b) from above.

In the following, we will assume that $\mathcal{P}$ is propositional, the background knowledge Π is empty, and that the requires parts of all declarations are empty as well. This is no restriction since such a problem can be easily constructed from an arbitrary problem $\mathcal{P}$ by the following transformation:

Recall that $\mathcal{L}_{PD}$ is the set of all legal action instances and literals of legal fluent instances in *PD* (cf. Definition 3.9, p. 33). Let M be the unique answer set of the background

knowledge Π. Then, we denote by $\mathcal{P}$' the ground planning problem obtained from $\mathcal{P}\downarrow$, where we substitute $PD\downarrow = \langle \Pi\downarrow, \langle D, R\downarrow\rangle\rangle$ by $PD' = \langle \emptyset, \langle D', R'\rangle\rangle$ such that D' amounts to a ground action/fluent declaration with empty `requires`-part for each $l \in \mathcal{L}^+_{PD}$ and we obtain R' from $R\downarrow$ by dropping all $r \in R\downarrow$ such that

(i) $\mathsf{h}(r) \not\subseteq \mathcal{L}_{PD}$, or

(ii) $(\mathsf{pre}^+(r) \cup \mathsf{post}^+(r)) \cap \mathcal{L}_{typ} \not\subseteq M$, or

(iii) $(\mathsf{pre}^-(r) \cup \mathsf{post}^-(r)) \cap M \neq \emptyset$

and removing all $l \in \mathcal{L}_{typ}$ from the remaining rules. Intuitively, this means that we remove all "unnecessary" rules from $R\downarrow$ which do not comply with the legal action/fluent instances and the background knowledge. Note that $\mathcal{L}_{PD} = \mathcal{L}_{PD'}$ and each causation rule or executability condition $r' \in PD'$ is satisfied iff the corresponding $r \in PD\downarrow$ is satisfied. On the other hand, each $r \in PD\downarrow$ which does not correspond to any rule in PD' is unsatisfied by definition. We get:

Proposition 4.11. *The optimistic (or secure, resp.) plans of $\mathcal{P}$ and $\mathcal{P}$' coincide.*

4.3.3.1 $lp^{SC}(\mathcal{P}, P)$ – Guess Part

As for the guess part of our integrated secure check program $lp^{SC}(\mathcal{P}, P)$, we need a HEDLP which guesses all legal trajectories T wrt. plan P. Translation $lp(\mathcal{P})$ serves as a basis for this program $lp^T(\mathcal{P}, P)$. Since we may assume that macros have already been expanded and Π is empty, we can skip Step 0 and Step 1. Steps 2 and 3 are included as is. In Step 4 (see p. 88), we have to fix the plan $P = \langle A_1, \ldots, A_n\rangle$ by the following modification:

Step 4a (Executability Conditions): We now only consider those executability conditions which involve actions in P. So, for any $a \in A_j$, $1 \leq j \leq n$, and executability condition e of the form `executable` a `if B.` in C_R, we add the following rules:

1. A fact e'_1 of the form
$$a(j-1).$$
Facts of this form represent all actions in P.

2. A rule e'_2 with
$Head(e'_2) =$ $\mathtt{exec}_a(j-1)$
$Body(e'_2)$ consists of the following literals:
(`not`) $b(j-1)$, where (`not`) $b \in$ `B` and $b \in \mathcal{L}_{dyn}$;
Type literals and safety information can be ignored due to our assumption that $\mathcal{P}$ is propositional and Π is empty.

3. Finally, a constraint e'_3 which forbids that action $a \in A_j$ is not executable:
$$\texttt{:-}\ a(j-1),\ \texttt{not}\ \mathtt{exec}_a(j-1).$$

Step 5 again remains as in $lp(\mathcal{P})$. Finally, we add Step 6, but remove the constraint `:- not goal_reached.` there.

We denote this modified program by $lp^T(\mathcal{P}, P)$. Since the modification in Step 4a enforces the actions in P and ensures that each $A_i \in P$ is an executable action set, by similar considerations as used in the proof of Theorem 4.7, we obtain the following result:

Lemma 4.12. *For any plan $P = \langle A_1, A_2, \ldots, A_n\rangle$ the answer sets of $lp^T(\mathcal{P}, P)$ are in one-to-one correspondence with all legal trajectories $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{n-1}, A_n, s_n\rangle\rangle$ of length n. Furthermore, there exists an answer set S of $lp^T(\mathcal{P}, P)$ such that* `goal_reached` $\in S$ *if and only if P is an optimistic plan.*

Informally, $lp^T(\mathcal{P}, P)$ guesses all legal trajectories T of length n corresponding to plan $P = \langle A_1, \ldots, A_n\rangle$. We further define $lp^T_j(\mathcal{P}, P)$ for $j \leq n$ as the program obtained in the same way as $lp^T(\mathcal{P}, P)$ from the planning problem $\mathcal{P}(j) = \langle PD, \emptyset?(j)\rangle$ where we only consider the prefix $\langle A_1, \ldots, A_j\rangle$ of P. I.e., $lp^T_j(\mathcal{P}, P)$ is the program computing all partial legal trajectories of length j in PD which match a prefix of P with length j.

4.3.3.2 $lp^{SC}(\mathcal{P}, P)$ – Check Part

Intuitively, in order to check insecurity of $P = \langle A_1, \ldots, A_n\rangle$ we could proceed as follows: Iteratively, compute $lp^T_j(\mathcal{P}, P)$ for $j = 0, \ldots, n$ and check for each answer set S of $lp^T_j(\mathcal{P}, P)$, i.e. each (partial) trajectory, whether

(a) $j = n$ and the goal is not satisfied in s^S_j (as defined in Theorem 4.7, p. 90).

(b) $j < n$ and no legal transition $\langle s^S_j, A_{j+1}, s_{j+1}\rangle$ exists.

While (a) can be checked easily, (b) is a co-NP-complete check in general which can be integrated in a single program "guess and check" program by our method from Section 4.2.3 for any fixed reachable state s_j. However, we have to check (b) *for each* reachable state s_j at $j < n$, so the method from Section 4.2.3 can not be applied straightforward but needs some refinements. To this end, we will adapt the approach from Section 4.2 in order to reflect the stepwise computation for states reachable in $1, 2, \ldots, n$ steps.

As a basis, we take the Steps 2 to 5 from $lp^T(\mathcal{P}, P)$, i.e. the auxiliary facts from Step 2, and rules of the following form from Steps 3 (causation rules), 4 (executability conditions), and 5 (initial state constraints):

$$\begin{aligned} r : f(t_1) \texttt{ :- } & g_1(t_1), \ldots, g_k(t_1), \texttt{not } g_{k+1}(t_1), \texttt{not } g_l(t_1), \\ & h_1(t_0), \ldots, h_m(t_0), \texttt{not } h_{m+1}(t_0), \texttt{not } h_n(t_0), Aux. \end{aligned} \tag{4.8}$$

Particularly, t_0, t_1 and Aux have the following assignments in the rules from Steps 3 to 5:

Step 3 For rule r', we have $t_0 = \mathtt{T_0}, t_1 = \mathtt{T_1}$, and $Aux = \mathtt{next(T_0, T_1)}$ (resp. $\mathtt{time(T_1)}$) for dynamic (resp. static) rules.

Step 4a For each $a \in A_j$, for rules e'_1 and e'_2 we have: $t_1 = j - 1$ and $Aux = \emptyset$.
Remark: We ignore the constraint e'_3 for the moment.

Step 5 For rule r', we have $t_1 = \mathtt{0}$ and $Aux = \emptyset$.

We want to generate an integrated program by means of the meta-interpreter from Section 4.2 which checks for any state that is reachable by executing a prefix of plan P whether some legal transition to a successor state exists wrt. the actions in P. Informally, we need one meta-interpreter for every stage $1, \ldots, n$ of the plan.

For each rule of the form (4.8) coming from Step 3, we define

$$\mathtt{body}_{pre}(r, t_1) = \mathtt{inS}(h_1, t_0) \ldots, \mathtt{inS}(h_m, t_0), \mathtt{ninS}(h_{m+1}, t_0), \ldots, \mathtt{ninS}(h_n, t_0), Aux$$

Furthermore, we define $\mathtt{body}_{pre}(r, t_1) = \emptyset$ for the rules from Steps 4a and 5.

Now we are ready to construct the integrated program for secure checking $lp^{SC}(\mathcal{P}, P)$ which consists of the following rules:

1. For every rule r of form (4.8), we add

$$\begin{array}{rll}
\mathtt{lit(h,}f, r, t_1\mathtt{)}\text{:-} & \mathtt{body}_{pre}(r, t_1). & \text{for } l \in Head(r); \\
\mathtt{atom(}l, |l|, t_1\mathtt{)}\text{:-} & \mathtt{body}_{pre}(r, t_1). & \\
\mathtt{lit(p,}g_i, r, t_1\mathtt{)}\text{:-} & \mathtt{body}_{pre}(r, t_1). & \text{for each } i \in \{1, \ldots, k\}; \\
\mathtt{lit(n,}g_j, r, t_1\mathtt{)}\text{:-} & \mathtt{body}_{pre}(r, t_1). & \text{for each } j \in \{k+1, \ldots, l\}.
\end{array}$$

   These rules correspond to the conditional version of $F(\Pi)$ from Section 4.2 where we enrich the input predicates by a timestamp t_1 as defined in rule (4.8). We denote the rules added in this step as $F^{SC}(\mathcal{P}, P)$.

2. Further, we add timestamped versions of Π_{meta}, called $\Pi_{meta}(j)$, for each $j \in \{0, \ldots, n\}$, where $\Pi_{meta}(j)$ is obtained from Π_{meta} by adding parameter j to every literal in Π_{meta}.

3. Note that in $F^{SC}(\mathcal{P}, P)$ we have not considered the constraints e_3' from Step 4a. Instead, for any $a \in A_j$ we add an extra rule:

$$\mathtt{notok(j)} \text{ :- } \mathtt{inS(a, j-1)}, \ \mathtt{ninS(exec_a, j-1)} \tag{4.9}$$

   Intuitively, this amounts to the meta-interpreter representation of constraint e_3' similar to rule in (4.1) in Section 4.2.2 where `notok(j)` will be derived for in the state *after* a has been executed in plan P.

4. Finally, we add the following rule:

$$\mathtt{notok(T_1)} \text{ :- } \mathtt{notok(T_0)}, \ \mathtt{next(T_0, T_1)}. \tag{4.10}$$

   This rule propagates the timestamped version of `notok` to all subsequent time steps, leading to saturation in all $\Pi_{meta}(j)$ for $i < j \leq n$ as soon as some `notok(`i`)` has been derived.

In analogy to $lp_j^T(\mathcal{P}, P)$ we define $lp_j^{SC}(\mathcal{P}, P)$ as the program obtained by restricting $lp^{SC}(\mathcal{P}, P)$ to timestamps up to j, that is,

$$lp_j^{SC}(\mathcal{P}, P) = \Pi_{meta}(0) \cup \ldots \cup \Pi_{meta}(j) \cup F_j^{SC}(\mathcal{P}, P)$$

where $F_j^{SC}(\mathcal{P}, P)$ is the ground instantiation of $F^{SC}(\mathcal{P}, P)$ obtained from instantiating t_1 with 0 up to j for all rules in $F^{SC}(\mathcal{P}, P)$.

Similar to s_i^S in Theorem 4.7, for any answer set S of $lp_j^{SC}(\mathcal{P}, P)$ we define

$$s_{SC,i}^S = \{l \mid l \in \mathcal{L}_{fl} \wedge \texttt{inS(l,i)} \in S\}$$

for $0 \leq i \leq j$ and

$$A_{SC,i}^S = \{l \mid l \in \mathcal{L}_{act} \wedge \texttt{inS(l,i-1)} \in S\}$$

for $0 < i \leq j$.

Now let us state some properties of $lp_j^{SC}(\mathcal{P}, P)$:

Proposition 4.13. *Given a planning problem $\mathcal{P}$ and sequence of action sets $P = \langle A_1, \ldots, A_n \rangle$*

(i) If $lp_j^{SC}(\mathcal{P}, P)$ has an answer set S, such that $\texttt{notok}(j) \notin S$ then there exists a legal trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ of length j for $\mathcal{P}$ corresponding to the prefix $P = \langle A_1, \ldots, A_j\rangle$ of P.

(ii) Each legal trajectory T corresponding to the prefix $P = \langle A_1, \ldots, A_j\rangle$ of P is encoded by some answer set S of $lp_j^{SC}(\mathcal{P}, P)$, where $s_i = s_{SC,i}^S$ for $0 \leq i \leq j$ and $A_i = A_{SC,i}^S$ is an executable action set wrt. s_{i-1}.

(iii) Each answer set S of $lp_j^{SC}(\mathcal{P}, P)$ which contains $\texttt{notok}(j)$, where $i < j$ is the smallest timestamp such that $\texttt{notok}(i+1) \in S$, corresponds to a partial trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{i-1}, A_i, s_i\rangle\rangle$ where $s_{SC,i}^S$ marks a state s_i such that no legal transition $\langle s_i, A_{i+1}, s_{i+1}\rangle$ exists.

Proof. (Sketch) We proceed inductively:

- $lp_0^{SC}(\mathcal{P}, P)$: Since $lp_0^{SC}(\mathcal{P}, P)$ in principle amounts to $tr(lp_0^T(\mathcal{P}, P))$ from Theorem 4.2, by this Theorem and Lemma 4.12 each answer set S of $lp_0^{SC}(\mathcal{P}, P)$, such that $\texttt{notok(0)} \notin S$, represents a legal initial state of $\mathcal{P}$ by $s_{SC,0}^S$ (i.e. a legal trajectory of length 0). This proves (i), and (ii), respectively. We remark that the existence of an answer set $\Omega(0)$ of $lp_0^{SC}(\mathcal{P}, P)$ containing $\texttt{notok(0)}$ indicates that $\mathcal{P}$ has no legal initial state.[24] Therefore, (iii) is trivially satisfied for $j = 0$, since non-existence of a legal initial state also implies that no legal transition $\langle s_0, A_1, s_1\rangle$ exists.

- $lp_{j+1}^{SC}(\mathcal{P}, P)$, for $j \geq 0$:

  (i) Assume that S is an answer set of $lp_{j+1}^{SC}(\mathcal{P}, P)$ such that $\texttt{notok(}j+1\texttt{)} \notin S$. By this assumption and rule (4.10) we can further conclude that $\texttt{notok(}j\texttt{)} \notin S$. We again use the Splitting Set Theorem: Obviously, $Lit(lp_j^{SC}(\mathcal{P}, P))$ splits $lp_{j+1}^{SC}(\mathcal{P}, P)$, so there exists a subset $S' \subseteq S$, such that S' is an answer set of $lp_j^{SC}(\mathcal{P}, P)$. By the induction hypothesis, S' represents a partial trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ where $s_j = s_{SC,j}^{S'}$. It remains to show that $\langle s_j, A_{j+1}, s_{j+1}\rangle$ is a legal transition. First, by rule (4.9) above and our assumption that $\texttt{notok(}j+1\texttt{)} \notin S$, we conclude that $A_{j+1} = A_{SC,j+1}$ must be an executable action set. Note that $s_{SC,j}^{S'}$ denotes exactly the literals from S' which are relevant for the evaluation of the top part of $lp_{j+1}^{SC}(\mathcal{P}, P)$,

[24] $\Omega(0)$ is defined analogously to Ω in Section 4.2 taking the additional parameters 0 in $\Pi_{meta}(0)$ into account.

since only those literals from S affect the bodies of rules in $F^{SC}_{j+1}(\mathcal{P},P) \setminus F^{SC}_{j}(\mathcal{P},P)$: Intuitively, $F^{SC}_{j+1}(\mathcal{P},P) \setminus F^{SC}_{j}(\mathcal{P},P)$ encodes the "input representation" of the rules for $\Pi_{meta}(j+1)$ corresponding to causation rules and executability conditions r with a satisfied $\mathsf{pre}(r)$-part, i.e. $\Pi_{meta}(j+1)$ together with this input and the rule (4.9) computes all legal successor states satisfying the causation rules. Again, by similar considerations as in Theorem 4.2 combined with Theorem 4.7, we conclude that $s^{S'}_{SC,j+1}$ represents one such successor state, since $\texttt{notok}(j+1) \notin S$. This proves (i).

(ii) follows from similar considerations: We assume that $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle, \langle s_j, A_{j+1}, s_{j+1}\rangle\rangle$ is a legal trajectory, and by the induction hypothesis the trajectory $\langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ is represented by an answer set S' such that $s_j = s^{S'}_{SC,j}$. We know that A_{j+1} is an executable action set wrt. s_j, and $\langle s_j, A_{j+1}, s_{j+1}\rangle$ is a legal transition. Again, by the layered construction of $lp^{SC}_{j+1}(\mathcal{P},P)$ together with the Splitting Set Theorem and in analogy to Theorem 4.2, we can conclude that S' can be extended to an answer set S such that $s_{j+1} = s^{S}_{SC,j+1}$ and $\texttt{notok}(j+1) \notin S$. This proves (ii).

(iii) It is sufficient to consider the case, where $\texttt{notok}(j+1) \in S$ and $\texttt{notok}(i) \notin S$ for any $i \leq j$: By rule (4.10) whenever notok(i) is derived, all subsequent states will be saturated, so only the first step where $\texttt{notok}(j+1)$ holds could be a witness for a problematic state.

Now let $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ be a partial trajectory witnessing prefix $\langle A_1 \ldots, A_j\rangle$ of P. We know there is a corresponding answer set S' of $lp^{SC}_{j}(\mathcal{P},P)$ and there could be two reasons for the nonexistence of a transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$:

- A_{j+1} is not an executable action set wrt. s_j, or
- there is no state s_{j+1} such that the causation rules in PD are satisfied wrt. $s_j \cup A_{j+1}$.

While the former is checked by rule (4.9), the latter amounts to the existence of an answer set for the logic program encoded $(F^{SC}_{j+1}(\mathcal{P},P) \setminus F^{SC}_{j}(\mathcal{P},P)) \cup \Pi_{meta}(j+1)$. By similar arguments as in the proof of Theorem 4.7, $\texttt{notok}(j+1) \in S$ holds whenever one of these two conditions applies.

On the contrary, let S be an answer set of $lp^{SC}_{j+1}(\mathcal{P},P)$ such that $\texttt{notok}(j+1) \in S$ and $\texttt{notok}(i) \notin S$ for any $i \leq j$. Then again by the Splitting Set Theorem, we can conclude that there is an answer set S' of $lp^{SC}_{j}(\mathcal{P},P)$ which represents a partial trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ corresponding to the prefix $\langle A_1 \ldots, A_j\rangle$ of P. On the other hand, by construction of $lp^{SC}_{j+1}(\mathcal{P},P)$, $\texttt{notok}(j+1)$ will only be derived in S if

- A_{j+1} is not an executable action set wrt. $s_j = s^{S}_{SC,j}$ by rule (4.9), or
- no legal successor state exists by the causation rules in PD.

Thus, by our assumption that $\texttt{notok}(j+1) \in S$, s_j marks a state where no legal transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$ exists.

□

Note that (iii) is particularly interesting since it indicates that by the answer sets of $lp^{SC}(\mathcal{P}, P)$ we can not only derive all legal trajectories of length n corresponding to P, but also all partial trajectories which, by execution of P, lead to a state where P can not be further executed.

Still, we are not yet completely done. Now, we extend $lp^{SC}(\mathcal{P}, P)$ by the following rules:

- Similar to Step 6 in translation $lp(\mathcal{P})$ above, for goal query q

  $\texttt{goal}: \quad g_1, \ldots, g_k, \texttt{not } g_{k+1}, \ldots, \texttt{not } g_l \ ? \ (n)$

  we add

  $r_{goal}: \quad \texttt{goal :- inS}(\texttt{g}_1, \texttt{n}), \ldots, \texttt{inS}(\texttt{g}_\texttt{k}, \texttt{n}), \texttt{ninS}(\texttt{g}_{\texttt{k+1}}, \texttt{n}) \ldots, \texttt{ninS}(\texttt{g}_1, \texttt{n}).$

- and the constraint

  $c_{secure}: \quad \texttt{:- goal, not notok}(n).$

r_{goal} together with the constraint c_{secure} check exactly the conditions (a) and (b) for insecure plans by allowing only answer sets such that at least one of the conditions is violated. This leads to the following theorem:

Theorem 4.14. *Given a $\mathcal{K}$ planning problem $\mathcal{P} = \langle PD, q\rangle$ and a sequence of action sets $P = \langle A_1, A_2, \ldots, A_n\rangle$, the program $\Pi_{SC} = lp^{SC}(\mathcal{P}, P) \cup \{r_{goal}, c_{secure}\}$ has no answer set if and only if P is a secure plan.*

Proof. ($\Rightarrow$) Assume that P is insecure, i.e. (a) or (b) from above holds. Let us first consider (b), i.e., we assume that there is a state s_j with $j < n$ such that no legal transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$ exists. From Proposition 4.13 (iii) above and by rule (4.9), we can conclude that any such state s_j is witnessed by an answer set S of $lp^{SC}(\mathcal{P}, P)$ such that $\texttt{notok}(n) \in S$. Since therefore constraint c_{secure} is not violated, S is also an answer set of Π_{SC}.

(a) on the other hand means that there is a supporting trajectory for P which does not achieve the goal. Again, by Proposition 4.13 we know that any such supporting trajectory for P corresponds to an answer set of $lp^{SC}(\mathcal{P}, P)$ such that $\texttt{notok}(n) \notin S$. However, since goal can not be derived from r_{goal} (and goal does not appear in any other rule head in Π_{SC}), again constraint c_{secure} is not violated and S is also an answer set of Π_{SC}.

($\Leftarrow$) Now assume that Π_{SC} has an answer set S. Thus, constraint c_{secure} must not be violated, i.e., either $\texttt{goal} \notin S$ or $\texttt{notok}(n) \in S$. By similar considerations as above and from Proposition 4.13 it follows that P must be insecure. □

Note that the answer sets of Π_{SC} (if existing) allow to extract a counterexample for plan security. Let S be an answer set of $lp^{SC}(\mathcal{P}, P)$; we distinguish two cases:

- Assume that there exists a $j \leq n$ such that $\texttt{notok}(j) \in S$. If $j = 0$, then there exists no legal initial state for $\mathcal{P}$. If $j > 0$ and $\texttt{notok}(j-1) \notin S$, then $T = \langle\langle s^S_{SC,0}, A_1, s^S_{SC,1}\rangle, \ldots, \langle s^S_{SC,j-2}, A_j - 1, s^S_{SC,j-1}\rangle\rangle$ marks a partial trajectory such that no legal transition $\langle s^S_{SC,j-1}, A_j, s^S_{SC,j}\rangle$ exists.

- Otherwise $\texttt{notok}(n) \notin S$, and S marks a supporting trajectory $T = \langle\langle s^S_{SC,0}, A_1, s^S_{SC,1}\rangle, \ldots, \langle s^S_{SC,n-1}, A_n, s^S_{SC,n}\rangle\rangle$ such that the goal is not achieved by $s^S_{SC,n}$.

Although this method for checking plan security is universally applicable to any $\mathcal{K}$ planning problem and plan at hand, it is very intricate and obviously expensive in general as it uses full DLPs with head-cycles.

In the next subsection, we will identify security checks for some syntactic subclasses of planning domains which are easier to compute.

4.3.4 Secure Checking for Proper Domains

As we have seen, in arbitrary planning domains the security check is Π^P_2-complete and thus, by widely believed complexity hypotheses, it is not polynomially reducible to a SAT solver or any other computational logic system with expressiveness bounded by NP or co-NP. The checking method proposed in the previous section is computationally expensive and in many cases an "overkill": As shown in Section 3.3, a polynomial reduction from the secure checking problem to co-NP is possible for the class of proper propositional planning problems. This is in particular interesting, as a co-NP check could, by the method proposed in Section 4.2, be integrated into a single DLP computing secure plans at once without having to split the problem into guessing optimistic plans and then checking each plan in a separate computation.

We recall that a planning problem $\mathcal{P}$ is *proper*, if the underlying planning domain *PD* is proper, i.e., given any state s and any set of actions A, deciding whether some legal state transition $\langle s, A, s'\rangle$ exists is possible in polynomial time.

In the $\texttt{DLV}^{\mathcal{K}}$ system, described in the next chapter, we have focused on certain proper propositional planning domains, and we have implemented security checking by polynomial reductions to logic programs with complexity in co-NP.

Note that for any proper planning domain *PD*, there is an algorithm $\mathcal{A}_{PD}$ which, given an arbitrary state s and a set of actions A, decides in polynomial time whether some legal transition $\langle s, A, s'\rangle$ exists. The existence of such an algorithm $\mathcal{A}_{PD}$ is difficult to decide, even in the propositional case, and $\mathcal{A}_{PD}$ is not efficiently constructible under widely accepted complexity beliefs. We thus looked for suitable semantic properties of planning domains which can be ensured by syntactic conditions and allow for a simple (or even trivial) check for the existence of a legal transition $\langle s, A, s'\rangle$, which uniformly works for a class of accepted planning domains.

Let us recall the conditions for insecurity of a plan $P = \langle A_1, \ldots, A_n\rangle$ from above: P is insecure if and only if there is a (partial) trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \langle s_1, A_2, s_2\rangle, \ldots, \langle s_{j-1}, A_j, s_j\rangle\rangle$ such that either:

(a) $j = n$ and the goal is not satisfied in s_j, or

(b) $j < n$ and no legal transition $\langle s_j, A_{j+1}, s_{j+1}\rangle$ exists, i.e.,

  (b1) A_{j+1} is not an executable action set in s_j or

  (b2) A_{j+1} is executable but there is no state s_{j+1} such that $\langle s_j, A_{j+1}, s_{j+1}\rangle$ is a legal transition.

While checking (a) and (b1) is clearly feasible in polynomial time (cf. Lemma 3.1 in Section 3.3), (b2) is the critical (co-NP-complete) part. We will now introduce syntactic subclasses of planning problems, where checking (b2) is easier.

4.3.4.1 `false`-Committed Domains and Security Check $\mathcal{SC}_1$

One such condition is when, informally, the existence of a legal transition $\langle s, A, s' \rangle$ can only be blocked by a causal rule with head `false` or by an (implicit) consistency constraint `:- f,-f`. That is, if such constraints are disregarded, some legal transition $\langle s, A, s' \rangle$ always exists, and otherwise, if some constraint is violated in any such s', no legal transition $\langle s, A, s' \rangle$ exists. As we will see, this condition can be ensured by a syntactic condition which employs stratification on the causation rules.

With this in mind, we develop a security check $\mathcal{SC}_1$, which, given an optimistic plan $P = \langle A_1, \ldots, A_n \rangle$ of length $n \geq 0$ for a planning problem $\mathcal{P}$, rewrites the logic program $lp(\mathcal{P})$ from Section 4.3.1 to a logic program $lp^{SC_1}(\mathcal{P}, P)$ and returns "yes" (i.e., the plan is secure) if this program has no answer set, and "no" otherwise.

The modifications are as follows:

(1) As for checking condition (b1), we substitute Step 4 by a further variant, which is more or less a combination of the original Step 4 from $lp(\mathcal{P})$ (see p. 88) and the alternative Step 4a (see p. 98) from $lp^{SC}(\mathcal{P}, P)$. We call this Step 4b:

Step 4b (Executability Conditions):[25] We again only consider those executability conditions which involve actions in P. So, for any $a(\vec{t})$ in A_j and corresponding executability condition e of the form `executable` $a(\vec{t})$ `if` B in C_R, we introduce the following rules in $lp^{SC_1}(\mathcal{P}, P)$:

1. A fact e_1' of the form
$$a(\vec{t}, j-1).$$
These facts add all actions in P. Note that we do not restrict ourselves to propositional planning problems as in $lp^{SC}(\mathcal{P}, P)$.
2. Rule e_2' is taken from Step 4 of $lp(\mathcal{P})$, i.e. we add a rule e_2' with
$Head(e_2') =$ `exec`$_a(\vec{t}, T_0)$
$Body(e_2')$ consists of the following literals:
   - Each (default negated) type literal in e, i.e., (`not`)$l \in B$ where $l \in \mathcal{L}_{typ}$;
   - (`not`) $b(\vec{t}, T_0)$, where (`not`) $b(\vec{t}) \in B$ and $b(\vec{t}) \in \mathcal{L}_{dyn}$;
   - `next`(T_0,T_1) where T_1 is a new variable;
   - for typing and safety, type information literals for $a(\vec{t})$ and every default literal `not` $(\neg)p(\vec{t}) \in B$ such that $p(\vec{t}) \in \mathcal{L}_{dyn}$, as in the original Step 4 above.
3. Finally, we slightly modify the constraint e_3' from Step 4 which forbids that the action is not executable:
$$\texttt{notex :- } a(\vec{t}, T_0), \texttt{ not exec}_a(\vec{t}, T_0).$$
where T_0, T_1 are new variables.

[25] A slightly different version for the simplified Step 4 for acyclic executability conditions mentioned above has been introduced in [EFL+03a].

Here, `notex` is a new auxiliary predicate which intuitively expresses that the plan P can not be properly executed; its truth allows building a witness for the insecurity of P.

(2) Concerning condition (b2), in any situation where a causal rule with head `false` is violated or a fluent inconsistency arises, an answer set witnessing the insecurity of P should be generated. To this end, the transformation is modified as follows:

Each constraint :- $Body$, `time`(T_1). in $lp(\mathcal{P})$ which stems from a static causal rule of the form `caused false if ...` is rewritten to

$$\begin{array}{l}\texttt{notex} \texttt{ :- } Body, \texttt{next}(T_0, T_1).\\ \texttt{:- } Body\theta.\end{array} \tag{4.11}$$

where θ is a substitution mapping T_1 to 0. Observe that the violations of constraints referring to an initial state do not generate a counterexample which is reflected by changing `time`(T_1) to `next`(T_0, T_1) and by leaving the constraints for timestamp 0 unaffected, expressed by the final constraint.

Each constraint :- $Body$, `next`(T_0, T_1). in $lp(\mathcal{P})$ which stems from a dynamic causal rule `caused false` (`if` ...) `after` ... is rewritten to

$$\texttt{notex} \texttt{ :- } Body, \texttt{next}(T_0, T_1). \tag{4.12}$$

And for each fluent $f(\vec{X})$, the (implicit) consistency constraint (discussed in Footnote 5 in Section 2.1) is transformed to a rule for non-initial states

$$\texttt{notex} \texttt{ :- } f(\vec{X}, T_1), -f(\vec{X}, T_1), \texttt{ next}(T_0, T_1). \tag{4.13}$$

while those for the initial state remain unchanged:

$$\texttt{:- } f(\vec{X}, 0), \ -f(\vec{X}, 0). \tag{4.14}$$

Constraint violations (explicit or implicit) in non-initial states therefore lead to a witnessing answer set containing `notex`.

(3) Finally, for condition (a), the goal constraint :- `not goal_reached`. is modified to

$$\texttt{:- goal\_reached, not notex.} \tag{4.15}$$

We can read the rewritten goal constraint as follows: The constraint is satisfied, and thus the plan P is not secure, if (i) either `notex` is true, which means that some action in P cannot be executed or a constraint is violated when executing the actions in P, or (ii) if `goal_reached` is false, which means that after successfully executing all actions in P, the goal is not established.

Before we can state the informal conditions, under which the security check $\mathcal{SC}_1$ works, more precisely, we need some auxiliary concepts.

Definition 4.4 (constraint-free, constraint- & executability-condition-free shadow). For any planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, let $cfs(PD)$ denote the planning domain which results from PD by dropping all causal constraints with head `false` and interpreting negative fluents as new (positive) fluents, and call it the *constraint-free shadow of PD*. Furthermore, let $cefs(PD)$ denote the planning domain derived from $cfs(PD)$ by omitting all executability conditions and adding `executable a.` for each legal action instance `a`, and call it the *constraint- and executability–condition-free shadow of PD* .

Definition 4.5 (`false`-committed planning domains). We call a planning domain PD `false`-*committed*, if the following conditions hold:

(i) If s is a legal state in PD and A is an action set which is executable in s wrt. PD, then either (i.1) *every* legal transition $\langle s, A, s' \rangle$ in $cfs(PD)$ is also a legal transition in PD, or (i.2) *no* $\langle s, A, s'' \rangle$ is a legal transition in PD, for all states s'' in PD.

(ii) For any state s and action set A in $cefs(PD)$, there exists some legal transition $\langle s, A, s' \rangle$ in $cefs(PD)$.

We call a planning problem $\mathcal{P} = \langle PD, q \rangle$ `false`-committed, whenever PD is `false`-committed.

Example 4.10 (Bridge Crossing with Nondeterminism (cont'd)). Let us reconsider problem $\mathcal{P}_{BCPsec}$ from page 41. As easily verified, $\mathcal{P}_{BCPsec}$ is `false`-committed. Indeed, PD_{BCPsec} contains four occurrences of default negation `not`, via the `inertial` statements (after macro expansion)

```
caused at(X,S) if not -at(X,S) after at(X,S).
caused hasLamp(X) if not -hasLamp(X) after hasLamp(X).
```

and the two statements from expansion of the macro `total`

```
caused hasLamp(X) if not -hasLamp(X) after cross(X), X != joe.
caused -hasLamp(X) if not hasLamp(X) after cross(X), X != joe.
```

all of which are not critical for the existence of a successor state in $cefs(PD_{BCPsec})$, so condition (ii) is guaranteed. (Informally, this is the case, because all cycles in the dependency graph of the respective logic program containing default negation have even length, which is a sufficient condition for answer set existence, resp. successor state existence in our case, cf. [PS94].) As for condition (i), for each consistent state s and action set A, which is executable in s, every legal transition $\langle s, A, s' \rangle$ in $cfs(PD)$ is also a legal transition in PD:

- Except for the `noConcurrency` macro, our domain is essentially free of explicit constraints. Thus, the only constraints possibly violated emerge from $|A| > 1$, where (i.2) holds.

- Furthermore, we have to consider the implicit constraints by possible inconsistencies in successor states for the remaining A with $|A| \leq 1$: For the empty action set $A = \emptyset$, which is trivially executable, obviously the only rules applicable are those from `inertial` statements: Here, no inconsistencies can arise, i.e. condition (i.1) holds. On the other hand, for all other action sets involving a single action, one can easily verify that

  1. for action set $A = \{a\}$ where $a \in \{\texttt{cross}(x) \mid x \in \{\texttt{jack}, \texttt{william}, \texttt{averell}\}\}$, the `total` rule is applicable and the two possible successor states which either contain $\texttt{hasLamp}(x)$ or $\neg\texttt{hasLamp}(x)$ but not both, are both consistent;
  2. for all other (single action) action sets A, a unique consistent successor state exists.

  Therefore, we have (i.1) for all action sets A with $|A| \leq 1$.

Consider the secure plan P_2 from page 41:

$$P_2 = \langle\ \{\texttt{takeLamp(joe)}\},\ \{\texttt{crossTogether(joe, jack)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, william)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, averell)}\}\ \rangle$$

Indeed, an attempt to build an answer set S of $lp^{SC_1}(\mathcal{P}_{BCP_{sec}}, P_2)$ fails: starting from any initial state, the actions in A_i are always executable and no constraint is violated, thus `notex` cannot be included in S. Thus, in order to satisfy the rewritten goal constraint `:- goal_reached, not notex.`, `goal_reached` must not be included in S. As easily seen, the atoms $\texttt{at}(\texttt{joe}, \texttt{across})$, $\texttt{at}(\texttt{jack}, \texttt{across})$, $\texttt{at}(\texttt{william}, \texttt{across})$, $\texttt{at}(\texttt{averell}, \texttt{across})$ are included in any answer set S. This, however, means that `goal_reached` has to be included in S, which is a contradiction. Thus, no answer set S of $lp^{SC_1}(\mathcal{P}_{BCP_{sec}}, P_2)$ exists, which means that the plan P_2 is secure.

Let us now modify the number of steps in the goal to $i = 5$, and consider the optimistic plan P_1

$$P_1 = \langle\ \{\texttt{crossTogether(joe, jack)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, william)}\},\ \{\texttt{cross(joe)}\},\ \{\texttt{crossTogether(joe, averell)}\}\ \rangle$$

In this case we can build an answer set of $lp^{SC_1}(\mathcal{P}_{BCP_{sec}}, P_1)$ starting from an initial state in which `joe` does not have the lamp, by including at each stage the literals that are enforced. Then, the first action is no longer executable and `notex` can be derived in Step 4b. Thus, the constraint `:-goal_reached, not notex.` in $lp^{SC_1}(\mathcal{P}_{BCP_{sec}}, P_1)$ is satisfied, admitting an answer set which witnesses the insecurity of P_1. Hence, the check outputs "no", i.e., the plan is not secure. ◇

To show that the security check $\mathcal{SC}_1$ works properly for all `false`-committed planning domains, we need the notions of soundness and completeness for security checks.

Definition 4.6 (security check). A *security check* for a class of planning domains $\mathcal{PD}$ is any algorithm which takes as input a planning problem $\mathcal{P}$ in a planning domain from the class $\mathcal{PD}$ and an optimistic plan P for $\mathcal{P}$, and outputs "yes" or "no." A security check is *sound*, if it reports "yes" only if P is a secure plan for $\mathcal{P}$, and it is *complete* if it reports "yes" whenever P is a secure plan for $\mathcal{P}$.

In other words, for a sound security check only "yes" can be trusted, while for a complete security check only "no" can be trusted.

Theorem 4.15. *The security check $\mathcal{SC}_1$ is sound and complete for the class of* `false`*-committed planning domains.*

Proof. We outline the proof, but omit the details. Let $P = \langle A_1, \ldots, A_n \rangle$ be an optimistic plan for a planning problem $\mathcal{P}$ in a `false`-committed planning domain *PD*.

(Soundness) Suppose that P is not secure. This means that an initial state s_0 and a trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$ in *PD* (where $0 \leq j \leq n$) exist, such that one of the conditions (a) or (b) for plan security stated at the beginning of this section is violated. We then can build an answer set S of the program $lp^{SC_1}(\mathcal{P}, P)$, in which, starting from s_0, respective literals are included which correspond to the legal transitions in T as in $lp(\mathcal{P})$. We consider the three cases:

Suppose first that condition (b1) is violated, i.e., some action $a(\vec{c})$ in the action set A_j of P is not executable. Then, no rule with head $a(\vec{c}, j)$ fires, and thus we may add `notex` to S, as it can be derived from the rule `notex :- not` $a(\vec{c}, j)$. By (ii) of `false`-committedness for *PD*, we can add literals for the stages $j+1$, ..., n modeling transitions in *cefs*(*PD*) to S such that we obtain an answer set of $lp^{SC_1}(\mathcal{P}, P)$.

Suppose next that condition (b2) is violated, i.e., no successor state exists. By (ii) of `false`-committedness for *PD*, we can add literals to S modeling a legal transition $\langle s_j, A_{j+1}, s_{j+1} \rangle$ in *cfs*(*PD*), and by (i) of `false`-committedness for *PD*, `notex` will be derived, as in *PD* some rule with head `false` fires or opposite fluent literals `f`, `-f` are in S. Using (ii) again, we can add literals for the remaining stages $j+2$, ..., n modeling transitions in *cefs*(*PD*), such that we obtain an answer set S of $lp^{SC_1}(\mathcal{P}, P)$.

Suppose finally that condition (a) is violated. That is, $j = n$ and the goal is not satisfied by s_n. Then, the rule with head `goal_reached` is not applicable, the modified goal constraint is satisfied, and an answer set S exists. Note that this also includes the case $n = 0$.

In any of these three cases, an answer set S of $lp^{SC_1}(\mathcal{P}, P)$ exists, and $\mathcal{SC}_1(\mathcal{P}, P)$ outputs "no."

(Completeness) Suppose $\mathcal{SC}_1(\mathcal{P}, P)$ outputs "no," i.e., $lp^{SC_1}(\mathcal{P}, P)$ has some answer set S. Then, either `notex` $\in S$ or `goal_reached` $\notin S$ must hold. In the former case, `notex` must be derived either i) from some rule r : `notex :- not` $a(\vec{c}, j)$., or ii) from some rule `notex :- ... time(j).` corresponding to a rewritten constraint with head `false` or a consistency constraint for strong negation. Let r be such that j is minimal. Then S encodes with respect to the stages $0, \ldots, j-1$ a trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-1}, A_j, s_j \rangle \rangle$, such that A_{j+1} is not executable in s_j wrt. *PD*. In case i), we immediately obtain that condition (b1) of security is violated and hence that P is not secure. In case ii), a trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{j-2}, A_{j-1}, s_{j-1} \rangle \rangle$ in *PD* exists such that executing A_j in s_{j-1} wrt. *cfs*(*PD*) as encoded in S leads to a state s_j which violates some constraint of *PD* with head `false` or contains opposite literals. By item (i) of `false`-committedness for *PD*, we can conclude that no legal transition $\langle s_{j-1}, A_j, s_j \rangle$ exists in *PD*, which violates condition (b2) of security. On the other hand, if `goal_reached` $\notin S$ while `notex` $\notin S$, then S encodes a trajectory $T = \langle \langle s_0, A_1, s_1 \rangle, \ldots, \langle s_{n-1}, A_n, s_n \rangle \rangle$ wrt. *PD* such that in the final state s_n the goal is false, i.e., condition (a) of security is violated. That is, in all cases P is not secure. □

Now that we have introduced the class of `false`-committed planning domains, we look for syntactic conditions on planning domains which can be efficiently checked and guarantee `false`-committedness. One such condition can be obtained by imposing stratification on causation rules as follows: For any causation rule r of the form (3.2) let $lp(r)$ be the corresponding logic programming rule f :- $b_1, \ldots, b_k$, `not` $b_{k+1}, \ldots,$ `not` b_l. which emerges by skipping the `after`-part.

Definition 4.7 (stratified planning domain). A planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$ is *stratified*, if the logic program Π_{PD} consisting of all rules $lp(r)$, where $r \in C_R$ has $\mathtt{h}(r) \neq$ `false` and a non-empty `if`-part, is stratified in the usual sense (and strongly negated atoms are treated as new atoms).

For example, the planning domain PD_{btc} described in Section 6.3.1 is stratified whereas PD_{btuc} is obviously not.

It is easy to see that stratified planning domains are `false`-committed. Indeed, since any constraint-free stratified logic program is guaranteed to have an answer set, item (ii) of Definition 4.5 holds. Furthermore, for each legal state s and action set A which is executable in s wrt. PD, there exists a single candidate state s' for a legal transition $\langle s, A, s' \rangle$ in $cfs(PD)$, which is computed by evaluating a subset of the rules of Π_{PD}; this transition is not legal in PD if s' violates some constraint in C_R with head `false` or introduces inconsistency. Note that stratified planning domains PD are proper.

Corollary 4.16. *The security check $\mathcal{SC}_1$ is sound and complete for the class of stratified planning domains.*

A possible extension of Corollary 4.16 allows for limited usage of unstratified causation rules. For example, pairs

```
inertial f.
inertial -f.
```

of positive or negative inertia rules for the same ground fluent `f`, which amount to the rules

r_f^+ : `caused f if not -f after f.`
r_f^- : `caused -f if not f after -f.`

violate stratification. Nevertheless, pairwise inertia for a fluent `f` can be allowed safely, if each of the two rules together with the remainder of the planning domain is stratified. That is, we check for stratification of the two sub-domains that result from the planning domain PD by omitting the positive and negative inertia rules for f, denoted by PD^{-f} and PD^{+f}, respectively. If both PD^{-f} and PD^{+f} are stratified, then $\mathcal{SC}_1$ is sound and complete for PD. This holds because in any state s, at most one of the rules r_f^+ and r_f^- can be active with respect to s.

We can further extend this to multiple pairs of ground inertia rules, where combinations of positive and negative inertia rules have to be checked. We go one step further and extend it to mux-stratified planning domains, which we define next.

Two causation rules r_0, r_1 in PD are a *mutually exclusive pair* (*mux-pair*), if their `after`-parts are not simultaneously satisfiable in any state s and for any executable action set A wrt. s in PD.

Definition 4.8 (mux-stratified planning domains). Let *PD* be a planning domain and $E = \{(r_{i,0}, r_{i,1}) \mid 1 \leq i \leq n\}$, $n \geq 0$, a set of mux-pairs in *PD*. Then, *PD* is called *mux-stratified* wrt. E, if each planning domain *PD'* that results from *PD* by removing one of the rules $r_{i,0}$ and $r_{i,1}$ for all $i = 1, \ldots, n$ is stratified.

Notice that E does not necessarily contain all mux-pairs occurring in *PD*; we may even choose $E = \emptyset$, where the notion of mux-stratified coincides with stratified planning domain.

Note that E induces a bipartite graph G_E, whose vertices are the rules occurring in E and whose edges are the pairs in E. The removal sets for building *PD'* which need to be considered are given by the maximal independent sets of G_E. There may be exponentially many such sets, and thus the cost for (simple) mux-stratification testing grows fast.

We now establish the following result.

Theorem 4.17. *Every planning domain PD which is mux-stratified wrt. some set of mux-pairs E is* `false`*-committed.*

Proof. Consider any state s and executable action set A wrt. s in *PD*. Denote by $active(s, A, PD)$ the set of all ground rules in $\Pi_{PD}{\downarrow}$ which correspond to instances r' of causation rules in *PD* such that the `after`-part of r' is true wrt. s, A and the answer set M of the background knowledge.

Then, we claim that $active(s, A, PD)$ is stratified, i.e., its (ground) dependency graph does not contain a negative cycle. Indeed, towards a contradiction assume that the (ground) dependency graph of $active(s, A, PD)$ contains a negative cycle C. Then, C involves only rules which correspond to instances of causation rules not occurring in E, and rules which correspond to instances of causation rules from $r_{1,i_1}, \ldots, r_{n,i_n}$, where $E = \{(r_{i,0}, r_{i,1}) \mid 1 \leq i \leq n\}$ and $i_j \in \{0, 1\}$, for all $j = 1, \ldots, n$. (A rule R is involved in all edges $l_1 \rightarrow l_2$ of the dependency graph, where $l_1 \in H(R)$ and $l_2 \in B(R)$.) This means that C is also present in the ground dependency graph of $\Pi_{PD'}$ for some *PD'* which results from *PD* by removing the causation rules that correspond to instances of $r_{1,1-i_1}$, $r_{2,1-i_2}$, $\cdots$, $r_{n,1-i_n}$. Consequently, the (non-ground) dependency graph of $\Pi_{PD'}$ contains a negative cycle. This, however, contradicts that *PD* is mux-stratified wrt. E; the claim is proved.

Since the ground program $active(s, A, PD)$ is stratified, it is easily seen that conditions (i) and (ii) of `false`-committedness hold for s. Since s was arbitrary, it follows that *PD* is `false`-committed. □

By combining Theorems 4.15 and 4.17, we obtain the following corollary.

Corollary 4.18. *The security check $\mathcal{SC}_1$ is sound and complete for the class of mux-stratified planning domains, and in particular if E consists of opposite ground* `inertial`*-rules.*

A generalization of the result in Corollary 4.18 to sets E of non-ground opposite inertial rules fails. The reason is that in this case multiple transition candidates $\langle s, A, s' \rangle$ may exist in $cfs(PD)$, which correspond to multiple answer sets of the program $active(s, A, PD)$. However, some of these might not be legal in *PD*, and condition (i) of `false`-committedness may be violated.

Take for instance the following short example:

Example 4.11. Let $\mathcal{P} = \langle PD, q\rangle$ be given by background knowledge $\Pi = \{\texttt{bk(a).}\ \texttt{bk(b).}\}$. and the following $\mathcal{K}$ program:

```
fluents:   f(X) requires bk(X).
           g.
initially: -f(a). f(b).
always:    g.
           inertial f(X).
           inertial -f(X).
           caused f(a) if not f(b) after f(b).
           caused -f(b) if not -f(a) after f(b).
           caused false if -f(a), f(b) after f(b).
goal:      g?(1)
```

This planning domain has the (non-obvious) secure plan $P = \langle\emptyset\rangle$. Secure check $\mathcal{SC}_1$ does not apply here, since the domain is not `false`-committed: We obtain $cfs(PD)$ $(=cefs(PD))$ essentially by ignoring the last causation rule. Starting with the single initial state $s_0 = \{\texttt{-f(a)}, \texttt{f(b)}\}$, the domain $cfs(PD)$ allows for two legal transitions $t_1 = \langle s_0, \emptyset, \{\texttt{-f(a)}, \texttt{f(b)}\}\rangle$ and $t_2 = \langle s_0, \emptyset, \{\texttt{f(a)}, \texttt{-f(b)}\}\rangle$ where t_1 is not legal in PD as it violates condition (i) in Definition 4.5. Furthermore, $\mathcal{P}$ is also not mux-stratified wrt. opposite ground inertia rules. However, as easily verified, ignoring either of the *non-ground* inertia rules would lead to a stratified domain in both cases. This shows that checking mux-stratification for opposite inertia rules on the non-ground level is not sufficient to guarantee `false`-committedness.

Indeed, if we apply secure check $\mathcal{SC}_1$ to this example it will misleadingly answer "no" when testing $P = \langle\emptyset\rangle$. Therefore, the example shows that $\mathcal{SC}_1$ is too cautious in some cases and proves incompleteness of $\mathcal{SC}_1$ in general. ◇

The next example shows that planning problems not even need to involve default negation or constraints in order to prove incompleteness of $\mathcal{SC}_1$:

Example 4.12. Let $\mathcal{P} = \langle PD, q\rangle$ be given by the following $\mathcal{K}$ program (with empty background knowledge Π):

```
fluents:   f. g.
always:    caused g if f.
           caused -g.
           total f.
goal:      -g?(1)
```

There is one legal initial state $s_0 = \{\texttt{-f}, \texttt{-g}\}$ in this domain. In $cfs(PD)$ $(=cefs(PD))$, we have two legal transitions $t_1 = \langle s_0, \emptyset, s_0\rangle$ and $t_2 = \langle s_0, \emptyset, \{\texttt{f}, \texttt{g}, \texttt{-g}\}\rangle$. However, only t_1 is also legal in PD, so `false`-committedness does not hold by violation of condition (i) in Definition 4.5. Again, $\mathcal{SC}_1$ will wrongly reject the secure plan $P = \langle\emptyset\rangle$. ◇

In fact, $\mathcal{SC}_1$ is not even sound in general, witnessed by another simple example:

Example 4.13. Let $\mathcal{P} = \langle PD, q\rangle$ be given by the following $\mathcal{K}$ program (with empty background knowledge Π):

```
fluents:    f. g.
initially:  total g.
always:     f.
            caused g if not g after g.
goal:       f?(1)
```

Obviously, this domain has two legal initial states $s_{0,0} = \{\mathtt{f}, \mathtt{g}\}$ and $s_{0,1} = \{\mathtt{f}, \neg\mathtt{g}\}$. Furthermore, $P = \langle\emptyset\rangle$ is an optimistic plan witnessed by the trajectory $\langle s_{0,1}, \emptyset, \{\mathtt{f}\}\rangle$. However, there is no legal transition starting with $s_{0,0}$, i.e. P is not secure. As easily verified, this is not recognized by $\mathcal{SC}_1$, which returns that P is a secure plan. Clearly, *PD* is not `false`-committed due to violation of condition (ii) of Definition 4.5. ◇

The DLV$^{\mathcal{K}}$ system provides limited support for testing mux-stratification, which currently works for the set E consisting of all pairs of rules (r_0, r_1) such that r_0 and r_1 contain "opposite" literals in the `after`-parts of ground causation rules (which covers in particular opposite inertia rules). Formally, by opposite literals we mean that for rules r_0 and r_1

1. there exist $l \in \mathsf{pre}^+(r_0)$, such that $\neg.l \in \mathsf{pre}^+(r_1)$, or
2. $(\mathsf{pre}^+(r_0) \cap \mathsf{pre}^-(r_1)) \cup (\mathsf{pre}^+(r_1) \cap \mathsf{pre}^-(r_0)) \neq \emptyset$

Clearly, all such pairs of rules form mux-pairs in any planning domain.

Notice, that deciding whether a given pair of causation rules (r_0, r_1) is a mux-pair in a given planning domain is intractable in general.

4.3.4.2 Serial Planning Domains and Security Check $\mathcal{SC}_2$

Besides $\mathcal{SC}_1$, we provide an alternative security check $\mathcal{SC}_2$ for handling other classes of proper planning domains. This check somehow combines the ideas from the general secure check $lp^{SC}(\mathcal{P}, P)$ and the simpler secure check $lp^{SC_1}(\mathcal{P}, P)$ to a further class of proper planning domains, so-called *serial* planning domains. These are planning domains where in each state s for each action set A which

1. satisfies at least one `executable` statement, and
2. does not violate any `nonexecutable` statement
   (more precisely, any rule of the form `caused false after B`)

at least one legal trajectory $\langle s, A, s'\rangle$ is guaranteed to exist.

The check $\mathcal{SC}_2$ is obtained by the following modifications of $lp^{SC_1}(\mathcal{P}, P)$, resulting in a program $lp^{SC_2}(\mathcal{P}, P)$:

1. Each literal `notex` from $lp^{SC_1}(\mathcal{P}, P)$ is extended by a timestamp as follows:
   (a) In Step 4b from above we write `notex`(T_0) in the head of e'_3.
   (b) For all other rules in $lp^{SC_1}(\mathcal{P}, P)$ with $Head(r) =$ `notex` we change the head to `notex`(T_1) instead.
   (c) We change the final constraint (4.15) from $lp^{SC_1}(\mathcal{P}, P)$ to

:- goal_reached, not notex(l).

where l is the plan length of $\mathcal{P} = \langle PD, Q?(l)\rangle$.

2. Furthermore, we add a rule

notex(T_1) :- next(T_0, T_1), notex(T_0).

which propagates violations of plan security to the successor states.

3. The head notex(T_1) of

(a) each rule of the form (4.11) or (4.12) which stems from a causation rule such that h(r) = false and the if-part is *not empty*, and

(b) each rule of the form (4.13) which stems from a consistency rule

is shifted to the negative body such that

notex(T_1) :- *Body*.

is rewritten to

:- *Body*, next(T_0, T_1), not notex(T_0).

This shift means that the violation of a constraint on the successor state s' is tolerated, and we eliminate s' as a counterexample to the security of the plan. Here, we have to add not notex(T_0) in the negative body of the constraint, because a constraint shall not eliminate counterexamples which stem from problems in previous states.

Remark 4.4. *By the labeling of predicate* notex *we achieve a machinery similar to the general secure check $lp^{SC}(\mathcal{P}, P)$. The derivation of* notex(t) *can be usefully exploited for finding counterexamples which indicate time step t at which an error occurred.*

We will see that the check $\mathcal{SC}_2$ works for the following class of planning domains.

Definition 4.9 (serial planning domains). [26] A planning domain *PD* is *serial*, if it has the following property: Let s be a state, and A be a set of actions in *PD*. Whenever

(i) A is an executable action set in s wrt. *PD*, and,

(ii) no rule of the form caused false after B.[27] is violated in s

then some legal transition $\langle s, A, s'\rangle$ is guaranteed to exist

We call a planning problem $\mathcal{P} = \langle PD, q\rangle$ *serial*, whenever *PD* is serial.

Given a state s and a set of actions A both conditions (i) and (ii) can be checked in polynomial time and therefore such domains/problems are obviously proper. The following can be observed:

[26] Note that this definition of serial domains covers a broader class of proper domains than the corresponding definition in [EFL+03a].

[27] In particular, this applies to nonexecutable statements.

Theorem 4.19. *The security check $\mathcal{SC}_2$ is sound and complete for serial planning domains.*

The proof argument of this result is similar to the proof of Theorem 4.15 combined with the ideas from Proposition 4.13, and we thus omit it.

Example 4.14. Let $\mathcal{P} = \langle PD, q \rangle$ be given by the following $\mathcal{K}$ program (with empty background knowledge Π):

```
fluents:   f. g.
actions:   a.
always:    executable a if f.
           caused g after a.
           total f.
           caused false if  -f.
goal:      g? (2)
```

The final two causation rules in the `always`-section are semantically equivalent to `caused f.` As easily verified, the domain is serial. Let us now consider the (secure) plan $P = \langle \emptyset, \{\mathtt{a}\} \rangle$. The check program $lp^{SC_2}(\mathcal{P}, P)$ for this plan looks as follows.

```
% Auxiliary predicates:
time(0). time(1). time(2).
next(0,1). next(1,2).

% ''inherit'' notex:
notex(T1) :- notex(T0), next(T0,T1).

% Add the plan:
a(1).

% executable a after f.
exec_a(T0) :- f(T0),next(T0,T1).
notex(T0) :- a(T0), not exec_a(T0).

% caused g after a.
g(T1) :- a(T0), next(T0,T1).

% total f.
f(T1) v -f(T1) :- time(T1).

% caused false if -f.
:- -f(0).
:- -f(T1), time(T1), next(T0,T1), not notex(T0).

% goal: g? (2)
goal_reached :- g(2).
:- goal_reached, not notex(2).
```

Indeed, this program has no answer set and thus, the plan P is secure. ◇

Remark 4.5. *We remark that the additional timestamps are essential for the correct working of secure check $\mathcal{SC}_2$: Simply taking $\mathcal{SC}_1$ and shifting* `notex` *as in item 3. on page 114 is not sufficient in general. This applies, for instance, to Example 4.14 above where this simplified version of $\mathcal{SC}_2$ from [EFL+03a] does not work.*

A syntactic class which guarantees serial domains are for instance stratified planning domains PD which contain no rules r such that $\mathtt{h}(r) = \mathtt{false}$ and which do not employ strong negation. The serial property is preserved if we also allow arbitrary totalization

statements and limited use of strong negation, e.g. if either all occurrences of a fluent are strongly negated or none is. Note that such planning domains are not `false`-committed in general.

We remark that the problems from Example 4.11 and Example 4.12 are serial and therefore can be correctly solved using security check $\mathcal{SC}_2$. However, Example 4.13, is clearly not serial, so $\mathcal{SC}_2$ is not applicable, and unsound in general. We leave it as an exercise for the reader to show that for Example 4.13 $P = \langle\emptyset\rangle$ will wrongly be accepted.

Note that $\mathcal{SC}_2$ is not subsuming $\mathcal{SC}_1$. There are also domains, where $\mathcal{SC}_1$ applies rather than $\mathcal{SC}_2$:

Example 4.15. Let $\mathcal{P} = \langle PD, q\rangle$ be given by the following $\mathcal{K}$ program (again with empty background knowledge Π):

```
fluents:   f. g. abs.
initially: g.
always:    total f after g.
           caused abs after -f.
           caused false if abs.
           caused f after f.
goal:      f?(2)
```

Let us consider the optimistic plan $P = \langle\emptyset, \emptyset\rangle$, which is not secure: There is one legal initial state $s_0 = \{\mathtt{g}\}$ in this domain. In $cfs(PD)$ $(=cefs(PD))$, we have two legal trajectories of length 2: $T_1 = \langle\langle s_0, \emptyset, \{\mathtt{f}\}\rangle, \langle\{\mathtt{f}\}, \emptyset, \{\mathtt{f}\}\rangle\rangle$ and $T_2 = \langle\langle s_0, \emptyset, \{\mathtt{f}\}, \langle\{\mathtt{-f}\}, \emptyset, \{\mathtt{abs}\}\rangle\rangle\rangle$. However, T_2 is invalid for PD: State $\{\mathtt{-f}\}$ does not allow for *any* legal transition to a successor state in PD. Thus, the domain is not serial, and indeed, $\mathcal{SC}_2$ fails by incorrectly accepting plan P. However, any state including $\mathtt{-f}$ satisfies condition (i.2) in Definition 4.5 and on all other states condition (i.1) applies; condition (ii) holds as well. Thus, PD is `false`-committed and $\mathcal{SC}_1$ works properly, rejecting P. ◇

We see that $\mathcal{SC}_2$ might be too brave in certain cases, where SC_1 works. On the other hand, $\mathcal{SC}_2$ may also be profitably combined with $\mathcal{SC}_1$ in order to enlarge classes for which security checking is supported, as shown next.

4.3.4.3 Incomplete Security Checking

The DLV$^{\mathcal{K}}$ system described in the next chapter provides secure checks $\mathcal{SC}_1$ and a simplified version of $\mathcal{SC}_2$ as described in [EFL$^+$03a], and the system design easily allows the incorporation of further security checks.

We may combine (fast) security checks which are sound and security checks which are complete to obtain checks which return the correct answer if possible, and leave the answer open otherwise. This is similar to the use of incomplete constraint solvers in constraint programming, which return either "yes," "no," or "unknown" if queried about satisfiability of a constraint; the obvious requirement is that the answer returned does not contradict the correct result.

Suppose that we have a suite of security checks $\mathcal{SC}_1, \ldots, \mathcal{SC}_n$, where $\mathcal{SC}_1, \ldots, \mathcal{SC}_j$, for some $j \leq n$, are known to be sound for a class of planning domains $\mathcal{PD}$ and $\mathcal{SC}_k, \ldots, \mathcal{SC}_n$,

for some $k \leq n$, are known to be complete for $\mathcal{PD}$. Then, we can combine them to the following test $\mathcal{T}$:

$$\mathcal{T}(\mathcal{P},P) = \begin{cases} \text{“yes”}, & \text{if } \mathcal{SC}_i(\mathcal{P},P) = \text{“yes,” for some } i \in \{1,\ldots,j\}; \\ \text{“no”}, & \text{if } \mathcal{SC}_i(\mathcal{P},P) = \text{“no,” for some } i \in \{k,\ldots,n\}; \\ \text{“unknown”}, & \text{otherwise.} \end{cases}$$

Observe that in the “yes” case of $\mathcal{T}$, $\mathcal{SC}_i(\mathcal{P},P) =$ “yes” must hold for all $i \in \{k,\ldots,n\}$, and symmetrically in the “no” case that $\mathcal{SC}_i(\mathcal{P},P) =$ “no,” for all $i \in \{1,\ldots,j\}$; this can be used for checking integrity of the sound or complete security checks involved.

Note that we can always use a dummy complete security check which reports “yes” on every input. By merging the “unknown” case into the “no” case, we thus can combine sound security checks $\mathcal{SC}_1,\ldots,\mathcal{SC}_j$ to another, more powerful sound security check $\mathcal{SC}$ for the class $\mathcal{PD}$. In particular, if $\mathcal{SC}_1,\ldots,\mathcal{SC}_j$ are known to exhaust all secure plans, then $\mathcal{SC}$ is a sound and complete security check for $\mathcal{PD}$.

4.3.5 Secure Planning

Our general secure check $lp^{SC}(\mathcal{P},P)$ from Section 4.3.3 solves a problem at the second level of the PH. Thus, in general the best we can do in order to compute secure plans for a given planning problem $\mathcal{P}$ is interleaving optimistic plan generation using the translation $lp(\mathcal{P})$ with checking each resulting plan P using one of our secure checks $lp^{SC}(\mathcal{P},P)$, $lp^{SC_1}(\mathcal{P},P)$, or $lp^{SC_2}(\mathcal{P},P)$. Such an interleaved computation is implemented in the DLV$^{\mathcal{K}}$ planning system, which we will present in the next chapter.

On the other hand, the programs $lp(\mathcal{P})$ and each of the easier check programs $lp^{SC_1}(\mathcal{P},P)$ and $lp^{SC_2}(\mathcal{P},P)$ are HEDLPs, i.e., they represent an NP guess for optimistic plans and respective checks in co-NP. So we can integrate guessing optimistic plans by program $lp(\mathcal{P})$ and check programs $lp^{SC_1}(\mathcal{P},P)$ or $lp^{SC_2}(\mathcal{P},P)$ into a single program $\Pi^{\mathcal{P}}_{sec}$ with slight modifications by the method from Section 4.2.3.

Integrated Secure Planning - Guess Part:

Note that $lp^{SC_1}(\mathcal{P},P)$ and $lp^{SC_2}(\mathcal{P},P)$ both check action executability and goal achievement for all possible trajectories of P. Hence, we can simplify the guessing part:

The idea is not to guess an optimistic plan, but an arbitrary sequence of actions, independent of state information. Let $\Pi^{\mathcal{P}}_{guess}$ be the program obtained from only the rules e'_1 of Step 4 in $lp(\mathcal{P})$ plus background knowledge Π. Then, the answer sets of $\Pi^{\mathcal{P}}_{guess}$ represent any possible sequence of actions in $\mathcal{P}$ enriched by the unique answer set M of Π. $\Pi^{\mathcal{P}}_{guess}$ will serve as guess program for an integrated encoding in the sense of Section 4.2.3.

Note that, in order to prune the number of guesses, we still might want to have only action sequences which actually correspond to optimistic plans as candidates. To this end, we can use an alternative guess program based on $lp(\mathcal{P})$: We remark that $lp(\mathcal{P})$ does not satisfy the splitting condition on $lp(\mathcal{P}) \cup lp^{SC_i}(\mathcal{P},P)$, where $i \in \{1,2\}$. Therefore, in order to use $lp(\mathcal{P})$ as a basis for the integrated program $\Pi^{\mathcal{P}}_{sec}$, we first have to rename fluent literals in $lp(\mathcal{P})$ such that they do not conflict with the fluent names in $lp^{SC_i}(\mathcal{P},P)$, i.e. we rename any predicate $f(\vec{x},t)$ in $lp(\mathcal{P})$ emerging from a fluent $f(\vec{x})$ to $f'(\vec{x},t)$. This modified

program, $\Pi^{\mathcal{P}}_{guessOpt}$, can then be used alternatively instead of $\Pi^{\mathcal{P}}_{guess}$. We remark that no such renaming is necessary for type literals, since we conceive background knowledge Π as part of the guess.

Integrated Secure Planning - Check Part:

We slightly modify $lp^{SC_i}(\mathcal{P}, P)$, $i \in \{1, 2\}$, by omitting the rules e'_1 from Step 4b, since the input is no longer a fixed plan to be checked but given by the individual answer sets of $\Pi^{\mathcal{P}}_{guess}$. We denote the modified programs by $\Pi^{\mathcal{P}}_{\mathcal{SC}_1}$ or $\Pi^{\mathcal{P}}_{\mathcal{SC}_2}$, respectively.

For integrating these check programs, we obtain $\Pi'^{\mathcal{P}}_{\mathcal{SC}_1}$ ($\Pi'^{\mathcal{P}}_{\mathcal{SC}_2}$, resp.) from $\Pi^{\mathcal{P}}_{\mathcal{SC}_2}$ ($\Pi^{\mathcal{P}}_{\mathcal{SC}_2}$, resp.) as defined in Theorem 4.6.

In total, we obtain the following results from Theorems 4.6, 4.15 and 4.19:

Proposition 4.20. *Let $\mathcal{P}$ be a* `false`*-committed planning problem. Then, the answer sets of*

$$\Pi^{\mathcal{P}}_{sec,\mathcal{SC}_1} = \Pi^{\mathcal{P}}_{guess} \cup \Pi'^{\mathcal{P}}_{\mathcal{SC}_1}$$

correspond with the secure plans of $\mathcal{P}$.

Proposition 4.21. *Let $\mathcal{P}$ be a serial planning problem. Then, the answer sets of*

$$\Pi^{\mathcal{P}}_{sec,\mathcal{SC}_2} = \Pi^{\mathcal{P}}_{guess} \cup \Pi'^{\mathcal{P}}_{\mathcal{SC}_2}$$

correspond with the secure plans of $\mathcal{P}$.

Both propositions above analogously hold for the alternative guess program $\Pi^{\mathcal{P}}_{guessOpt}$.

As pinpointed above, for general $\mathcal{P}$, due to the inherent complexity of security checking, no such integrated encoding $lp(\mathcal{P})_{sec}$ exists.

4.3.6 Secure Optimal Planning:

As for optimal secure planning, things are getting more involved. In principle, the secure checking methods discussed above carry over to planning with action costs in a straightforward way, and optimal (or admissible) secure plans can be similarly computed by answer set programming.

However, we are bound to interleaved computation here, since checking plan optimality, can not simply be integrated for secure plans.

Note that checking whether a *cheaper* plan of fixed length i than a given plan P exists amounts to deciding admissible plan existence (wrt. $cost_{\mathcal{P}}(P) - 1$). This problem is NP-complete (cf. Theorem 3.12 in Section 3.3), i.e., not harder than checking general plan existence for fixed plan length. Analogously, checking whether a cheaper *secure* plan of length i exists is again Σ^P_3-complete in general and Σ^P_2-complete for proper planning domains. So, for proper domains, like the serial and `false`-committed domains mentioned above, a NP program for checking plan insecurity could be combined with an additional check whether a cheaper secure plan exists in a single Σ^P_2 check program, using the same building blocks as the encodings above. As easily recognized, the immediately resulting algorithm for computing optimal secure plans for proper domains lies in Σ^P_3MV: So, a

naive algorithm in worst case has to check all possible plans by means of an expensive Π_2^P check (checking plan security plus non-existence of a cheaper secure plan), even for proper domains.

However, combining results from Theorems 3.6 and 3.13, we can conclude with the following corollary:

Corollary 4.22 (Complexity of optimal secure planning in proper domains). *For plans of fixed length l, computing an optimal secure plan for $\langle PD, Q\,?\,(l)\rangle$ in $\mathcal{K}^c$ is $\mathrm{F}\Delta_3^P$-complete, if PD is proper.*

Proof. (Sketch) Membership can be easily shown from the considerations above and the proof of Theorem 3.13, where compared with the original proof we can, by Theorem 3.6, employ a Σ_2^P oracle, since PD is proper. The hardness part can be shown starting from a similar idea like the reduction in Theorem 3.13, and making appropriate modifications. □

It is widely believed that (unless the PH collapses) problems in Δ_3^P are a proper subset of Σ_3^P and therefore also that the inclusion $\mathrm{F}\Delta_3^P \subseteq \Sigma_3^P\mathrm{MV}$ is strict. Indeed, we can find a more sophisticated algorithm for computing optimal secure plans, as it has been implemented in the $\texttt{DLV}^{\mathcal{K}}$ planning system which will be presented in Chapter 5:

To find an optimal secure plan for a $\mathcal{K}^c$ planning problem $\mathcal{P}$, we proceed with the following interleaved computation.

A0) initialize $currentBestPlan := nil$ and $currentBestCost := \infty$.

A1) Compute candidate optimistic plans P admissible wrt. $currentBestCost - 1$ (e.g. by using the program $lp^w(\mathcal{P})$) and their costs $cost_{\mathcal{P}}(P)$, and check each plan in an interleaved manner:

A2) While **A1** produces plans, check for each plan P whether it is secure (co-NP for proper domains). If P is secure and $cost_{\mathcal{P}}(P) < currentBestCost$, then store P, and set $currentBestPlan := P$ and $currentBestCost := cost_{\mathcal{P}}(P)$.

Finally, output $currentBestPlan$.

Obviously, this computation returns an optimal secure plan, if one exists and *nil* otherwise. Note that in **A1** due to the interleaved checks the costs are unbound ($\infty = \infty - 1$) until the first secure plan has been found and they are steadily improving for further secure plans found.

Remark 4.6. *Note that in the light of the architecture of the answer set solver* DLV*, which is separated into a* model generator *and a* model checker*, the algorithm can be viewed as follows: Let Π be a program with weak constraints. For optimal answer set computation (as described in Section 2.1.5.1)* DLV*'s model generator generates one candidate model after another, similar to* ***A1*** *outlined above, and for each such candidate model* DLV*'s model checker checks whether this model is indeed an answer set, similar to* ***A2*** *above. The currently cheapest answer set is always recorded and the model generator will iteratively return "better" answer set candidates S wrt. $\mathsf{cost}_\Pi(S)$. This is where we hook in: When using $lp^w(\mathcal{P})$ as input program, this method computes optimal answer sets corresponding to optimal optimistic plans (cf. Theorem 4.8). So, for computing optimal* secure *plans, all we have to do is to*

intercept the model checker and admit only such candidate answer sets, which correspond to secure plans. This can be decided by one of the secure check methods described above in a separate call to DLV. *The overall computation perfectly matches the algorithm described in Steps* ***A0–A2*** *above. We refer to Chapter 5 for a more detailed description of the system.*

4.3.7 Checking Well-Definedness

To conclude this section, we will, show how to check the well-definedness of a planning problem $\mathcal{P} = \langle PD, Q\,?\,(l)\rangle$ by evaluating a logic program. To that end we construct such logic program Π_{WD} extending the background knowledge Π as follows. Similar to Step 7 of the translation $lp^w(\mathcal{P})$, for any action declaration in PD with a non-empty `costs`-part

$$\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_n)\ \mathtt{requires}\ B\ \mathtt{costs}\ C\ \mathtt{where}\ W.$$

we add the rules:

$$\mathtt{c_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T},C\theta)\ \mathtt{:-}\ B, W\theta, \mathtt{t}(\mathtt{T}).$$
$$\mathtt{sc_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T})\ \mathtt{:-}\ \mathtt{c_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T},\mathtt{C}).$$

to D_{WD}, where $\theta = \{\mathtt{time/T}\}$, `T` is a new variable, and `t` is a new predicate. Here, predicate $\mathtt{c_p}$ encodes some cost value of a legal action instance according to the respective declaration and the projection $\mathtt{sc_p}$ is used to derive that *some cost* value exists for a legal action instance.

Furthermore, we add the facts:

$$\mathtt{t}(1).\ \ldots\ \mathtt{t}(l).$$

For each action declaration in AD, we add three constraints using the auxiliary predicates defined before:

$$\mathtt{:-}\ B, \mathtt{t}(\mathtt{T}), \mathtt{not}\ \mathtt{sc_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T}).$$
$$\mathtt{:-}\ \mathtt{c_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T},\mathtt{C}),\ \mathtt{not}\ \#\mathtt{int}(\mathtt{C}).$$
$$\mathtt{:-}\ \mathtt{c_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T},\mathtt{C}),\ \mathtt{c_p}(\mathtt{X}_1,\ldots,\mathtt{X}_n,\mathtt{T},\mathtt{C1}),\ \mathtt{C} < \mathtt{C1}.$$

where B again is the `requires`-part of the respective declaration.

Knowing that Π has a unique answer set, the following result is obvious as the three constraints exactly check the conditions of well-definedness (i.e. that for any legal action instance costs are defined, integer and unique):

Proposition 4.23. *The logic program Π_{WD} has an answer set if and only if the planning problem $\mathcal{P}$ is well-defined wrt. action costs.*

Note that by the predicate `t(X)` we only consider a limited time range from 1 up to the plan length l of $\mathcal{P}$ in our translation in order to keep the program finite. Although this does not exactly reflect Definition 3.23 where we claimed that an action-declaration is only well-defined if costs are uniquely defined for all integers, it is sufficient for well-definedness of the costs of a plan: Concerning the unique costs of a plan for problem $\mathcal{P}$, only the observed range between 1 and l is relevant, while ambiguous costs for integers outside this range do not influence plan costs. If, on the other hand, we define the predicate `t(X)` over all

integers, this would result in a logic program over an infinite domain, which is, in general, undecidable.

In a similar way, checking well-definedness is alternatively possible via planning in the language $\mathcal{K}$: From the original $\mathcal{K}^c$ planning problem $\mathcal{P} = \langle PD, \mathtt{Q}\,?\,(\mathtt{l})\rangle$, we construct a $\mathcal{K}$ planning domain PD_{WD} and goal q, such that the planning problem $\langle PD_{WD}, q\rangle$ has a plan just if PD is well-defined.

As background knowledge, we take the background knowledge Π augmented by the facts

$\mathtt{t}(1).\ \ldots\ \mathtt{t}(l).$

for a fresh predicate `t`. Similar as above, we add for any action declaration in D fluent declarations

$\mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, C\theta)$ `requires` $B, W\theta, \mathtt{t}(\mathtt{T})$.
$\mathtt{sc_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T})$ `requires` $B, \mathtt{t}(\mathtt{T})$.

to D_{WD}, where $\theta = \{\mathtt{time}/\mathtt{T}\}$ and `T` is a new variable. Furthermore, we add a distinct fluent `wd` with an empty `requires`-part in its declaration. The `initially`-section in PD_{WD} merely contains

```
caused wd.
```

In order to check well-definedness, it is now sufficient to add the following rules in the `initially`-section of PD_{WD}:

$\mathtt{caused}\ \mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, \mathtt{C}).$
$\mathtt{caused}\ \mathtt{sc_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T})\ \mathtt{if}\ \mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, \mathtt{C}).$
$\mathtt{forbidden}\ \mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, \mathtt{C}), \mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, \mathtt{C1}),\ \mathtt{C} < \mathtt{C1}.$
$\mathtt{forbidden}\ \mathtt{not}\ \mathtt{sc_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}).$
$\mathtt{forbidden}\ \mathtt{c_p}(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{T}, \mathtt{C}),\ \mathtt{not}\ \#\mathtt{int}(\mathtt{C}).$

The `always`-section in PD_{WD} is empty. The rules are similar to those in the logic program Π_{WD}, but exploit implicit typing.

Proposition 4.24. *A $\mathcal{K}^c$ planning problem $\mathcal{P}$ is well-defined, if and only if the $\mathcal{K}$ planning problem $\mathcal{P}_{WD} = \langle PD_{WD}, \mathtt{wd}?(0)\rangle$ has an optimistic plan. (assuming that PD_{WD} has a legal initial state).*

Similar considerations upon the finiteness of the background predicate `t(X)` as above apply here, since the planning domain $\mathcal{P}_{WD}$ constructed here is a one-to-one transformation of the logic programming encoding Π_{WD} for checking well-definedness shown before.

Remark 4.7. *A well-definedness check is not yet included in the current implementation presented in Chapter 5. Strictly speaking, in the current implementation of* DLV *(or* $\mathtt{DLV}^{\mathcal{K}}$*, respectively) such a check is not feasible in exactly the form presented, since the built-in predicate* `#int` *must not be used under default negation. Nevertheless, this could be easily remedied by substituting* `#int` *in the encoding above with a new unary predicate symbol* `integer` *and adding a rule* `integer(X) :- #int(X).`

Chapter 5

The System DLV$^{\mathcal{K}}$

In this chapter, we describe the DLV$^{\mathcal{K}}$ planning system, which provides an implementation of the language $\mathcal{K}$ and its extension $\mathcal{K}^c$ as a front-end of the Answer Set Programming system DLV [LPF+02]. The implementation is based on the translations presented in Chapter 4, in principle transforming planning problems to logic programs which are then solved by calls to the underlying DLV solver.

We first describe how planning problems are specified in DLV$^{\mathcal{K}}$, followed by the usage of the system, and finally the architecture of DLV$^{\mathcal{K}}$. For exposition of the diverse features, we occasionally refer to the Bridge Crossing example already known from the previous chapters; further examples and details will be given in Chapter 6 below.

The chapter will be concluded with an outlook on further possible optimizations of the system.

5.1 Input

The DLV$^{\mathcal{K}}$ system accepts as input $\mathcal{K}^c$ planning problems specified in the enhanced syntax presented in Section 3.1.3.

A planning problem $\mathcal{P} = \langle\langle, , \langle Q, ?\rangle\rangle(i)\rangle$ has to be provided by the user in one or more input files as follows.

- The background knowledge Π, consisting of a DLP in DLV syntax is expected to be specified in one or more files $bk_1, \ldots, bk_n$ whose names do not end with suffix `.plan`. Π is then assumed to be the union of all rules in those files. Note that we do not allow full DLV syntax; special DLV features like weak constraints and aggregates (cf. Section 2.1.5.1) are prohibited in the background knowledge.

- The planning domain and goal are specified in one or more files pp_1`.plan`, ..., pp_m`.plan` whose names end with suffix `.plan` in sections of the form

  `fluents:` <list of fluent declarations>
  `actions:` <list of action declarations>
  `always:` <list of rules, macros, and executability conditions>

```
initially : <list of initial state constraints>
goal :      <goal query>
```

similar to the form defined in Section 3.1.3, with the extension that the sections may be specified in arbitrary order possibly split over several input files. Multiple sections of the same kind, where in case of multiple `goal :` sections only the last goal specified in the input is taken. Furthermore, each fluent/action has to be declared before being used in an `initially :` or `always :`) section. Apart from $\mathcal{K}^c$ declarations, rules, and macros, the user might add the command `securePlan.` anywhere in an `initially :` or `always :` section which lets DLV$^{\mathcal{K}}$ compute only secure plans.

5.2 Usage and Features

DLV$^{\mathcal{K}}$ is a command-line oriented system, which is realized as front-end to the DLV logic programming system. To start DLV$^{\mathcal{K}}$ you have to use DLV with one of the command-line options `-FP`,`-FPopt`, or `-FPsec` in order to invoke the planning front-end:

```
Usage:
DLV -FP|-FPopt|-FPsec [options] [filename [filename [...]]]
```

DLV$^{\mathcal{K}}$ knows the following special command-line options (we refer to the DLV-Manual [FP96] for other options not directly related to the planning front-end):

`-FP`
This option starts DLV$^{\mathcal{K}}$ in the standard interactive mode where each optimistic plan is displayed and the user can choose per plan whether she wants to check plan security interactively.

`-FPopt`
This option lets DLV$^{\mathcal{K}}$ only compute optimistic plans without asking whether a secure check should be performed for each plan. Note that this command-line option overrides `securePlan.` in the input.

`-FPsec`
This option lets DLV$^{\mathcal{K}}$ compute only secure plans without asking whether a secure check should be performed for each plan. This option is equivalent to the `securePlan.` command in the input. By default, check $\mathcal{SC}_1$ is performed.

`-FPcheck=`n
This option lets the user chose which method of secure checking is used in combination with options `-FP` or `-FPsec`. Currently, we have implemented secure checks $\mathcal{SC}_1$, and $\mathcal{SC}_2$ as described in Sections 4.3.4.1 and 4.3.4.2 invoked by $n = 1$, resp. $n = 2$. By default, check $\mathcal{SC}_1$ is performed.

`-FPsoundcheck=`n`, -FPcompletecheck=`m
The user can even refine this by choosing separate check methods which she knows to be sound, resp. complete, for the problem at hand. These options only take effect any front-end option `-FP`, `-FPopt` or `-FPsec` has been chosen. In this case, DLV$^{\mathcal{K}}$ does displays possibly secure plans as follows:

`PLAN (secure):` *plan* in case the sound check method succeeds.

`PLAN (security unknown):` *plan* in case the sound check fails but the complete check succeeds.

`PLAN`: for plans for which neither check succeeds (in case of front-end option `-FPsec` these plans are omitted).

Whenever only one of `-FPsoundcheck=`n `-FPcompletecheck=`m has been specified by the user, $\mathcal{SC}_1$ is taken by default for the other. These options are rather experimental in the sense that at the current state they might not often seem significant, but might be informative in presence of more than the two checks implemented so far. Similarly, other checks which do not only check plan security but also other properties could be included in future versions and checked this way. We remark that we could add arbitrary such checks within our framework.

`-planlength=`n
This option lets the user set the plan length and overrides the plan length specified in the goal query in the input. DLV$^{\mathcal{K}}$ computes plans of fixed plan length but using this option one can easily write a script which incrementally computes the shortest plan for a planning problem.

`-planminactions=`n`, -planmaxactions=`m
These two options can be used to specify the minimum and maximum number of actions per time in the plans computed by DLV$^{\mathcal{K}}$ (cf. Section 4.3.1.2 for details). Default for n is 0 and m is unbound by default. If set to 0 and 1, respectively this is semantically equivalent to macro `noConcurrency`. Note that either of these options overrides `noConcurrency`. in the input.

`-plancache=`n
In order avoid checking the same plan for security several times, DLV$^{\mathcal{K}}$ keeps a fixed number of n plans and the results of secure checks for these plans cached (see below). Cache size currently defaults to 10,000 plans.

`-costbound=`c
By default, in presence of action costs DLV$^{\mathcal{K}}$ computes optimal plans, whereas this option lets DLV$^{\mathcal{K}}$ compute admissible plans wrt. cost c.

`-n=`n
This option limits the number of plans computed by DLV$^{\mathcal{K}}$. In order to compute only one plan, set $n = 1$. By default, DLV$^{\mathcal{K}}$ prints all possible plans.

`-N=`N
This option sets an upper bound of N for the builtin integer arithmetics in DLV, DLV$^{\mathcal{K}}$. It limits the integers which may be used in a program wrt. to its ground instantiation; the built-in predicate `#int` is true for all integers $0 \ldots N$. Setting N high enough, taking into account the outcome of built-in arithmetic predicates $A = B + C$ and $A = B * C$, is important to get correct results. In general, it is not sufficient to set N simply to the maximum integer occurring in the background knowledge in our system since the pre-defined arithmetic predicates "$a = b + c$" and "$a = b * c$" might produce larger results. You have to

keep in mind to set the upper bound higher than the maximum expected integers occurring in the program with respect to these pre-defined predicates (cf. DLV-Manual [FP96] for details of the built-in integer arithmetics). Examples will be provided in Chapter 6.

For e detailed description of warnings and error messages, we refer to Appendix B.

5.3 Solving the Bridge Crossing Example in DLV$^{\mathcal{K}}$

Assume the program from Figure 3.1 given in a file named `crossing.plan` and background knowledge Π_{Bridge} in a file `crossing.bk`.

A sample run of DLV$^{\mathcal{K}}$ looks as follows:

```
$ DLV -FP crossing.plan crossing.bk

STATE 0: at(joe,here), at(jack,here), at(william,here), at(averell,here),
         hasLamp(joe)
ACTIONS: crossTogether(joe,jack):2
STATE 1: -at(joe,here), -at(jack,here), at(william,here), at(averell,here),
         hasLamp(joe), at(joe,across), at(jack,across)
ACTIONS: cross(joe):1
STATE 2: at(joe,here), at(william,here), at(averell,here), hasLamp(joe),
        -at(joe,across), at(jack,across)
ACTIONS: takeLamp(averell)
STATE 3: at(joe,here), at(william,here), at(averell,here), -hasLamp(joe),
         hasLamp(averell), at(jack,across)
ACTIONS: crossTogether(william,averell):10
STATE 4: at(joe,here), -at(william,here), -at(averell,here), hasLamp(averell),
         at(jack,across), at(william,across), at(averell,across)
ACTIONS: takeLamp(jack)
STATE 5: at(joe,here), hasLamp(jack), -hasLamp(averell), at(jack,across),
         at(william,across), at(averell,across)
ACTIONS: cross(jack):2
STATE 6: at(joe,here), at(jack,here), hasLamp(jack), at(william,across),
         at(averell,across), -at(jack,across)
ACTIONS: crossTogether(joe,jack):2
STATE 7: -at(joe,here), -at(jack,here), at(joe,across), hasLamp(jack),
         at(jack,across), at(william,across), at(averell,across)
PLAN: crossTogether(joe,jack):2; cross(joe):1; takeLamp(averell);
      crossTogether(william,averell):10; takeLamp(jack);
      cross(jack):2; crossTogether(joe,jack):2  COST: 17

Check whether that plan is secure (y/n)? y
The plan is secure.

Search for other plans (y/n)? n
```

In interactive mode (`-FP`) as chosen in this example (similarly, in optimistic mode, `-FPopt`) for each plan found the whole corresponding trajectory is displayed, i.e., the states (`STATE`) and the action sets (`ACTIONS`) performed at each state. Finally, the whole plan (`PLAN`) is displayed again, where parallel actions are separated by ',' and action sets are

separated by ';'. Whenever an action produces some costs, they are displayed after the resp. action separated by a colon, and the summed up overall plan cost (COST) is printed after the plan. For each plan found, the user can decide whether she wants to perform a secure check, and whether she wants to proceed searching for more plans.

5.4 Architecture

The architecture of the DLV$^{\mathcal{K}}$ system is outlined in Figure 5.1. It accepts files containing DLV$^{\mathcal{K}}$ input and background knowledge stored as plain Datalog files. Then, by means of suitable transformations from $\mathcal{K}^c$ to disjunctive logic programming as described in the previous chapter, it uses the classic DLV core to solve the corresponding planning problem.

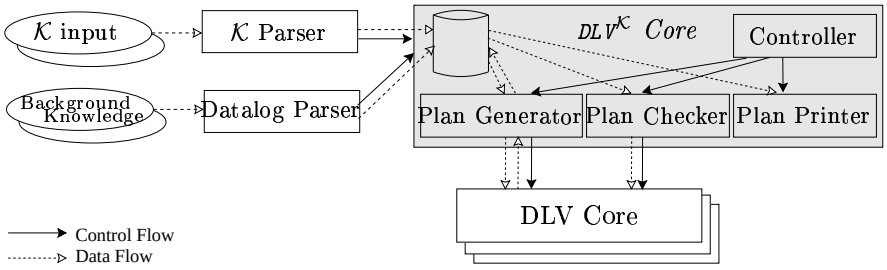

Figure 5.1: DLV$^{\mathcal{K}}$ System Architecture

DLV$^{\mathcal{K}}$ comes with two parsers for the respective different input files: The first accepts DLV$^{\mathcal{K}}$ files, that is, files with a filename extension of `.plan` that constitute a DLV$^{\mathcal{K}}$ program, while the second parser accepts optional background knowledge files as defined above. Both parsers are able to read their input from an arbitrary number of files, and both convert this input to an internal representation and store it in a common database.

The actual DLV$^{\mathcal{K}}$ front-end consists of four main modules, the Controller, the Plan Generator, the Plan Checker, and the Plan Printer. The *Controller* manages the other three modules; it performs user interactions (where appropriate), and controls the execution of the entire front-end.

To that end, the Controller first invokes the *Plan Generator*, which translates the planning problem at hand into a suitable program in the core language of DLV (disjunctive logic programming, eventually including weak constraints as described in Section 2.1) according to the transformation $lp^w(\mathcal{P})$ provided in Section 4.3.2.

The Controller then invokes the DLV kernel to solve the corresponding problem. In case that optimistic planning is not explicitly enforced by option `-FPopt` after grounding , before starting the actual answer set computation, the Controller interrupts DLV and performs a check for mux-stratification, wrt. opposite literals in the `after` parts of causation rules on the ground instantiation of the program, considering all rules emerging from the translation of causation rules. A warning is printed, if this mux-stratification check fails.

Then, the DLV model generator is invoked in order to produce answer sets. The resulting

answer sets (if any) are fed back to the Controller, which extracts the solutions to the original planning problem from these answer sets, transforms them back to the original planning domain, and saves them into the common database.

The Controller then optionally (if the user specified the `securePlan.` command, option `-FPsec`, or invoked a secure check interactively) invokes the *Plan Checker*. Similarly to the Plan Generator, the Checker uses the original problem description together with the optimistic plan computed by the Generator to generate a disjunctive logic program that solves the problem of verifying whether this (optimistic) plan is in fact also a secure plan, depending on which check method has been chosen. Currently implemented secure checks are SC_1 and SC_2 as introduced in Sections 4.3.4.1 and 4.3.4.2. Here, in order avoid checking the same plan for security several times the checker keeps a fixed number of plans and the results of secure checks cached. Since the number of plans might be exponential, we have chosen a fixed cache size which adds/deletes plans with a first-in-first-out strategy.

In normal (non-optimal) planning, the checker is simply invoked for each answer set returned right before transforming it back to user output. In the case of optimal secure planning, on the other hand, the candidate answer set generation of the DLV kernel has to be "intercepted": The kernel proceeds computing candidate answer sets, returning an answer set with minimal sum of violation costs wrt. the weak constraints from translation $lp^w(\mathcal{P})$, by running through all candidates. Here, in order to generate *optimal secure plans*, the planning front-end interrupts computation, allowing only answer sets which represent secure plans to be considered as candidates by means of interleaved calls to the Plan Checker. The computation of candidate answer sets in the DLV kernel which is paused during this plan check, is resumed after the check. In fact, the system implements exactly the algorithm outlined in Section 4.3.6 for computing the first optimal plan and starts a second run where it only computes admissible plans wrt. the optimal costs found in the first run in order to compute further optimal plans.

Finally, the *Plan Printer*, translates the solutions found by the Generator (and optionally verified by the Checker) back into suitable output for the planning user and prints it in the format shown above.

Chapter 6

Knowledge Representation in $\mathcal{K}^c$

In this chapter, we will show how standard planning problems, as well as interesting elaborations beyond the classical formulations of these problems, can be encoded in $\mathcal{K}^c$ and solved by the DLV$^{\mathcal{K}}$ planning system.

We start with a short discussion of the Crossing the Bridge example which has been used for exemplification throughout Chapter 3, recalling some the presented features again.

Thereafter, we will continue with a classical problem from the Blocks World, where we will discuss basic principles of knowledge representation in $\mathcal{K}^c$, and which we will reuse later on for further elaborations.

Next, we will provide several encodings of problems with incomplete information and nondeterministic actions. Among those, we will discuss several elaborations of the famous "Bomb in the Toilet"problem, which, as we will see, in fact can be formalized as a deterministic planning problem under the inherent knowledge state view of language $\mathcal{K}$. Furthermore, we will discuss another well-known problem on self-location of a robot in a grid with unknown initial location.

We also provide two novel scenarios. The first example which models a simple counter shall illustrate reasoning in belief states with $\mathcal{K}$. The second example about painting a house concentrates on modeling disjunctive knowledge and the applicability of our knowledge state view to different domains.

A further block is dedicated to optimal planning, where we will make use of the extension of our basic language $\mathcal{K}$ by action costs. We will discuss the combination of different optimization criteria in a simple Blocks World domain on the basis of respective encodings in $\mathcal{K}^c$. Furthermore, we will investigate elaborations of another optimization problem, Traveling Salesperson, where we add exceptional time-dependent costs. At the end of this block we will discuss resource based planning in our approach with a simple example on admissible planning.

Finally, we will discuss general design principles for $\mathcal{K}$ and $\mathcal{K}^c$, further explaining our knowledge state view, which should enable the reader to encode arbitrary domains in our language.

We have chosen a description of the relevant concepts by example as we believe that these examples give the reader a general idea of the relevant modeling issues in our language and

serve to transfer the concept into modeling new domains.

6.1 Crossing the Bridge

Let us first recall the features we have already seen in the elaborations of Problem 3.1 about Bridge Crossing. Figure 3.1 (p. 38) shows an encoding of the basic Bridge Crossing problem $\mathcal{P}_{BCP}$ in $\mathcal{K}$.

First of all, by use of background knowledge Π, $\mathcal{K}$ allows for a very flexible way of *typing*. While we have used this only for simple factual knowledge about persons, and the different sides of the river, or walking times in the Bridge Crossing example, the background knowledge allows much more flexibility in general. The only restriction we have to obey, is that Π needs to have a unique answer set, but apart from that it allows the full flexibility of logic programming. We will see some example, where exceptions are encoded in the background knowledge in Section 6.5.

As for modeling action qualifications in $\mathcal{K}$, we have seen in $\mathcal{P}_{BCP}$ (see Figure 3.1 on p. 38) that the user might specify several *alternative preconditions* by multiple positive executability conditions and/or negative executability conditions for actions. For instance, crossing in two is possible, whenever either of the two involved persons has the lamp. Furthermore, actions might also depend on each other wrt. to executability. Apart from defining only *sequential* action execution by the macro `noConcurrency`, we may define arbitrary constraints describing which actions might occur in *parallel*. As a short example we will model parallel moves of Blocks in the subsequent section.

As for action effects, $\mathcal{K}$ allows for *conditional effects* as well as static effects (ramifications). An example for the former is the effect of the action `cross(X)`, which depends on where person `X` has been in the previous state. We will see several more examples in the following. Also here, $\mathcal{K}$ allows for full flexibility of logic programming, allowing to model nondeterminism by negation as failure, adding state constraints, etc. We have seen some example of modeling incomplete knowledge about the initial state and nondeterministic action effects in another elaboration of the Bridge Crossing example, namely $\mathcal{P}_{BCPsec}$ (see p. 41). Furthermore, negation as failure was used in an elegant formulation of *frame axioms* by macro `inertial`. We will further exploit these features in the subsequent examples.

Next, we have considered an elaboration of the Bridge Crossing example where action costs where used to obtain optimal plans. This can also be combined with *incomplete information* and *nondeterministic action effects*, which we have elaborated in the examples $\mathcal{P}_{QBCP}$ and $\mathcal{P}_{QBCPsec}$ (see p. 44). We will see further examples how action costs can be used for various optimization tasks and also examples modeling time-dependent, dynamic costs of actions.

The goal of the remainder of this chapter is to further explain and elaborate the use of these features for knowledge representation in $\mathcal{K}$ and $\mathcal{K}^c$.

6.2 A Classical Problem – Blocks World

Let us now have a closer look at one of the most well-known scenarios when introducing the concepts of AI Planning, namely the Blocks World. Here, the goal is to build some stacks

of blocks which are located on a table. Figure 6.2 shows a simple instance. The planning problem consists of an initial configuration of blocks and a (probably partly specified) goal configuration. The only action allowed is *moving* a block x to a location l, i.e. onto the table or on top of another block which is not occupied. We refer to Blocks World with parallel moves allowed, where minimizing the total number of moves is an issue. A $\mathcal{K}^c$ encoding for this domain, where plans are serializable, is shown in Figure 6.1. Serializability here means that parallel actions are non-interfering and can also be executed sequentially in any order, i.e. each parallel plan can be arbitrarily "unfolded" to a sequential plan.

```
fluents:   on(B,L) requires block(B), location(L).
           blocked(B) requires block(B).
           moved(B) requires block(B).

actions:   move(B,L) requires block(B), location(L) costs 1.

always:    executable move(B,L) if B != L.
           nonexecutable move(B,L) if blocked(B).
           nonexecutable move(B,L) if blocked(L).
           nonexecutable move(B,B1) if move(B1,L).
           nonexecutable move(B,L) if move(B1,L), B < B1, block(L).
           nonexecutable move(B,L) if move(B,L1), L < L1.

           caused on(B,L) after move(B,L).
           caused blocked(B) if on(B1,B).
           caused moved(B) after move(B,L).
           caused on(B,L) if not moved(B) after on(B,L).
```

Figure 6.1: $\mathcal{K}^c$ encoding for the Blocks World domain

A concrete instance is depicted in Figure 6.2. The planning problem emerging from the initial state and the goal state in Figure 6.2 can be modeled using the background knowledge Π_{bw}:

```
block(1). block(2). block(3). block(4). block(5). block(6).
location(table).
location(B) :- block(B).
```

and extending the program in Figure 6.1 as follows:

```
initially: on(1,2). on(2,table). on(3,4). on(4,table).
```

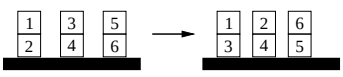

Figure 6.2: A simple Blocks World instance

```
                on(5,6). on(6,table).
goal :          on(1,3), on(3,table), on(2,4), on(4,table),
                on(6,5), on(5,table) ? (l)
```

Before discussing the solutions of this instance, let us take a closer look on this $\mathcal{K}^c$ formulation of the Blocks World problem: As opposed to the formulations discussed in the Preliminaries we have chosen a formalization with three fluents `on`, `blocked` and `moved`. Furthermore, we have one action `move` describing the usual movement of blocks to locations. Each of these moves costs 1, i.e. any optimal plan minimizes the total number of moves. In the `always` section, serializable, parallel moves are described by `executable` and `nonexecutable` statements restricting moves such that:

- a block can (by default) be moved to any location except onto itself,
- occupied blocks can not be moved,
- a block can not be moved to an occupied block,
- a block can not be moved on top of another block which is moved at the same time.
- two different[28] blocks can not be moved to the same block at once, and
- a block can not be moved to two divergent[28] locations at once.

Only positive effects are described in this encoding, which we will explain in further detail below. Being blocked is an indirect/static effect here. A succinct description of the only relevant frame axiom in this domain stating that an unaffected block remains at its location is also described as an "indirect effect" of not being moved, where we use default negation. The other effects are direct consequences of the `move` action.

The fluent `on(B,L)` expresses the location of a block in the usual way, i.e. that a certain block `B` resides at location `L`.

In analogy to the fluent `clear` often used in other formulations of this problem (cf. Section 2.4), fluent `blocked(B)` expresses that a block is occupied by another block at the current state. One might argue, that this is just the opposite view and we could similarly use `clear` instead. As we will see, a reasonable choice for formulating fluents positively or negatively is essential in $\mathcal{K}^c$ in order to exploit the inherent knowledge state view: We only want to express what we know in each state, as opposed to complete truth assignments for all fluents (i.e., world states). In fact, by design choices, modeling the domain the above $\mathcal{K}$ program does not contain any occurrence of true negation. We do not need the usual `inertial` macro (which involves true negation) here in favor of the final frame axiom. The additional fluent `moved` indicates which blocks have actually been affected by moves per time step. For moved blocks, information about their former location is simply "forgotten" since the frame rule does not apply to carry this information over to the next state. We see that, besides the standard `inertial` macro, $\mathcal{K}$ allows for different formulations of frame axioms as well.

[28] Note that we used built-in "<" in the final two `nonexecutable` rules in order to express inequality. In these two cases "<" is semantically equivalent with "!=", but avoids symmetric ground instantiations.

Obviously, any state reachable by a legal transition in the domain PD_{bw} given by the program in Figure 6.1 does only consist of positive fluents $\texttt{on}(\texttt{B},\texttt{L})$ and $\texttt{blocked}(\texttt{L})$ describing a "relevant clipping" of knowledge. We do not care which blocks are unblocked or wherever a block is *not* located currently. In $\mathcal{K}$, we do not have to completely specify truth values for all fluents but only describe what is necessary.

Each move is penalized with cost 1, which results in a minimization of the total number of moves.

Let $\mathcal{P}(l)$ denote the planning problem for plan length l. For $l = 2$, we have an optimal plan which involves six moves, i.e. $cost^*_{\mathcal{P}(2)} = 6$:

$$P_2 = \langle\ \{\texttt{move}(1,\texttt{table}), \texttt{move}(3,\texttt{table}), \texttt{move}(5,\texttt{table})\},\ \ \{\texttt{move}(1,3), \texttt{move}(2,4), \texttt{move}(6,5)\}\ \rangle$$

By unfolding the steps, this plan gives rise to similar plans of length $l = 3, \ldots, 6$ that have cost 6. For $l = 3$, we can find the following optimal plan, which has cost 5:

$$P_3 = \langle\ \{\texttt{move}(3,\texttt{table})\},\ \ \{\texttt{move}(1,3), \texttt{move}(5,\texttt{table})\},\ \ \{\texttt{move}(2,4), \texttt{move}(6,5)\}\ \rangle$$

This plan can not be further parallelized to having only two steps. For any plan length $l > 3$, we will obtain optimal plans similar to P_3, extended by void steps. So, the cheapest plans wrt. the total number of moves all have cost 5 and need at least three steps. Note that shortest parallel plans (of length 2) are more expensive, as explained above.

Thus, when minimizing the number of moves is our goal, the shortest plan is not necessarily the optimal one. This motivates a more detailed discussion about different, possibly conflicting optimization criteria in $\mathcal{K}^c$ which we will provide in Section 6.4.

6.3 Planning with Incomplete Knowledge

Let us now focus on domains comprising incomplete knowledge about the initial state and nondeterministic action effects. We have already seen representation techniques for such scenarios in the example $\mathcal{P}_{BCPsec}$ from Section 3.1.4. We will now discuss some $\mathcal{K}^c$ encodings of standard conformant planning problems. As we will show, the language $\mathcal{K}$ is capable of expressing classical encodings based on states of the world. However, by its design it is very well-suited for encodings based on states of knowledge. We will see how particular conformant planning problems can profit from knowledge state encodings as opposed to world state encodings. We show both types of encodings on some "Bomb in the Toilet" planning problems, and discuss the two different approaches, highlighting some computational advantages of the encodings based on states of knowledge. On the other hand, we will also show the limitations of the knowledge state view by examples where encoding all contingencies can not be avoided.

6.3.1 Bomb in the Toilet - World State Encodings

We will first turn our attention to the "Bomb in the Toilet" problem [McD87] and its variations. We will describe these domains gradually, starting with two versions which involve deterministic action effects and incomplete initial state specifications. Only after these, variants comprising nondeterministic action effects and some additional elaborations are presented. We employ a naming convention which is due to [CR00].

BT(p) - Bomb in Toilet with p packages We have been alarmed that there is a bomb (exactly one) in a lavatory. There are p suspicious packages which could contain the bomb. There is one toilet bowl, and it is possible to dunk a package into it. If the dunked package contained the bomb, the bomb is disarmed.

For the $\mathcal{K}$ encoding, the background knowledge Π_{bt} consists of a definition of the packages:

```
package(1).  package(2).  ...  package(p).
```

Using the DLV$^{\mathcal{K}}$ builtin predicate #int we can equivalently define this background knowledge such that we can set the number of packages by command-line option -N=p:

```
package(P) :- #int(P), P > 0.
```

We use two fluents: armed(P) holds if package P contains an armed bomb (this is an inertial property), and unsafe expresses the fact that there are armed bombs. Only one action, dunk(P), is required, which is always executable and the effect of which is that package P is no longer armed.

For the initial state, total armed(P). expresses the fact that the armed bomb might be in any package P, while forbidden armed(P), armed(P1), P != P1. enforces that at most one package can contain an armed bomb. The statement forbidden not unsafe. is included to guarantee that at least one package contains an armed bomb in the initial state.

The goal is to achieve a state in which no armed bomb exists, i.e., which is not unsafe. This goal q_{bomb} will be the same for all following variations of the "Bomb in the Toilet" problems, the respective plan lengths j will be stated for each problem. We thus arrive at the following planning problem $\mathcal{P}_{bt} = \langle PD_{bt}, q_{bomb} \rangle$:

```
fluents:    armed(P) requires package(P).
            unsafe.
actions:    dunk(P) requires package(P).
always:     inertial armed(P).
            caused -armed(P) after dunk(P).
            caused unsafe if armed(P).
            executable dunk(P).
initially:  total armed(P).
            forbidden armed(P), armed(P1), P != P1.
            forbidden not unsafe.
goal:       not unsafe ? (j)
```

Note that in the formulation of this simple domain there is only one deterministic action, while the initial state is incomplete since it is not known which of the p packages contains the bomb. In this encoding we adopt a world state view for the initial state, since any legal initial state contains complete information on the fluent armed for each package. Note that, strictly speaking, this is not the case for the successor states, where we keep only positive information on armed by statement inertial armed(P). In principle this is a world state view with the closed world assumption (CWA) applied to fluent armed, i.e. as soon as we do not

know that a package is armed, we assume it not to be. True negation via the initial `total` statement and the effect of dunking (`caused -armed(P) after dunk(P).`) is only needed to override positive inertia. A strict world state view with explicit representation of negative information in any state can be achieved by adding `inertial -armed(P).` as well.

Usually, a plan should be produced which establishes the goal no matter in which package the bomb is in, so we look for a secure plan. If concurrent actions are allowed, the following secure plan for $j = 1$ (dunking all packages at the same time) can be found:

$P = \langle$ {`dunk(1)`, ..., `dunk(`p`)`} $\rangle$

Secure sequential plans (achieved by adding macro `noConcurrency.` in the program above or by setting command-line option `-planmaxactions=1` consists of dunking all packages sequentially, so $j = p$:

$P = \langle$ {`dunk(1)`}, ..., {`dunk(`p`)`} $\rangle$

Any permutation of these action sets is also a valid secure plan.

Given Π_{bt} in file `bt.bk` and the program above in file `bt.plan` we can compute these (sequential) plans for BT(p) using $\texttt{DLV}^{\mathcal{K}}$ by :

```
$ DLV -FPsec -planmaxactions=1 bt.plan bt.bk -planlength=p -N=p
```

BTC(p) - Bomb in Toilet with certain clogging Let us now consider a slightly more elaborate version of the problem: Assume that dunking a package clogs the toilet, making further dunking impossible. The toilet can be unclogged by flushing it. The toilet is assumed to be unclogged initially. Note that this domain still comprises only deterministic action effects.

We extend $PD_{bt} = \langle \Pi_{bt}, \langle D_{bt}, R_{bt} \rangle \rangle$ to $PD_{btc} = \langle \Pi_{bt}, \langle D_{btc}, R_{btc} \rangle \rangle$ by adding a new fluent, `clogged`, and a new action, `flush`, to D_{btc}:

```
fluents:   clogged.
actions:   flush.
```

`clogged` is inertial, is a deterministic effect of `dunk`, and is terminated by `flush`. `flush` is always executable, so the following rules are added to $C_{R_{btc}}$:

```
always:    inertial clogged.
           caused -clogged after flush.
           caused clogged after dunk(P).
           executable flush.
```

The executability statement for `dunk` has to be modified, as `dunk` is not executable if the toilet is clogged.

```
           executable dunk(P) if not clogged.
```

Since `clogged` is assumed not to hold initially, and since it is interpreted under the CWA, nothing has to be added to $I_{R_{btc}}$.

For the planning problem $\mathcal{P}_{btc} = \langle PD_{btc}, q_{bomb} \rangle$ we are only interested in sequential plans, as dunking and flushing concurrently is not permitted. A minimal secure plan can be found for $j = 2p - 1$:

$P = \langle$ {dunk(1)}, {flush}, {dunk(2)}, ..., {flush}, {dunk(p)}$\rangle$

Again, the action sets containing dunk actions can be arbitrarily permuted, as long as the flush actions are executed between the dunk actions.

BTUC(p) - Bomb in Toilet with uncertain clogging Consider a further elaboration of the domain, in which clogged may or may not be an effect of dunking. In other words, the action dunk has a nondeterministic effect, and the toilet is clogged or not clogged after having executed dunk.

This behavior is modeled by declaring clogged to be total after dunk has occurred. Therefore the action effect

```
caused clogged after dunk(P).
```

in PD_{btc} is modified to

```
total clogged after dunk(P).
```

yielding the planning domain PD_{btuc}. The planning problem $\mathcal{P}_{btuc} = \langle PD_{btuc}, q_{bomb} \rangle$ admits the same secure plans as $\mathcal{P}_{btc}$.

BMTC(p,t), BMTUC(p,t) - Bomb in Toilet with multiple toilets Yet another elaboration is to assume that several toilet bowls (t, rather than just one) are available in the lavatory. The modifications to PD_{btc} yielding $PD_{bmtc} = \langle \Pi_{bmt}, \langle D_{bmtc}, R_{bmtc} \rangle \rangle$ and to PD_{btuc} yielding $PD_{bmtuc} = \langle \Pi_{bmt}, \langle D_{bmtuc}, R_{bmtuc} \rangle \rangle$ are rather straightforward.

The background knowledge Π_{bt} is simply extended to contain also a definition of the t toilets, by adding:

```
toilet(1).  toilet(2).  ...  toilet(t).
```

arriving at Π_{bmt}. The fluent and action declarations for clogged, dunk, and flush must be parameterized wrt. the affected toilet. The updated definitions wrt. D_{btc} (resp. D_{btuc}) are as follows:

```
clogged(T) requires toilet(T).
dunk(P, T) requires package(P), toilet(T).
flush(T) requires toilet(T).
```

Furthermore, each occurrence of clogged, dunk, and flush in R_{btc} (resp. R_{btuc}) must be updated by adding a variable T (representing the toilet) to its parameters.

Since multiple resources can be used concurrently here, we add some concurrency conditions for the actions to PD_{btc} (resp. PD_{btuc}): dunk and flush should never be executed concurrently on any toilet. Furthermore, at most one package should be dunked into a toilet, and any package should be dunked into at most one toilet at a time. These conditions are captured by the following rules:

```
always:    nonexecutable dunk(P,T) if flush(T).
           nonexecutable dunk(P,T) if dunk(P1,T), P!=P1.
           nonexecutable dunk(P,T) if dunk(P,T1), T!=T1.
```

In total, $\langle D_{bmtuc}, R_{bmtuc} \rangle$ of PD_{bmtuc} looks as follows:

```
fluents:   clogged(T) requires toilet(T).
           armed(P) requires package(P).
           unsafe.
actions:   dunk(P,T) requires package(P), toilet(T).
           flush(T) requires toilet(T).
always:    inertial armed(P).
           inertial clogged(T).
           caused -clogged(T) after flush(T).
           caused -armed(P) after dunk(P,T).
           total clogged(T) after dunk(P,T).
           caused unsafe if armed(P).
           executable flush(T).
           executable dunk(P,T) if not clogged(T).
           nonexecutable dunk(P,T) if flush(T).
           nonexecutable dunk(P,T) if dunk(P1,T), P!=P1.
           nonexecutable dunk(P,T) if dunk(P,T1), T!=T1.
initially: total armed(P).
           forbidden armed(P), armed(P1), P!=P1.
           forbidden not unsafe.
```

The secure plans for $\mathcal{P}_{bmtc} = \langle PD_{bmtc}, q_{bomb} \rangle$ and $\mathcal{P}_{bmtuc} = \langle PD_{bmtuc}, q_{bomb} \rangle$ are similar to those for $\mathcal{P}_{btc}$ and $\mathcal{P}_{btuc}$, respectively. The differences are that up to t `dunk` and `flush` actions, respectively, can be executed in parallel (so the plans are no longer sequential), and that $t-1$ flushing actions can be saved since no final flushing is required for any toilet. Therefore any secure plan consists of $2p-t$ actions and in q_{bomb}, the minimal plan length is: $j = 2\lceil \frac{p}{t} \rceil - 1$. We remark that the `nonexecutable` conditions concerning concurrent actions again guarantee serializability in the sense discussed in Section 6.2 above, i.e. parallel dunks or flushes on different toilet bowls can be sequentially applied in arbitrary order.

The world-state encodings of BT(p), BTC(p), and BMTC(p,t) are stratified, so the security check $\mathcal{SC}_1$ is guaranteed to be sound and complete for these problems by Corollary 4.16. In the case of BTUC(p) and BMTUC(p,t) in the world-state programs, the macro `total` violates stratification. However, both BTUC(p) and BMTUC(p,t) are `false`-committed domains, and thus the security check $\mathcal{SC}_1$ is sound and complete for these problems by Theorem 4.15 as well. Indeed, the respective programs have no cycle with an odd number of negative arcs in their dependency graphs (see the BMTUC(p,t) encoding above; BTUC(p,t) and BMTUC(p,t) have the same dependency graph, since the only difference is that some fluents get an additional argument), so by well-known results at least one answer set is guaranteed, and thus condition (ii) of `false`-committedness holds. Furthermore, the only constraints are those resulting from expanding the `nonexecutable` statements. Since these

constraints refer only to actions, either all s' in a transition $\langle s, A, s' \rangle$ satisfy them or no s' does. Therefore, condition (i) of `false`-committedness is enforced as well.

6.3.2 Bomb in the Toilet - Knowledge State Encodings

In this section, alternative planning domains for "Bomb in the Toilet" will be presented. These encodings will be based on states of knowledge, a distinguishing feature of $\mathcal{K}$, rather than states of the world as in the previous sections. We will use the same background knowledge Π_{bt} (resp. Π_{bmt}) and the same goal q_{bomb} with the same values for the plan length j as in Section 6.3.1.

BT(p) In Section 6.3.1 we have represented the initial situation by means of totalization on `armed(P)`, leading to multiple initial states, corresponding to different possible states of the world. From the knowledge perspective, nothing definite is known about `armed(P)` (and about `−armed(P)`) for a particular package `P`, so the initial situation can be represented by one state in which neither `armed(P)` nor `−armed(P)` holds. The action `dunk(P)` has the effect that `−armed(P)` is known to hold, and `−armed(P)` is inertial. We state the planning domain PD_{btk} as follows:

```
fluents:   armed(P) requires package(P).
           unsafe.
actions:   dunk(P) requires package(P).
always:    inertial -armed(P).
           caused -armed(P) after dunk(P).
           caused unsafe if not -armed(P).
           executable dunk(P).
```

The advantage of this encoding is that multiple initial states do not have to be dealt with. Note that in this formulation, it is not of much help to encode in addition the restriction that exactly one package is armed: Nothing is known about the armed status of any individual package whatsoever, and any of the packages could be the armed package. Without sensing, or other appropriate determining actions, we can not detect it, and thus we can not fruitfully make use of definite knowledge `armed(P)` or `−armed(P)`. Furthermore, since the domain is deterministic, optimistic and secure plans coincide.

BTC(p) PD_{btck} is extended from PD_{btk} in the same way as PD_{btc} was obtained from PD_{bt} in Section 6.3.1, i.e., by adding declarations for `clogged` and `flush`, adding rules for action effects wrt. `clogged`, defining `clogged` to be inertial, stating `flush` to be always executable, and by modifying the executability condition for `dunk(P)`.

Note that in this encoding `clogged` is still interpreted under the CWA.

BTUC(p) In the variant with uncertain clogging, the effect of `dunk(P)` is that the truth of `clogged` is unknown. $\mathcal{K}$ has the capability of representing a state in which neither `clogged` nor `−clogged` holds, but to do this, we should no longer interpret `clogged` under the CWA, as we would not like to assume that `clogged` does not hold if it is unknown. For this reason

`inertial − clogged.` is included, and for the initial state, it must be stated explicitly that the toilet is unclogged.

Unfortunately, there is no construct in $\mathcal{K}$, with which an action effect of some fluent being unknown can be expressed directly. However, it is possible to modify the inertial rules for `clogged` and `−clogged`, so that inertia applies only if no package has been dunked. That means that dunking stops inertia for `clogged`, and `clogged` will be unknown unless it becomes known otherwise. Since this technique encodes inertia under some conditions, we call it *conditional inertia*.

To achieve this, a new fluent `dunked` is introduced, which holds immediately after `dunk(P)` occurred for some package `P`. The `inertial` macros are then extended by the additional condition. The precise meaning of the resulting program is that neither `clogged` nor `−clogged` will hold after `dunk(P)` has been executed for some package `P`, unless one of them is caused by some other rule different from inertia.

To summarize, the following is added to PD_{btck}:

```
fluents:   dunked.
always:    inertial clogged if not dunked.
           inertial -clogged if not dunked.
           caused dunked after dunk(P).
           caused -clogged after flush.
           executable dunk(P) if -clogged.
initially: -clogged.
```

while a few statements are dropped:

```
always:    inertial clogged.
           caused clogged after dunk(P).
           executable dunk(P) if not clogged.
```

yielding PD_{btuck}.

Note that also PD_{btuck} is deterministic and has a unique initial state, so optimistic and secure plans coincide. These examples shows that it is possible to find an encoding which is substantially less complex to solve by using techniques, which exploit the "state of knowledge" paradigm of the language $\mathcal{K}$. We would like to point out that this is not a contradiction to complexity results in Section 3.3 below (finding secure plans is more complex than finding optimistic plans): BTUC(p) per se is an easy problem (it is solvable in linear time), it is just the representation requiring examination of alternatives which makes it appear hard.

BMTC(p,t), BMTUC(p,t) As in Section 6.3.1, a generalization to domains involving multiple toilets is straightforward and can be achieved by applying the changes described there, resulting in the planning domains PD_{bmtck} and PD_{bmtuck}, respectively. Find PD_{bmtuck} as an example below (Π_{bmt} is omitted):

```
fluents:   clogged(T) requires toilet(T).
           armed(P) requires package(P).
```

```
            dunked(T) requires toilet(T).
            unsafe.
actions:    dunk(P,T) requires package(P), toilet(T).
            flush(T) requires toilet(T).
always:     inertial -armed(P).
            inertial clogged(T) if not dunked(T).
            inertial -clogged(T) if not dunked(T).
            caused dunked(T) after dunk(P,T).
            caused -clogged(T) after flush(T).
            caused -armed(P) after dunk(P,T).
            caused unsafe if not -armed(P).
            executable flush(T).
            executable dunk(P,T) if -clogged(T).
            nonexecutable dunk(P,T) if flush(T).
            nonexecutable dunk(P,T) if dunk(P1,T), P!=P1.
            nonexecutable dunk(P,T) if dunk(P,T1), T!=T1.
initially: -clogged(T).
```

Also in this case the resulting problem domains are deterministic and hence optimistic plans and secure plans coincide. This indicates that planning problems of this section can be solved faster than those of Section 6.3.1. Indeed, we have observed this also experimentally; as will be shown in Chapter 8, the encodings of Section 6.3.2 can often be solved several orders of magnitudes faster than those of Section 6.3.1 using the DLV$^{\mathcal{K}}$ system.

For this particular domain, the informal reason why such a concise encoding is possible is the fact, that both actions `dunk` and `flush` *gain* knowledge in some sense, i.e. we *know* that package p is unarmed after being dunked and we *know* that toilet t is unclogged after being flushed. Both actions prune the set of possible world states "deterministically" in some sense.

On the other hand, the nondeterminism arising from dunking in BTUC and BMTUC is not knowledge-gaining at all but leaves *all* possibilities on the value of `clogged` open. In this sense, by dunking we lose all knowledge about the fact whether the toilet is dunked or not and can *forget* about it in a knowledge state view. Here, one knowledge state represents several belief states. We will discuss "forgetting" and where it can be applied in more detail below in Section 6.3.5.

It has been recognized that the "Bomb in the Toilet" domain is in fact easy under our view. Also other planning systems like for instance Conformant-FF [BH03] which solve "Bomb in the Toilet" rather quickly as support the assumption that this is not a hard conformant problem. The authors of [BH03] claim that in their formalism (which, however, only allows for initial uncertainty so far in a language close to ADL) problems are generally easier to solve with their planner, if actions do not have conditional effects: this is indeed the case for the "Bomb in the Toilet" domain. The relation between these results and possible structural analysis of problems in our more general language in order to eventually optimize domains beforehand by structural analysis appears to be promising.

6.3.3 Square

In this section, we consider a conformant planning domain which has seemingly harder problems than the above-mentioned "Bomb in the Toilet" domain. It is about self-location of a robot which moves in a wall-bounded $n \times n$ grid. The robot can move in four directions (`up`, `down`, `left`, `right`) and we do not know its initial position. Moving towards a wall has no effect, i.e. the robot stays in its position. The problem of finding a conformant plan for reaching a fixed position (for simplicity, we assume reaching the corner position $(0, 0)$), is referred to as SQUARE(n) in the literature, cf. [BG00, PR95]. Extensions to 3-dimensional movement, CUBE(n) are obvious, but we only consider the two-dimensional problem here.

A $\mathcal{K}$ encoding for this problem is as follows, where Π_{square} consists solely of the built-in predicates `#int` and `#succ`.

```
fluents:   atX(P) requires #int(P).
           atY(P) requires #int(P).
           anywhere.

actions:   up. right. left. down.

always:    executable up.
           executable right.
           executable left.
           executable down.
           nonexecutable up if down.
           nonexecutable left if right.
           caused atY(Y) after atY(Y1), #succ(Y,Y1),up.
           caused atY(Y1) after atY(Y), #succ(Y,Y1),down.
           caused atX(X) after atX(X1), #succ(X,X1),left.
           caused atX(X1) after atX(X), #succ(X,X1),right.
           caused -atX(X) if atX(X1), X1 != X after atX(X).
           caused -atY(Y) if atY(Y1), Y1 != Y after atY(Y).
           inertial atX(X).
           inertial atY(Y).

initially: total atX(X). total atY(Y).
           forbidden atX(X), atX(X1), X!=X1.
           forbidden atY(Y), atY(Y1), Y!=Y1.
           caused anywhere if atX(X), atY(Y).
           forbidden not anywhere.

goal:      atX(0),atY(0)?(n)
```

The encoding is more or less self-explanatory. Fluents `atX`$(\cdot)$ and `atY`$(\cdot)$ mark the coordinates of the current position of the robot in the grid, and the conditional effects of moving in either direction are modeled by simple causation rules. We refrained from encoding the position into a single fluent `at(X,Y)`, since this would need more overhead for modeling the respective effects rules. For the initial state we use an auxiliary fluent `anywhere` which enforces that the robot has an initial location, similar to the fluent `anybodyHasLamp` from our nondeterministic Bridge Crossing example $\mathcal{P}_{BCP_{sec}}$ (see p. 41). Note that diagonal

moves are allowed, i.e. executing up/down and left/right in parallel, and only parallel occurrences of opposite moves are constrained by the two `nonexecutable` statements. With this program, we can solve SQUARE($n+1$) by calling $\texttt{DLV}^{\mathcal{K}}$ as follows:

```
$ DLV -FPsec square.plan -N=n -planlength=n -n=1
```

We only need plan length n here, since we do allow diagonal moves.

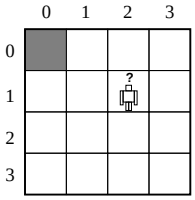

Figure 6.3: SQUARE(4)

For instance, a secure plan for SQUARE(4), as shown in Figure 6.3

$P = \langle$ {`left`, `up`}, {`left`, `up`}, {`left`, `up`}$\rangle$

is a three-step secure plan.

Note that in this domain the only source of uncertainty is the initial state. All actions are always executable and effects are deterministic. Therefore, the domain is trivially `false`-committed. However, the exponentially many initial states make this problem difficult and hard to solve. Furthermore, as opposed to the "Bomb in the Toilet" examples, the actions do not "gain" additional knowledge at once like above. Action effects are conditional, i.e. depend on the previous state, which hinders us from giving a more concise representation in terms of knowledge states here: Although we do not know the exact position in the beginning, we need to know the position at each step in order to determine the action effects, such that the best we can do is encoding all possible initial world states here. This also corresponds to the observations made by Brafman and Hoffman [BH03], who state that conditional effects complicate things in their formalism.

6.3.4 Counter Reset - Reasoning about Belief States

The following problem exemplifies the inherent complexity of secure planning in the general case in a descriptive way. As already shown in the Square problem above, it is not always possible to simply "compile down" belief states (i.e. sets of possible world states) to a single knowledge state like we demonstrated it for the knowledge state encodings of the "Bomb in the Toilet" problem. This is particularly true if the outcome of actions is conditional, i.e. depending on the previous state. In order to demonstrate how to reason on belief states in $\mathcal{K}$ in a more illustrative way, we will now show another small, but interesting secure planning problem.

Let us assume a binary counter $b_1, \ldots, b_n$ with n bits where again the initial state is unknown and our goal is to find a secure plan which brings it to a particular state, similar to the Square example above.

The counter has 2 buttons (actions):

1. **Add**: Adds one to the counter.

2. **Inverse**: If we press this button and the counter is currently in position $1, \ldots, 1$, all bits of the counter will be inverted. I.e., it resets the counter to zero, but this only works in position $1, \ldots, 1$, otherwise the counter remains unaffected.

Our goal is to bring the counter to position $1, \ldots, 1$.

Figure 6.4 shows the shortest possible secure plan for this problem with a 2-bit counter by transitions between the belief states, i.e. sets of possible world states. Each oval symbolizes a belief state, i.e. a set of knowledge states.

The following $\mathcal{K}$ program models the discussed problem $\mathcal{P}_{count}$, (where Π_{count} implicitly consists only of the built-in predicates, #int, #succ and $<$):

```
fluents:   bit(X) requires #int(X).
           some_zero.
           all_one_upto(X) requires #int(X).
actions:   add. inv.
initially: total bit(X).
always:    executable add.
           executable inv.

           caused some_zero if -bit(X).

           caused all_one_upto(0) if bit(0).
           caused all_one_upto(Y) if bit(Y), all_one_upto(X), #succ(X,Y).

           caused bit(0) after add, -bit(0).
           caused bit(Y) after add, -bit(Y), all_one_upto(X), #succ(X,Y).
           caused -bit(X) after add, all_one_upto(X).
           caused bit(X) after add, bit(X), -bit(Y), Y < X.
           caused -bit(X) after add, -bit(X), -bit(Y), Y < X.

           caused -bit(X) after inv, not some_zero.
           caused bit(X) after inv, bit(X), some_zero.
           caused -bit(X) after inv, -bit(X), some_zero.

           caused false after not add, not inv.
           caused false after add, inv.
goal:      not some_zero? (i)
```

There are two actions `add` and `inv` corresponding to pressing either of the buttons, which are both always executable. We use the following fluents to model the current state of the counter: `bit`(x) indicates whether bit x is currently set, where $x \in \{0, \ldots, n-1\}$ for an n-Bit counter. Furthermore, we have two auxiliary fluents `some_zero`, which says that there are some unset bits, and `all_one_upto`(x), which says that all bits $i \leq x$ are set. The values of these auxiliary fluents are described by static effects in the first block of causation rules. The effect of action `add` is described by the subsequent block of five causation rules. Intuitively, a bit is toggled if all previous bits are set. The subsequent three causation rules

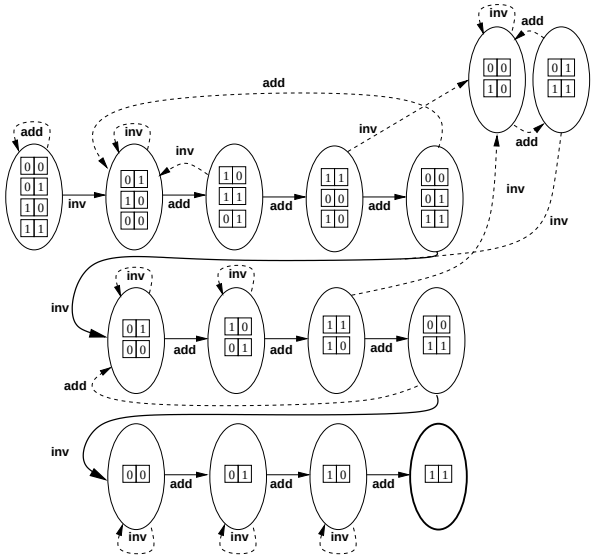

Figure 6.4: Reset a 2-bit counter.

describe the effect of action `inv` as described above. Finally, execution of exactly one action is enforced by the final two constraints.

For a 2-bit counter, by calling $\texttt{DLV}^{\mathcal{K}}$ with

```
$ DLV -FPsec counter.plan -N=1 -planlength=12 -n=1
```

we obtain the following 12-step plan, which is the shortest possible solution, as illustrated in Figure 6.4:

$P = \langle$ {inv}, {add}, {add}, {add}, {inv}, {add}, {add}, {add}, {inv},
{add}, {add}, {add}$\rangle$

As easily seen, the given $\mathcal{K}$ program is `false`-committed, even plain (strictly speaking, the ground instantiation is, since Π_{count} is not empty using built-ins); the only nondeterminism is in the initial state, and exactly one action occurs per step. Still, we see in Figure 6.4 that the problem is hard to solve: The picture illustrates that for secure planning, the number of states is not necessarily an upper bound for the plan length: In this problem the number of (counter) states (2^n, where n is the number of bits), is not an upper bound; the shortest secure plan is $2^n * (2^n - 1)$ for $n = 2$ which means that in any supporting trajectory of the secure plan there are states which are visited more than once.

The conditional effects make a knowledge state encoding again infeasible. Intuitively, the only knowledge "gaining" action here is `inv`, which, depending on the current belief state, collapses the two counter states $0, \ldots, 0$ and $1, \ldots, 1$.

Remark 6.1. *While we have not proven for $n > 2$ that the shortest plan takes always $2^n * (2^n - 1)$ steps, it is obvious that (i) we need at least $2^n - 1$* `inv` *actions plus $2^n - 1$ final* `add` *actions to achieve the goal state after the last* `inv`*, and (ii) that a secure plan of length $2^n * (2^n - 1)$ can always be found.*

In other domains it might even be necessary to step through all *possible belief states, before reaching the goal conformantly, making minimal plan length in worst case double exponential, i.e.* $O(2^{2^n})$ *as mentioned in Section 3.3. A respective example is constructible from the proof of Theorem 3.8 where the Turing machine to encode simply counts from 0 to* 2^{2^n}.

6.3.5 Paint the House Green – Nondeterminism vs. "Forgetting"

"Forgetting" in our sense cannot be emulated by formalisms which adopt a world state view. In those formalisms, leaving fluents open amounts to a disjunction over all possible world states, whereas we can explicitly distinguish between such "totalization" and "unknown" features. In Section 6.3.2 above, we have seen examples of nondeterminism, which in fact makes some fluents unknown. This allowed for a very concise knowledge state encoding in this domain. However, we point out that "forgetting" does not always make sense, as shown by the following example. Intuitively, we might distinguish between different forms of nondeterminism: Whenever the value of a some fluent is completely open after executing an action, this suggests that we can "forget" about it in some sense. On the other hand, nondeterminism is often more "specific", including disjunctive information. For example, we might know that the effect of some action *act* is either $f(a)$ or $f(b)$, but for sure $f(c)$ is false after executing *act*. Here, just leaving fluent $f(\cdot)$ open would be counterproductive, since we know *something*. We illustrate this by another example.

Paint the House Green – Basic Encoding: Assume that you tell `joe` from our Bridge crossing example to paint a blue house green, but `joe` can not distinguish between red and green[29] After `joe` having painted the house, we do not know whether it is green or red, but we do know that it is definitely not blue (true negation), for example. So, "forgetting" about the color might not be a good choice in this example, as we actually know something about the color of the house.

Let us take a closer look at an encoding of this domain in $\mathcal{K}$ where the background knowledge Π_{paint} defines the colors, painters and whether a painter is color blind:

```
c(blue). c(red). c(green).
painter(joe). painter(jack).
colorblind(joe).
```

An encoding of PD_{paint} is shown in Figure 6.5. The first two causation rules express that painting the house in a certain color is successful if either the painter is not color blind or the color is neither red nor green. Furthermore, we add negative knowledge by a static law expressing that whatever color the house currently has, it doesn't have any other color.The next four causation rules, which involve unstratified negation express the two possible outcomes of painting the house green (resp. red) in case the painter is color blind. Finally, we add inertia on positive knowledge on fluent `color`. Negative inertia for `-color` can safely be ignored since `-color` is a static effect for all colors which do not represent the current color of the house by the third causation rule in the encoding of Figure 6.5

[29] A form of color blindness which is known as Daltonism after the Physicist John Dalton.

```
fluents :   color(C) requires c(C).
actions :   paint(C,A) requires c(C), painter(A).
always :    executable paint(C,A).
            noConcurrency.

            caused color(C) after paint(C,A), not colorblind(A).
            caused color(C) after paint(C,A), C != red, C != green.
            caused -color(C1) if color(C), C != C1.

            caused color(red) if not color(green) after paint(green,A), colorblind(A).
            caused color(green) if not color(red) after paint(green,A), colorblind(A).
            caused color(red) if not color(green) after paint(red,A), colorblind(A).
            caused color(green) if not color(red) after paint(red,A), colorblind(A).
            inertial color(C).
initially : color(blue).
```

Figure 6.5: An encoding of the painting domain PD_{paint}

as mentioned above. Clearly, the knowledge states reachable from the initial state in this domain encoding are in one-to-one correspondence with the actual world states.

Remark 6.2. *Note that the effect rules for action* `paint` *suggest that allowing some limited form of disjunction in the syntax of* $\mathcal{K}$ *could be useful for more concise encodings. Indeed, we could define additional macros for disjunction in the* `after`*-part of causation rules as follows:*

```
                                                  caused f if B after f, A1.
caused f if B after A1 v ... v An.     ⇔          ⋮
                                                  caused f if B after f, An.
```

We remark that disjunction in the `caused`*- or* `if`*-parts requires more caution, for instance due to possible head-cycles (cf. Section 2.1.4.1).*

Our goal is to achieve that the color of the house changes, i.e. we accept any color but the initial blue.

```
goal :      -color(blue)? (1)
```

If we combine this line and the domain description in a file `paint.plan` and Π_{paint} in file `paint.bk` `DLV`$^{\mathcal{K}}$ computes the following secure plans:

```
$ DLV -FPsec paint.plan paint.bk

PLAN: paint(green,jack)
PLAN: paint(red,jack)
PLAN: paint(green,joe)
PLAN: paint(red,joe)
```

which indeed are all secure plans in this domain.

Paint the House Green, Incomplete Initial State – World State View: In a further elaboration, we assume that we do not know the initial color of the house. Obviously, we could adopt a world state view, i.e. encoding all possible initial states explicitly by:

```
initially: total color(X).
           forbidden -color(red), -color(green), -color(blue).
```

The resulting secure plans remain the same as above.

Paint the House Green, Incomplete Initial State – Knowledge State View: We can relax our assumptions a bit, leaving the initial state open by simply dropping the `initially:` section completely. This means we do not apply a world state view in the initial state any longer, leaving `color` unknown. This results in a single initial knowledge state, corresponding to three possible actual world states.

Note that intuitively, as opposed to the Square example, we may just leave `color` open in the initial state, since the effect of painting is not conditional and gains further knowledge. The resulting secure plans still remain the same as for the original problem.

Paint the House Green – Can we use "Forgetting"? Note that, however, there is no easy way to express the nondeterminism of action `paint` by "forgetting" like in the Bomb in Toilet domain in Section 6.3.2: For instance, one might come up with a solution like:

```
always:    executable paint(C,A).
           noConcurrency.
           caused color(C) after paint(C,A), not colorblind(A).
           caused color(C) after paint(C,A), C != red, C != green.
           caused -color(C1) if color(C), C != C1.

           caused painted after paint(C,A).
           inertial color(C) if not painted.
```

where `painted` is a new fluent. Here, painting only has an effect if the painter is either not color blind or the color is different from green or red. By the modified frame axiom, the previous color is only inherited to the next state if no painting action occurred. This leaves the value of fluent `color` open in the next state, whenever a color blind painter p executed `paint(red,`p`)` or `paint(green,`p`)`.

We could now modify the goal accordingly from

```
goal:      -color(blue)? (1)
```

by using default negation to

```
goal:      not color(blue)? (1)
```

in order to obtain the same plans as above for this domain, but this does not reflect the effects of the actions correctly: Assume we change our goal objective additionally claiming that the house should neither be red nor blue. In the original formulation of the goal, this is expressed by

```
goal :      −color(blue), −color(red)? (1)
```

and from the above plans, only

```
PLAN: paint(green,jack)
```

remains secure. If we rewrite the goal accordingly in our "forgetting" encoding, we end up with:

```
goal :      not color(blue), not color(red)? (1)
```

which still allows for plans

```
PLAN: paint(green,joe)
PLAN: paint(red,joe)
```

Since we do not know anything about the color after joe painted the house green, resp. red in this formulation both actions seem reasonable plans, which is wrt. to our intention incorrect. We see that in this particular example we need the explicit (negative) knowledge about what is known to be false illustrating the difference between default negation and true negation.

Analogously, we could think of disjunctive goals such as wanting the house to be green or red. In $\mathcal{K}$ we do not directly allow for such disjunctive goals but this could easily be emulated by an additional fluent, for instance named `green_or_red` where we add

```
caused green_or_red if color(red).
caused green_or_red if color(green).
```

to the `always :` section and change the goal to:

```
goal :      green_or_red? (1)
```

While this again works fine for the original encoding, the proposed "forgetting"-version still does not work.

Paint the House Green – Improved Encoding: However, combining the ideas from above we could still apply some improvement on the original encoding: At a closer look we can provide a more concise encoding of the nondeterministic action effect by adding fluent `painted` and changing the `always :` section from Figure 6.5 as follows:

```
always :  executable paint(C, A).
          noConcurrency.
          caused color(C) if not −color(C) after paint(C, A).
          caused −color(C1) if color(C), C != C1.
          caused color(red) if not color(green) after paint(green, A), colorblind(A).
          caused color(green) if not color(red) after paint(red, A), colorblind(A).

          caused painted after paint(C, A).
          inertial color(C) if not painted.
```

Here, the first causation rule states that painting normally works out fine, using default negation. The static effect remains as is.

Only the two special cases expressed by the following causations are needed in order to model the possible exceptions. Each of those two rules form a "guess" together with the first causation rule, in case the painter is color blind.

Note that we kept the modified frame axiom by conditional inertia formulation from the previous encoding here, in order to avoid unwanted nondeterminism by negation as failure "hidden" inside the `inertial` macro. The conditional formulation above overrides inertia, whenever a paint action has occurred. We leave it as an exercise to the reader to show why for instance

```
PLAN: paint(green,jack)
```

is *not* a secure plan if using simply `inertial color(C).` in the final rule. For further discussion of "hidden" default negation, we refer to Section 6.6.2 below.

6.4 Cost Efficient versus Time Efficient Plans

We will now show how the language $\mathcal{K}^c$ can be used to minimize plan length in combination with minimizing the costs of a plan. This is especially interesting in settings where parallel actions are allowed as in our Blocks World formalization above (cf. [KW99, LL01, EFL$^+$03c]).

For such domains with parallel actions, Kautz and Walser propose various optimization criteria, for instance the number of actions needed, or the number of time steps when parallel actions are allowed, as well as combinations of these two criteria [KW99]. By exploiting action costs and proper modeling, we can solve such optimization problems in $\mathcal{K}^c$ as well.

In particular, we will consider the following optimization criteria:

(α) Find a plan with minimal cost (*cheapest plan*) for a given number of steps.

(β) Find a plan with minimal time steps (*shortest plan*).

(γ) Find a shortest among the cheapest plans.

(δ) Find a cheapest among the shortest plans.

Problem (α) is what we have already defined as optimal planning so far. We will now show how to express (β) in terms of optimal cost plans, and how to extend this elaboration with respect to the combinations (γ) and (δ) with elaborations of our Blocks World example from Section 6.2.

6.4.1 Cheapest Plans with Given Plan Length (α)

Let again $\mathcal{P}(l)$ denote planning problem $\mathcal{P}$ where the plan length in goal q is changed to l. We recall the Blocks World encoding from Figure 6.1 above where our goal was to find a cheapest plan wrt. the total number of (possibly parallel) moves.

In DLV$^{\mathcal{K}}$ one can compute optimal plans for given plan length l by using the command-line option `-planlength=`l, i.e. assuming PD_{bw} and Π_{bw} are given in files `bwalpha.plan` and `bw.bk`, we can compute $cost^*_{\mathcal{P}(l)}$ as follows

```
$ DLV -FPopt bwalpha.plan bwalpha.bk -planlength=l
```

resulting for $l = 2$ and $l = 3$ in exactly the plans P_2 and P_3 shown in Section 6.2.

By simply assigning cost 1 to all moves as in the encoding in Figure 6.1, we only solve criterion (α) for our Blocks World example. This can be arbitrarily extended to other cost criteria expressible by action costs. For instance, we have shown some examples in our elaborations of the Bridge Crossing Problem. In particular, we can also minimize the plan length wrt. a given upper bound, which we show next.

6.4.2 Shortest Plans (β)

Intuitively, it should be possible to include the minimization of time steps in the cost function. We describe a preprocessing method which, given a $\mathcal{K}$ planning domain *PD*, a list Q of ground literals, and an upper bound $i \geq 0$ for the plan length, generates a planning problem $\mathcal{P}_\beta(PD, Q, i)$ such that the optimal plans for $\mathcal{P}_\beta$ correspond to shortest plans which reach Q in *PD* in at most i steps, i.e., to plans for $\langle PD, Q\,?\,(l)\rangle$ such that $l \leq i$ is minimal. We assume that no action costs are specified in the original planning domain *PD*, and minimizing time steps is our only target.

First we rewrite the planning domain *PD* to PD_β as follows: We introduce a new distinct fluent `gr` and a new distinct action `finish`, defined as follows:

```
fluents:   gr.
actions:   finish costs time.
```

Intuitively, the action `finish` represents a final action, which we use to finish the plan. The later this action occurs, the more expensive the plan as we assign `time` as cost. The fluent `gr` ("goal reached") shall be true and remain true as soon as the goal has been reached, and it is triggered by the `finish` action.

This can be modeled in $\mathcal{K}^c$ by adding the following statements to the `always` section of the program:

```
executable finish if Q, not gr.
caused gr after finish.
caused gr after gr.
```

Furthermore, we want `finish` to occur exclusively and we want to block the occurrence of any other action once the goal has been reached. Therefore, for every action `A` in *PD*, we add

```
nonexecutable A if finish.
```

and add `not gr` to the `if`-part of each executability condition for `A`. Finally, to avoid any inconsistencies from static or dynamic effects as soon as the goal has been reached, we add `not gr` to the `if`-part of any causation rule of the *PD* except `nonexecutable` rules which remain unchanged.[30]

[30] There is no need to rewrite `nonexecutable` rules because the respective actions are already "switched off" by rewriting the executability conditions.

We define now $\mathcal{P}_\beta(PD, Q, i) = \langle PD_\beta, \mathtt{gr}\,?(i+1)\rangle$. We take $i+1$ as the plan length since we need one additional step to execute the `finish` action.

By construction, it is easy to see that any optimal plan $P = \langle A_1, \ldots, A_j, A_{j+1}, \ldots, A_{i+1}\rangle$ for the planning problem $\mathcal{P}_\beta$ must have $A_{j+1} = \{\mathtt{finish}\}$ and $A_{j+2} = \ldots = A_{i+1} = \emptyset$ for some $j \in \{0, \ldots, i\}$. We thus have the following desired property.

Proposition 6.1. *The optimal plans for $\mathcal{P}_\beta$ are in 1-1 correspondence to the shortest plans reaching Q in PD. That is, $P = \langle A_1, \ldots, A_{j+1}, \emptyset, \ldots, \emptyset\rangle$ is an optimal optimistic plan for $\mathcal{P}_\beta(PD, Q, i)$ and $A_{j+1} = \{\mathtt{finish}\}$ if and only if $P' = \langle A_1, \ldots, A_j\rangle$ is an optimistic plan for $\langle PD, Q\,?\,(j)\rangle$ where $j \in \{0, \ldots, i\}$, and $\langle PD, Q\,?\,(j')\rangle$ has no optimistic plan for each $j' < j$.*

In our Blocks World example, using this method we get all 2-step plans, if we choose $i \geq 2$.

To compute shortest plans over all plan lengths, we can set the upper bound i large enough such that plans of length $l \leq i$ are guaranteed to exist. A trivial such bound is the total number of legal states which, however, is in general exponential in the number of fluents. Remarkably, as we have seen in Section 3.3, for secure planning this bound is even higher in general.

However, many typical applications have an inherent, much smaller bound on the plan length. For instance, in a Blocks World with n blocks, any goal configuration can be reached within at most $2n - s_{init} - s_{goal}$ steps, where s_{init} and s_{goal} are the numbers of stacks in the initial and the goal state, respectively.[31] Therefore, 6 is an upper bound for the plan length of our simple instance.

We remark that this approach for minimizing plan length is only efficient if an upper bound close to the optimum is known. Searching for a minimum length plan by iteratively increasing the plan length may be much more efficient if no such bound is known, since a weak upper bound can lead to an explosion of the search space (cf. the benchmarks in Section 8).

For computing the shortest plans for our Blocks World instance from above, we modify the program from Figure 6.1 as follows:

- remove the `costs`-part from action declaration for `move`.
- add action `finish` and fluent `gr` and the respective rules as defined above, in particular change the executability for `move` and the final four causation rules from the original program to

```
nonexecutable move(B,L) if finish.
executable move(B,L) if B!=L, not gr.

caused on(B,L) if not gr after move(B,L).
caused blocked(B) if on(B1,B), not gr.
caused moved(B) if not gr after move(B,L).
caused on(B,L) if not moved(B), not gr after on(B,L).
```

[31] One can solve any Blocks World problem sequentially by first unstacking all blocks which are not on the table ($n - s_{init}$ steps) and then building up the goal configuration ($n - s_{goal}$ steps).

- we further remove the original goal and add:

```
always : executable finish if on(1,3), on(3,table), on(2,4),
                              on(4,table), on(6,5), on(5,table), not gr.
goal :   gr?(7)
```

  where plan length 7 of the modified problem $\mathcal{P}_{bw,\beta}$ emerges from the upper bound 6 plus one additional time step for executing `finish`.

Upon call with the modified file `bwbeta.plan`

```
$ DLV -FPopt bwbeta.plan bw.bk
```

`DLV`$^{\mathcal{K}}$ returns the indented shortest plan:

```
PLAN: move(1,table), move(3,table), move(5,table);
      move(1,3), move(2,4), move(6,5); finish:3;
      (no action); (no action); (no action); (no action)  COST: 3
```

i.e., the plan

$P_\beta = \langle \{\texttt{move}(1,\texttt{table}), \texttt{move}(3,\texttt{table}), \texttt{move}(5,\texttt{table})\},$
$\{\texttt{move}(1,3), \texttt{move}(2,4), \texttt{move}(6,5)\},\ \{\texttt{finish}\},\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset\rangle$

with $cost_{\mathcal{P}_{bw,\beta}}(P_\beta) = 3$, corresponding to P_2 from above. Using our method, the shortest plan length can be easily derived from $cost_{\mathcal{P}_{bw,\beta}}(P_\beta) - 1 = 2$.

6.4.3 Shortest among the Cheapest Plans (γ)

In the previous subsection, we have shown how to calculate shortest plans for $\mathcal{K}$ programs without action costs. Combining arbitrary $\mathcal{K}^c$ programs and the rewriting method described there is easy. If we want to find a shortest among the cheapest plans, we can use the same rewriting, with just a little change. All we have to do is setting the costs of all actions except `finish` at least as high as the highest possible cost of the `finish` action. The highest action cost for `finish` is obviously the plan length $i + 1$. So, we simply modify all action declarations

```
A requires B costs C where D.
```

in $\mathcal{P}_\beta$ by multiplying the costs with factor $i + 1$:

```
A requires B costs C1 where C1 = (i + 1) * C,  D.
```

This lets all other action costs take priority over the cost of `finish` and we can compute plans satisfying criterion (γ). Let $\mathcal{P}_\gamma$ denote the resultant planning problem. Then we have:

Proposition 6.2. *The optimal plans for $\mathcal{P}_\gamma$ are in 1-1 correspondence to the shortest among the cheapest plans reaching Q in PD within i steps. That is, $P = \langle A_1, \ldots, A_{j+1}, \emptyset, \ldots, \emptyset\rangle$ is an optimal optimistic plan for $\mathcal{P}_\gamma(PD, Q, i)$ and $A_{j+1} = \{\texttt{finish}\}$ if and only if (i) $P' = \langle A_1, \ldots, A_j\rangle$ is a plan for $\mathcal{P}_j = \langle PD, Q\,?\,(j)\rangle$, where $j \in \{0, \ldots, i\}$, and (ii) if $P'' = \langle A_1, \ldots, A_{j'}\rangle$ is any plan for $\mathcal{P}_{j'} = \langle PD, Q\,?\,(j')\rangle$ where $j' \leq i$, then either $cost_{\mathcal{P}_{j'}}(P'') > cost_{\mathcal{P}_j}(P')$ or $cost_{\mathcal{P}_{j'}}(P'') = cost_{\mathcal{P}_j}(P')$ and $j' \geq j$.*

```
fluents:   on(B,L) requires block(B), location(L).
           blocked(B) requires block(B).
           moved(B) requires block(B).
           gr.

actions:   move(B,L) requires block(B), location(L) costs C where C = 7 * 1.
           finish costs time.

always:    executable move(B,L) if B != L, not gr.
           nonexecutable move(B,L) if blocked(B).
           nonexecutable move(B,L) if blocked(L).
           nonexecutable move(B,L) if move(B1,L), B < B1, block(L).
           nonexecutable move(B,L) if move(B,L1), L < L1.
           nonexecutable move(B,B1) if move(B1,L).

           caused on(B,L) if not gr after move(B,L).
           caused blocked(B) if on(B1,B), not gr.
           caused moved(B) if not gr after move(B,L).
           caused on(B,L) if not moved(B), not gr after on(B,L).

           executable finish if on(1,3),on(3,table),on(2,4),on(4,table),
                                on(6,5),on(5,table),not gr.
           caused gr after finish.
           caused gr after gr.
           nonexecutable move(B,L) if finish.

initially: on(1,2). on(2,table). on(3,4). on(4,table). on(5,6). on(6,table).

goal:      gr? (7)
```

Figure 6.6: Computing the shortest plan for a Blocks World instance with a minimum number of actions

Figure 6.6 shows $\mathcal{P}_{bw,\gamma}$ for our Blocks World instance where $i = 6$. One optimal plan for $\mathcal{P}_{bw,\gamma}$ is

$$P = \langle\ \{\texttt{move(3,table)}\},\ \{\texttt{move(1,3)},\texttt{move(5,table)}\},\ \{\texttt{move(2,4)},\texttt{move(6,5)}\},\ \{\texttt{finish}\},\ \emptyset,\ \emptyset,\ \emptyset\ \rangle,$$

which has $cost_{\mathcal{P}_\gamma}(P) = 39$. As easily seen, this plan corresponds to P_3 from above.

We can now compute the optimal cost wrt. optimization (γ) by subtracting the cost of `finish` and dividing by $i+1$: $(39-4) \div (i+1) = 35 \div 7 = 5$. Thus, we need a minimum of 5 moves to reach the goal. The minimal number of steps is obviously all steps, except the final `finish` action, i.e. 3. Thus, we need at least 3 steps for a plan with five moves.

The resp. call to DLV$^{\mathcal{K}}$ is as follows:

```
$ DLV -FPopt bwgamma.plan bw.bk -N=7
```

resulting in the above plan:

```
PLAN: move(3,table):7, move(5,table):7;
      move(1,3):7, move(6,5):7;
      move(2,4):7; finish:4;
      (no action); (no action); (no action)  COST: 39
```

where option -N=7 is needed because of the costs definition for `move` in $\mathcal{P}_{bw,\gamma}$ which uses the integer arithmetics built-in "$*$" as shown in Figure 6.6.

6.4.4 Cheapest among the Shortest Plans (δ)

Again, we can use the rewriting for optimization (β). The cost functions have to be adapted similarly as in the previous subsection, such that now the cost of the action `finish` takes priority over all other actions costs. To this end, it is sufficient to set the cost of `finish` high enough, which is achieved by multiplying it with a factor F higher than the sum of all action costs of all legal action instances at all steps $j = 1, \ldots, i+1$. Let $\mathcal{P}_\delta$ denote the resulting planning problem. We have:

Proposition 6.3. *The optimal plans for $\mathcal{P}_\delta$ are in 1-1 correspondence to the cheapest among the shortest plans reaching Q in PD within i steps. More precisely, $P = \langle A_1, \ldots, A_{j+1}, \emptyset, \ldots, \emptyset \rangle$ is an optimal optimistic plan for $\mathcal{P}_\delta(PD, Q, i)$ and $A_{j+1} = \{\texttt{finish}\}$ if and only if (i) $P' = \langle A_1, \ldots, A_j \rangle$ is a plan for $\mathcal{P}_j = \langle PD, Q\,?\,(j) \rangle$, where $j \in \{0, \ldots, i\}$, and (ii) if $P'' = \langle A_1, \ldots, A_{j'} \rangle$ is any plan for $\mathcal{P}_{j'} = \langle PD, Q\,?\,(j') \rangle$ where $j' \leq i$, then either $j' > j$, or $j' = j$ and $cost_{\mathcal{P}_{j'}}(P'') \geq cost_{\mathcal{P}_j}(P')$.*

In our example, there are 36 possible moves. Thus, we could take $F = 36 * (i+1)$ and would set the costs of `finish` to `time` $* 36 * (i+1)$. However, we only need to take into account those actions which can actually occur simultaneously. In our example, at most six blocks can be moved in parallel. Therefore, it is sufficient to set $F = 6 * (i+1)$ and assign `finish` cost `time` $* F =$ `time` $* 42$. Accordingly, the action declarations are modified as follows:

```
actions:   move(B,L) requires block(B), location(L) costs 1.
           finish costs C where C = time * 42.
```

An optimal plan for the modified planning problem $\mathcal{P}_\delta$ is:

$P = \langle$ {move(1, table), move(3, table), move(5, table)},
{move(1, 3), move(2, 4), move(6, 5)}, {finish}, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset\rangle$

We have $cost_{\mathcal{P}_\delta}(P) = 132$. Here, we can compute the optimal cost wrt. optimization (δ) by simply subtracting the cost of `finish`, i.e. $132 - 3 * 42 = 6$, since `finish` occurs at time point 3. Consequently, we need a minimum of 6 moves for a shortest plan, which has length $3 - 1 = 2$.

The resp. call to DLV$^{\mathcal{K}}$ is as follows:

```
$ DLV -FPopt bwdelta.plan bw.bk -N=1000
```

resulting in the above plan:

```
PLAN: move(1,table):1, move(3,table):1, move(5,table):1;
      move(1,3):1, move(2,4):1, move(6,5):1;
      finish:126;
      (no action); (no action); (no action); (no action) COST: 132
```

Note that we have to set the integer limit high enough again, in order to produce correct results. Any value for `-N=`n below 7*42=294 will cause bogus results; due to the limitations outlined in the previous chapter, for $n < 294$ `finish` will not produce any violation costs in $lp^w(\mathcal{P})$ when executed at time 7, since the resp. constraint

```
:~ costs$_finish$(6,294). [294:1]
```

will not be instantiated by DLV.

Indeed, we have seen that (and how) the optimization problems (α) through (δ) can be represented in $\mathcal{K}^c$. We remark that the transformations $\mathcal{P}_\beta$, $\mathcal{P}_\gamma$, and $\mathcal{P}_\delta$ all work under the restrictions to secure and/or sequential plans as well with respect to the integrated computation implemented in $\text{DLV}^{\mathcal{K}}$. Furthermore, we have seen some of the restrictions of this transformations with respect to the current implementation which we will try to circumvent in Section 7.1.1 below.

6.5 Route Planning - Variants of Traveling Salesperson

As another illustrating example for optimal cost planning, we will now introduce an elaboration of the Traveling Salesperson Problem with exceptional time-dependent action costs.

Traveling Salesperson Problem (TSP). We start with the classical Traveling Salesperson Problem (TSP), where we have a given set of cities and connections (e.g., roads, airways) of certain costs. We want to know a most economical round trip which visits all cities exactly once and returns to the starting point (if such a tour exists). Figure 6.7 shows an instance representing the capitals of all Austrian provinces. The dashed line is a flight connection, while all other connections are roads; each connection is marked with the costs in traveling hours.

brg ... Bregenz
eis ... Eisenstadt
gra ... Graz
ibk ... Innsbruck
kla ... Klagenfurt
lin ... Linz
sbg ... Salzburg
stp ... St. Pölten
vie ... Vienna

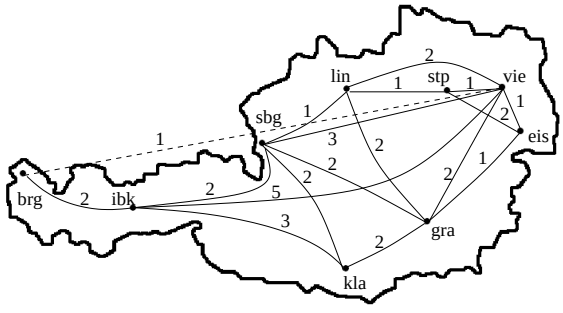

Figure 6.7: TSP in Austria

We represent this in $\mathcal{K}^c$ as follows. The background knowledge Π_{TSP} defines two predicates `city(C)` and `conn(F,T,C)` representing the cities and their connections with associated costs. Connections can be traveled in both ways:

```
conn(brg,ibk,2). conn(ibk,sbg,2). conn(ibk,vie,5). conn(ibk,kla,3).
conn(sbg,kla,2). conn(sbg,gra,2). conn(sbg,lin,1). conn(sbg,vie,3).
conn(kla,gra,2). conn(lin,stp,1). conn(lin,vie,2). conn(lin,gra,2).
conn(gra,vie,2). conn(gra,eis,1). conn(stp,vie,1). conn(eis,vie,1).
conn(stp,eis,2). conn(vie,brg,1).
conn(B,A,C) :- conn(A,B,C).
city(T) :- conn(T,_,_).
```

A possible encoding of TSP starting in Vienna (`vie`) is the $\mathcal{K}^c$ program in Figure 6.8. It includes one action for traveling from one city to another.

```
fluents:    unvisited.
            in(C) requires city(C).
            visited(C) requires city(C).
actions:    travel(X,Y) requires conn(X,Y,C) costs C.
always:     executable travel(X,Y) if in(X).
            nonexecutable travel(X,Y) if visited(Y).
            nonexecutable travel(X,vie) if city(C), not visited(C), C != vie.
            caused unvisited if city(C), not visited(C).
            caused in(Y) after travel(X,Y).
            caused visited(Y) after travel(X,Y).
            inertial visited(C).
noConcurrency.
initially: in(vie).
goal:       notunvisited, in(vie)? (9)
```

Figure 6.8: Traveling Salesperson

Note that compared to our original encoding provided in [EFL+03c] this encoding only needs one action. Every city traveled to is marked as `visited`. The goal is to reach Vienna, with having all cities visited, where the first `nonexecutable` statement prohibits traveling to a city already visited. The second `nonexecutable` statement adds control knowledge in some sense, pruning attempts to visit Vienna, as long as there are some other unvisited cities which has proven reasonable also in our experiments. We remark that for this example the only necessary frame axiom is the simple inheritance over states for fluent `visited`. Since the value of `visited` can never be invalidated once a city has been visited, there is no need to use the `inertial` macro here and we could equivalently write `caused visited after visited.` instead. As for the fluent `in`, although any no-op step would lead to "forget" about `in` we are implicitly only interested in plans where exactly one action `travel` per time occurs, such that the value of `in` is always determined and no extra frame axiom is needed.

The problem has ten optimal 9-step solutions with cost 15. We show only the first five here, as the others are symmetrical:

$P_1 = \langle$ {travel(vie, stp)}, {travel(stp, eis)}, {travel(eis, gra)}, {travel(gra, lin)},
{travel(lin, sbg)}, {travel(sbg, kla)}, {travel(kla, ibk)}, {travel(ibk, brg)},
{travel(brg, vie)} $\rangle$
$P_2 = \langle$ {travel(vie, eis)}, {travel(eis, stp)}, {travel(stp, lin)}, {travel(lin, sbg)},
{travel(sbg, gra)}, {travel(gra, kla)}, {travel(kla, ibk)}, {travel(ibk, brg)},
{travel(brg, vie)} $\rangle$
$P_3 = \langle$ {travel(vie, eis)}, {travel(eis, stp)}, {travel(stp, lin)}, {travel(lin, gra)},
{travel(gra, kla)}, {travel(kla, sbg)}, {travel(sbg, ibk)}, {travel(ibk, brg)},
{travel(brg, vie)} $\rangle$
$P_4 = \langle$ {travel(vie, lin)}, {travel(lin, stp)}, {travel(stp, eis)}, {travel(eis, gra)},
{travel(gra, kla)}, {travel(kla, sbg)}, {travel(sbg, ibk)}, {travel(ibk, brg)},
{travel(brg, vie)} $\rangle$
$P_5 = \langle$ {travel(vie, gra)}, {travel(gra, eis)}, {travel(eis, stp)}, {travel(stp, lin)},
{travel(lin, sbg)}, {travel(sbg, kla)}, {travel(kla, ibk)}, {travel(ibk, brg)},
{travel(brg, vie)} $\rangle$

TSP with variable costs. Let us now consider an elaboration of TSP, where we assume that the costs of traveling different connections may change during the trip. Note that three of the five solutions in our example above include traveling from St.Pölten to Eisenstadt or vice versa on the second day. Let us now assume that the salesperson, who starts on Monday, has to face some exceptions which might increase the cost of the trip. For instance, (i) heavy traffic jams are expected on Tuesdays on the route from St.Pölten to Eisenstadt or (ii) the salesperson shall not use the flight connection between Vienna and Bregenz on Mondays as only expensive business class tickets are available on this connection in the beginning of the week. So we have to deal with different costs for the respective connections depending on the particular day.

To this end, we first add a new predicate cost(A,B,W,C) to the background knowledge Π_{TSP} representing the cost C of traveling connection A to B on weekday W which can take exceptional costs into account:

```
cost(A,B,W,C) :- conn(A,B,C), #int(W), 0 < W, W <= 7, not ecost(A,B,W).
ecost(A,B,W)  :- conn(A,B,C), cost(A,B,W,C1), C != C1.
```

The original costs in the predicate conn(A,B,C) now represent defaults, which can be overridden by explicitly adding different costs. For instance, to represent the exceptions (i) and (ii), we add:

```
cost(stp,eis,2,10).  cost(vie,brg,1,10).
```

setting the exceptional costs for these two critical connections to 10. Weekdays are coded by integers from 1 (Monday) to 7 (Sunday). We represent a mapping from time steps to the weekdays by the following rules which we also add to Π_{TSP}:

```
weekday(1,1).
weekday(D,W) :- D = D1 + 1, W = W1 + 1, weekday(D1,W1), W1 < 7.
```

```
weekday(D, 1) :- D = D1 + 1, weekday(D1, 7).
```

Note that although the modified background knowledge Π_{TSP} is not stratified (since `cost` is defined by cyclic negation), it has a total well-founded model, and thus a unique answer set.

Finally, we change the costs of traveling in the $\mathcal{K}^c$ program from Figure 6.8:

```
actions:    travel(X,Y) requires conn(X,Y,C1) costs C
                        where weekday(time,W), cost(X,Y,W,C).
```

Since now the costs for P_1 (which includes traveling from St.Pölten to Eisenstadt) on the second day have increased due to exception (i), only four of the plans from above remain optimal. Note that unlike the default costs, exceptional costs do not apply bidirectionally, so the exception does not affect P_2 and P_3. Furthermore, due to exception (ii) the symmetrical round trips starting with the flight trips to Bregenz are no longer optimal.

The presented encoding proves to be very flexible, as it allows for adding arbitrary exceptions for any connection on any weekday by simply adding the respective facts; moreover, even more involved scenarios, where exceptions are defined by rules, can be modeled.

6.5.1 A Small Example for Planning under Resource Restrictions

Although planning with resources is not the main target of our approach, the following encoding shows that action costs can also be used in order to model optimization of resource consumption in some cases. An important resource in real world planning is money. For instance, let us consider a problem about buying and selling [LL01]:

> "I have \$6 in my pocket. A newspaper costs \$1 and a magazine costs \$3. Do I have enough money to buy one newspaper and two magazines?"

In $\mathcal{K}^c$, this can be encoded in a very compact way by the following background facts:

```
item(newspaper, 1).  item(magazine, 2).
```

combined with the following short $\mathcal{K}^c$ program:

```
actions:    buy(Item,Number) requires item(Item,Price), #int(Number)
                             costs C where C = Number * Price.
fluents:    have(Item,Number) requires item(Item,Price), #int(Number).
always:     executable buy(Item,Number).
            nonexecutable buy(Item,N1) if buy(Item,N2), N1 < N2.
            caused have(Item,Number) after buy(Item,Number).
goal:       have(newspaper,1), have(magazines,2)?(1)
```

The action `buy` is always executable, but one must not buy two different amounts of a certain item at once. Obviously, no admissible plan wrt. cost 6 exists, as the optimal plan for this problem, $\langle\{\texttt{buy(newspaper,1)}, \texttt{buy(magazine,2)}\}\rangle$ has $cost^*_{\mathcal{P}} = 7$. Therefore, the answer to the problem is "no."

The problem can be solved by the following call to our prototype:

```
$ DLV -FP buying.bk buying.plan -N=10 -planlength=1 -costbound=6
```

Correctly, no admissible plan is found. When calling the system again without cost bound, the prototype calculates the following optimal cost plan:

```
PLAN : buy(newspaper, 1) : 1,  buy(magazine, 2) : 6   COST :  7
```

Again, when solving th problem with DLV$^{\mathcal{K}}$ one has to bear in mind to set option `-N` properly. As for the cost function in the `where`-part of the declaration of action `buy` (see Section 6.5.1), buying 2 magazines would result in C to be instantiated with 6. If now option `-N` would be set too low, we get a bogus result, for instance:

```
$ DLV -FP buying.bk buying.plan -N=3 -planlength=1 -costbound=6
```

computes the same plan as above but with the wrong cost value `COST : 1`.

Our approach considers only positive action costs and does not directly allow modeling full consumer/producer/provider relations on resources in general, in favor of a clear non-ambiguous definition of optimality. For instance, by allowing negative costs one could always add a producer action to make an existing plan cheaper, whereas in our approach costs are guaranteed to increase monotonic, allowing for a clear definition of plan costs and optimality.

On the other hand, we can encode various kinds of resource restrictions by using fluents to represent these resources. We can then model production/consumption as action effects on these fluents and add restrictions as constraints. This allows us to model even complex resource or scheduling problems; optimization, however, remains restricted to action costs.

In this context, Lee and Lifschitz [LL01, LL03] proposed an interesting and useful extension of the action language $\mathcal{C}+$ called "Additive Fluents" which enables the user to flexibly model such producer/consumer relations in their language, by incremental/decremental effects and cumulative effects of parallel such productions and consumptions. Weak constraints or DLV's `#sum` aggregate which can be used to partly emulate these additive fluents are still limited to aggregation of positive values by DLV's restrictions on integer arithmetics. However, [LL01, LL03] do only allow for fixed limits via constraints on these fluents but do not offer means of optimization.

6.6 Features and Pitfalls

After having presented multiple aspects of knowledge representation in $\mathcal{K}^c$ by means of several well-known and also novel planning examples, we now summarize and discuss the features and pitfalls of encoding domains in our language in more detail.

We have seen that default negation and the concepts of $\mathcal{K}$ provide a flexible tool for knowledge representation in the field of planning but using negation as failure also involves some subtleties via the full freedom of normal logic programs to describe state constraints. In analogy to the term "Planning as Satisfiability" (coined by Kautz and Selman) our approach may well be conceived as "Planning as Answer Set Programming" or even "Answer Set Programming as Planning", to some extent.

In general, $\mathcal{K}$ (resp. $\mathcal{K}^c$) can not be seen as a classical action language where transitions are defined between completely defined world states or sets of such states (i.e. belief states) although this view can be expressed as shown for instance in Sections 6.3.4 and 6.3.1. In

fact, the knowledge state view implicit to the semantics of $\mathcal{K}$, while allowing great flexibility as shown above, requires the $\mathcal{K}$ user to know about basic principles of logic programming and especially how to deal with non-monotonic (default) negation.

We can state two major design principles in this context:

Design Principle 1: Only describe what is necessary.[32]

Design Principle 2: Forget unnecessary information rather than keep complete state information explicitly, where possible.[32]

Knowledge state encodings relieve the user somehow from encoding every possible constraint on legal states of a particular domain by simply leaving "irrelevant" information open. We have discussed the applicability of the knowledge state view vs. the world state view and the concept of "forgetting" about fluents with illustrative examples in the "Bomb in the Toilet" and "House Painting" domains.

In order to design planning domains in $\mathcal{K}$, one has to be aware of the inherent non-monotonicity of the knowledge state view by default negation (explicitly or implicitly via the `inertial`, `total` and `default` macros). We will discuss the use of macros in this context below.

Informally, a transition $\langle s, A, s' \rangle$ in $\mathcal{K}^c$ can be viewed as a transition between (answer sets of) normal logic programs where causation rules of the form

$$\begin{aligned} &\texttt{caused } f \texttt{ if } b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_l \\ &\qquad \texttt{after } a_1, \ldots, a_m, \texttt{not } a_{m+1}, \ldots, \texttt{not } a_n. \end{aligned}$$

form a logic program $\Pi_{s'}$ consisting of all rules r

$$f \texttt{ :- } b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_l.$$

such that $\{a_1, \ldots, a_m\} \in s \cup A$ and $\{a_{m+1}, \ldots, a_n\} \cap (s \cup A) = \emptyset$. $\Pi_{s'}$ then has all legal successor states for s and A as its answer sets.

We will give another example below showing the strength of this logic programming view in planning: Modeling transitive closure in $\mathcal{K}$ is more concise and in our opinion more natural than in similar formalisms.

6.6.1 Transitive Closure

The next short example shows how to model transitive closure in $\mathcal{K}$. This is straightforward due to the answer-set-programming-based semantics of $\mathcal{K}$. Let us assume there is a fluent `on(B,L)` which represents whether a block `B` resides on location `L` in the Blocks World.

Now, we want to define causation rules for a fluent `above(B,L)` which states that block `B` resides somewhere above location `L`. This can be modeled by static rules as follows:

```
caused above(B,L) if on(B,L).
caused above(B,L) if on(B,B1), above(B1,L).
```

[32] However, both these statements should also be viewed in the light of "elaboration tolerance" in the sense of McCarthy [McC99b]. Flexible frameworks such as our language $\mathcal{K}$ leave much of the responsibility how far domain- and problem-specific knowledge is exploited up to the user.

While in $\mathcal{K}$ these two rules sufficiently describe the values of fluent `above`, in action language $\mathcal{C}$ or other formalisms based on complete state information, we would have to add negative information on fluent `above` explicitly in order to avoid nondeterminism.

On the other hand, we can easily "complete" each knowledge state wrt. fluent `above` in the example, for instance by adding:

```
default -above(X,Y).
```

6.6.2 Using Macros

Here, we will discuss by example what the user has to bear in mind when using macros, mainly concerning knowledge states vs. world states again.

"Hidden" Default Negation in Macros

As we have already seen in the previous examples, default negation "`not` " allows a great degree of freedom and flexibility in the encoding of planning domains. However, default negation and nondeterminism might sometimes not be obvious when dealing with $\mathcal{K}$ macros. For instance, `inertial` statements might interfere with other rules with default negation as seen in the final variant of the example of Section 6.3.5. Furthermore, combinations of `inertial` and `default` might cause eventually unintended nondeterminism: Consider for instance the rules

```
default -hasLamp(joe).
inertial hasLamp(joe).
```

in a state $s = \{$`hasLamp(joe)`$\}$, with the empty action set $A = \emptyset$. Here, there are two legal transitions $\langle s, A, s' \rangle$ and $\langle s, A, s'' \rangle$, where $s' = \{$`hasLamp(joe)`$\}$, resp. $s'' = \{$`-hasLamp(joe)`$\}$ as in the semantics of $\mathcal{K}$ inertia does not take priority over defaults.

For the sake of simplicity we refrained from a more complicated definition of `default` and `inertial` such that this priority would hold. We could proceed similarly to encoding orders among Defaults in Reiter's Default logic as proposed for example in [Luk90]: A simple redefinition of `default` and `inertial` such that inertia takes priority over default could look as follows:

`inertial f if B after A.` $\Leftrightarrow$ `caused` $inert_{\texttt{f}}$ `if B after f, A.`
`caused f if not` $\neg$`.f, B after f, A.`

`default f.` $\Leftrightarrow$ `caused f if not` $\neg$`.f,not` $inert_{\neg\texttt{.f}}$`.`

where for any positively (resp. negatively) inertial fluent fl we additionally have to add a new distinct fluent $inert_{fl}$ (resp. $inert_{\neg fl}$) in a declaration analogous to the declaration of fl wrt. the `requires`-part.

Note that we have used a similar "trick" above to override inertia in the improved formulation of the "Paint the House Green" example.

Remark 6.3. *We remark that in the macro definition above, we assume that* `B` *is "reasonable" in the sense that it does not contain* `f` *(nor* `-f,not f,not -f`*, respectively) itself, since the occurrence of* `f` *in the* `if`*-part could again cause unstratified negation.*

Nonexecutability and Knowledge States

Similar considerations apply to the executability/non-executability of actions. For instance, in the above example, assume that we had expressed executability of `takeLamp` in line 9 of Figure 3.1 (see p. 38) by the following statement:

`nonexecutable takeLamp(X) if hasLamp(Y), at(X,SX), at(Y,SY), otherSide(SX,SY).`

Intuitively, this says that the lamp might not be taken by `X` if we know that `Y` holds it and `Y` is on the other side of the bridge. Here, default negation is intuitively "hidden" in a constraint. [33] While one might think on a first glance that this is a good way to model action `takeLamp` correctly, assume that nobody has the lamp, which could happen for example in our Scenario $\mathcal{P}_{BCPsec}$ (see p. 41) where the lamp might be lost. With this reformulation the lamp could "rematerialize" by being taken by any person at any point after being lost. So, modeling it this way is somehow "elaboration intolerant", in the sense of [McC99b] while it works well in the basic formulation of $\mathcal{P}_{BCP}$.

[33] In fact, any rule **caused false if** A **after** B. can semantically equivalent be rewritten to a rule **caused p if not p**, A **after** B. where **p** is a new propositional fluent not occurring elsewhere in the domain.

Chapter 7

Language and System Extensions

In this chapter, we will discuss some language extensions which are either under investigation or have already found their way in the implemented DLV$^{\mathcal{K}}$ system. Furthermore we will discuss open implementation issues and optimizations.

7.1 Language Extensions

In the following we will discuss some possible syntactic and semantic extensions of $\mathcal{K}^c$ which are currently under investigation. We will also sketch the realization of these features and their impact on computational complexity.

7.1.1 Improving the Implementation of $(\beta) - (\delta)$

When showing a method to encode the computation of shortest plans in $\mathcal{K}^c$ by means of action costs above (cf. Section 6.4), we have seen that combining the minimization of several criteria in parallel such as finding shortest among cheapest plans or cheapest among shortest plans requires some overhead. In particular for optimization (δ), i.e. finding the cheapest among the shortest plans, it might not always be obvious beforehand what the cost value of `finish` should be. Recall that we had to set the costs of action `finish` to a constant value higher than all other costs in order to achieve priority over all other action costs for optimization (δ); similar considerations were necessary for (γ). This approach had some drawbacks: First, as already mentioned, the particular cost value to be chosen for action `finish` is not necessarily obvious for an arbitrary domain; next, the optimal cost value with respect to the original criterion is no longer immediately visible from the plans after our preprocessing step. Furthermore, keeping in mind the limit for integer arithmetics (command-line option `-N`) in DLV$^{\mathcal{K}}$ for (γ) and (δ) seems to be inconvenient when actually solving the problems with our system.

Note that we can circumvent the adaption of cost values wrt. action `finish` in (γ) and (δ) by a simple and straightforward extension of the $\mathcal{K}^c$ syntax and/or translation $lp^w(\mathcal{P})$

which will be discussed next.

7.1.1.1 Cost Levels

When introducing weak constraints in DLV in Section 2.1.5.1 we mentioned that DLV allows for cost levels. Cost levels could with minor modifications of $lp^w(\mathcal{P})$ safely be allowed in $\mathcal{K}^c$ as well. To this end, we have to

(1) adapt the syntax of action declarations wrt. the `costs`-part by also allowing declarations of the form:

$$p(X_1, \ldots, X_n) \texttt{ requires } t_1, \ldots, t_m \texttt{ costs } C : L \texttt{ where } c_1, \ldots, c_k. \tag{7.1}$$

where for level L the same syntactical restrictions apply as for C in the original declarations of the form (3.5) in Section 3.2.

(2) modify Step 7 in of $lp^w(\mathcal{P})$ in Sections 4.3.2 accordingly:

Step 7 (Action Costs): For any action declaration d of form (7.1) with a non-empty `costs`-part, add:
(i) A new rule r_d of the form

$$\begin{aligned} \texttt{cost}_\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}, \texttt{C}\theta, \texttt{L}\theta) \texttt{ :- } \quad & \texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}), \texttt{t}_1, \ldots, \texttt{t}_\texttt{m}, \\ & \texttt{c}_1\theta, \ldots, \texttt{c}_\texttt{k}\theta, \texttt{U} = \texttt{T} + 1. \end{aligned} \tag{7.2}$$

where $\texttt{cost}_\texttt{p}$ is a new symbol, `T` and `U` are new variables and $\theta = \{\texttt{time} \rightarrow \texttt{U}\}$. As an optimization, $\texttt{U} = \texttt{T} + 1$ is only present if `U` occurs elsewhere in r_d. (ii) A weak constraint wc_d of the form

$$:\sim \texttt{ cost}_\texttt{p}(\texttt{X}_1, \ldots, \texttt{X}_\texttt{n}, \texttt{T}, \texttt{C}, \texttt{L}). \; [\texttt{C} : \texttt{L}] \tag{7.3}$$

This extension allows the user to encode $\mathcal{P}_\gamma$ (resp. $\mathcal{P}_\delta$) simply by assigning `finish` a lower (resp. higher) cost level than all the other actions in $\mathcal{P}$.

The extension to use cost levels in $\mathcal{K}^c$ action declarations has already been implemented in the latest $\texttt{DLV}^\mathcal{K}$ version to be released.

7.1.1.2 Alternative Translation

We propose another alternative which also makes use of cost levels in the translation $lp^w(\mathcal{P})$ itself in order to compute shortest plans. Here, no preprocessing of the domain $\mathcal{P}$ is necessary but the translation itself is affected.

Shortest plan in the sense of optimizations (β)–(δ) where an upper bound i for the plan length is given can be achieved alternatively by a simple extension of Step 2 in translations $lp(\mathcal{P})$ (resp.$lp^w(\mathcal{P})$):

Step 2 (Auxiliary Predicates): To represent steps, we add the following facts to $lp(\mathcal{P})$

```
time(I) :- #int(I), pl(L), I <= L.
next(I,J) :- #succ(I,J), pl(L), J <= L.
pl(0) v pl(1) v ... v pl(i).
:~ pl(X). [X : l]
```

where i is the upper bound for the plan length of $\mathcal{P}$ and l is a cost level which again has to be chosen lower (resp. higher) than all cost levels of all the other actions in $\mathcal{P}$ in order to achieve (γ), resp. (δ). For (β) we can simply fix l to 1.

Indeed, this simple extension of $lp^w(\mathcal{P})$ achieves optimizations (β), (γ), and (δ) analogously without having to rewrite the problem $\mathcal{P}$ itself beforehand and could be integrated into DLV$^{\mathcal{K}}$ easily.

As for secure planning, the extension again works fine wrt. the interleaved computation implemented in DLV$^{\mathcal{K}}$.

7.1.2 Iterative Search for Shortest Plans

Fixed plan length is obviously a limitation compared with other planners which search for the shortest possibly plan without a given upper bound. Most existing planners iteratively search for longer plans until one is found.

In our language, we consider plan length as part of the input and therefore, plan length is always fixed in some sense. However, if we leave plan length unspecified in the input, incremental search seems the best we can do.

We remark that such incremental search strategies for finding the shortest plan are only applicable for optimization criteria (β) and (δ) from Section 6.4. As soon as other objectives than minimizing plan length alone come into play, iterative plan search is infeasible. For instance, for criterion (γ), i.e. finding the shortest among t he cheapest plans wrt. some other cost criterion we can not proceed iteratively: Unless we know the optimal cost of a cheapest plan beforehand, we need to investigate plans of all possible lengths up to an upper bound.

However, in many cases iterative search makes sense, especially, as mentioned above, if no good upper bound is known. As for our approach, all we can do so far is incrementing plan length after failing to find a plan and restart from scratch.

Possible improvements, for instance by deriving counterexamples from an unsuccessful search for plans of length n which can be exploited to prune the search space for plans of length $n+1$ remain to be further investigated.

We remark that the impact of leaving the plan length unspecified on the complexity of computing optimistic and secure plans has already been discussed in Section 3.3 (cf. proof of Theorem 3.8, p. 56).

7.1.3 Multi-Valued Fluents

Fluent literals in $\mathcal{K}^c$ can only represent Boolean predicates. However, in planning we often have to face state variables which do not only take Boolean values but a range of values from a (finite) domain.

Taking a closer look to our Bridge Crossing Example in Figure 3.1 (see p. 38), one might criticize some deficiencies in the way fluents are represented: For the example above, a person can always be at at most one location (`here, across`) or exactly one person has the lamp. In order to express such settings, multi-valued fluents which are available in many other action languages, e.g. in $\mathcal{C}+$ [GLLT01, LL01, GLL+03], but also $\mathcal{A}_R$ [GKL97] would be desirable where fluents can take a distinct value from a certain range rather than only true or false. Multi-valued (also known as "functional" fluents) fluents have already been introduced in the Situation Calculus, and are also allowed in a functional extension to STRIPS called FSTRIPS [Gef00], and in the currently most wide-spread planning language PDDL, called "functions" there.

Intuitively, we would preferably write something like:

```
caused hasLamp = jack after takeLamp(jack).
```

instead of two causation rules:

```
caused hasLamp(jack) after takeLamp(jack).
caused -hasLamp(P) after takeLamp(jack), person(P), P != jack.
```

as effect of `takeLamp(jack)` in a state where `joe` has the lamp. Here, the fluent `hasLamp` changes its value rather than the Boolean fluent `hasLamp(jack)` becomes true and all other Boolean fluents `hasLamp(`p`)` become false. In $\mathcal{K}^c$ we often use classical negation only to "override" inertia, if fluents are concerned which can only take a distinct value at each point of time. Using this notation, our example in Figure 3.1 could be represented more compact and comprehensive.

We therefore extend the notion of fluent declarations of the form (3.1) in order to allow for *multi-valued fluent declarations* of the form:

$$p(X_1, \ldots, X_n) : range \texttt{ requires } t_1, \ldots, t_m \tag{7.4}$$

where p is the fluent name, $X_1, \ldots, X_n$ are variables and $n \geq 0$ is the arity of p. $t_1, \ldots, t_m$ refer to background predicates, $m \geq 0$, every X_i occurs in $t_1, \ldots, t_m$, and *range* is a unary predicate from the background knowledge. Furthermore, causation rules may contain *multi-valued fluent literals* of the form $\texttt{p(X}_1, \ldots, \texttt{X}_n) = \texttt{V}$ in this extended version, where `V` is a constant or a variable. Note that we do not allow for true negation of these special literals.

The semantics can be defined by viewing multi-valued fluent literals as macros for regular Boolean fluent literals, where we execute the following preprocessing steps:
(i) Each multi-valued fluent declaration

```
fluent:    fl(X̃) : type requires...
```

will be rewritten to a Boolean fluent declaration:

```
fluent:    fl(X̃, Val) requires type(Val)...
```

(ii) Causation rules with multi-valued fluent literals can be naively rewritten as follows: Each causation rule with a multi-valued fluent literal in the head will be transformed as follows:

$$\texttt{caused fl}(\tilde{\texttt{X}}) = \texttt{val} \ \ldots \quad \Leftrightarrow \quad \begin{array}{l} \texttt{caused fl}(\tilde{\texttt{X}}, \texttt{val}) \ \ldots \\ \texttt{caused} \ \texttt{-fl}(\tilde{\texttt{X}}, \texttt{Val1}) \ \texttt{if val} \ \texttt{!=} \ \texttt{Val1}, \ \texttt{type}(\texttt{Val1}) \ \ldots \end{array}$$

where `type` is the range predicate from the fluent declaration of `fl`. A (possibly default negated) multi-valued fluent literal $\texttt{fl}(\tilde{\texttt{X}}) = \texttt{Val}$ in the `if` (resp. `after`) part of a causation rule or executability condition can then be simply rewritten to its Boolean counterpart $\texttt{fl}(\tilde{\texttt{X}}, \texttt{Val})$.

Special attention in this context is needed for the `inertial` and `total` macros which need to be redefined for multi-valued fluents. For a multi-valued fluent `f`, we define:

$$\texttt{inertial f}(\tilde{\texttt{X}}) \ \texttt{if B after A.} \Leftrightarrow \texttt{caused f}(\tilde{\texttt{X}}, \texttt{Val}) \ \texttt{if not} \ \texttt{-f}(\tilde{\texttt{X}}, \texttt{Val}), \texttt{B after f}(\tilde{\texttt{X}}, \texttt{Val}), \ \texttt{A.}$$

For the `total` macro, we now want to totalize a fluent over all possible values of its range. Therefore, we define:

$$\texttt{total f}(\tilde{\texttt{X}}) \ \texttt{if B after A.} \quad \Leftrightarrow \quad \begin{array}{l} \texttt{caused f}(\tilde{\texttt{X}}, \texttt{Val}) \ \texttt{if not} \ \texttt{-f}(\tilde{\texttt{X}}, \texttt{Val}), \ \texttt{B after A.} \\ \texttt{caused} \ \texttt{-f}(\tilde{\texttt{X}}, \texttt{Val}) \ \texttt{if not f}(\tilde{\texttt{X}}, \texttt{Val}), \ \texttt{B after A.} \\ \texttt{forbidden f}(\tilde{\texttt{X}}, \texttt{Val}), \ \texttt{f}(\tilde{\texttt{X}}, \texttt{Val1}), \ \texttt{Val} \ \texttt{!=} \ \texttt{Val1.} \\ \texttt{caused total}_{\texttt{f}}(\tilde{\texttt{X}}) \ \texttt{if f}(\tilde{\texttt{X}}, \texttt{Val}). \\ \texttt{forbidden not total}_{\texttt{f}}(\tilde{\texttt{X}}). \end{array}$$

These rules guess exactly one value for $\texttt{f}(\tilde{\texttt{X}})$, if `B` is satisfied in the current state and `A` is satisfied in the predecessor.

We remark that in these macro translations, we do not yet make use of an important representational advantage of $\mathcal{K}^c$: As stated above, classical negation is only used here to "override" the old fluent value, and the subsequent state, i.e. we do not necessarily want to carry over all negative values of `-fl` to the next state. In $\mathcal{K}^c$'s knowledge state view, states are defined as consistent sets of fluent literals (i.e. the current knowledge of the planning agent about the world) and not as a mapping from fluents to truth values like in other approaches. As for the macros above again "hidden" guesses via default negation can play a role (cf. Section 6.6.2).

Further simplifications of this transformation under certain circumstances remains to be investigated.

7.1.4 Fluent-Dependent Action Costs

A further extension of $\mathcal{K}^c$ are fully dynamic action costs. Recall that plan costs in $\mathcal{K}^c$ are defined in Section 3.2 as the sum of the costs of all actions in the plan (cf. Definition 3.24), where plans are sequences of (sets of) actions. At present, $\mathcal{K}^c$ only allows for action costs with very restricted means of dynamic cost contributions: Action costs may only depend on background knowledge facts or on the time when the action occurs, but not on dynamic fluent values.

However, these current restrictions have a pragmatic reason: Whereas in deterministic planning with complete knowledge each plan corresponds to a unique sequence of states, whenever nondeterminism comes into play, each plan might have several possible trajectories. The current restrictions guarantee unique costs of plans, since different intermediate states, i.e. different fluent values can not influence action costs by definition.

Nevertheless, dynamic costs are an important issue, as shown by the following example.

Example 7.1. For instance, in our Quick Bridge Crossing example (see Section 3.2.3, p. 44) we have no possibility to express that a person gets tired when crossing several times. Recall that each person needs a certain amount of time for crossing the bridge, which is expressed by predicate `walk(X,SX)` as defined in the background knowledge Π_{QBCP}. Now we want to express that each time a person crosses the bridge, he/she becomes slower and needs one minute more for subsequent crosses.

If fluent-dependent costs were allowed, this could for instance be expressed by means of an additional fluent `fatigue(X)` as follows[34]:

```
fluents:   fatigue(X) : #int requires person(X).
actions:   cross(X) requires person(X) costs C
                    where walk(X,SX),
                    fatigue(X) = K,  C = SX + K.
⋮
always:    caused fatigue(X) = K1 after
                  cross(X),  fatigue(X) = K,  K1 = K + 1.
           inertial fatigue(X).
```

A person fatigues from subsequent crosses, so the cost of crossing is dependent on the new multi-valued fluent `fatigue` whose value increases by one after each crossing.

Things get even more involved if fatigue is nondeterministic, i.e. a person might fatigue or not from crossing. However, this could be modeled by a simple modification of the encoding above as follows.

```
always:    caused fatigue(X) = K1 if not fatigue(X) = K after
                  cross(X),  fatigue(X) = K,  K1 = K + 1.
           inertial fatigue(X).
```

Indeed, these two rules model a nondeterministic guess again; nondeterminism via unstratified negation is "hidden" inside an `inertial` macro (cf. Section 6.6.2) in this encoding. For exemplification, we restricted ourselves to the definitions of the effects of action `cross` here, and remark that the declaration and effects of action `crossTogether` would need to be modified analogously. Now that a plan might have different costs in different trajectories, the most important question is what the costs of an optimal plan are? We have opted for a cautious view where we take the *maximum*, i.e. worst cost possible for a plan execution, which we define below. ◇

In a setting with dynamic costs, we first have to modify the definition of action declarations (cf. Definition 3.20, p. 42) such that fluent literals are allowed in the `where` part of the `costs` clause:

Definition 7.1. *A* generalized action declaration *d in $\mathcal{K}^c$ is of the form:*

$$p(X_1,\ldots,X_n)\ \texttt{requires}\ t_1,\ldots,t_m\ \texttt{costs}\ C\ \texttt{where}\ c_1,\ldots,c_k. \tag{7.5}$$

where

[34]Here, we already use the multi-valued fluent notation from Section 7.1.3

(1) $p \in \sigma^{act}$,

(2) $X_1, \ldots, X_n \in \sigma^{var}$ *where* $n \geq 0$ *is the arity of* p,

(3) $t_1, \ldots, t_m$ *are from* $\mathcal{L}_{typ}$ *such that every* X_i *occurs in* $t_1, \ldots, t_m$,

(4) $c_1, \ldots, c_k$ *are from* $\mathcal{L}_{typ} \cup \mathcal{L}_{fl,typ}$.

(5) C *is either an integer constant, a variable from the set of all variables occurring in* $t_1, \ldots, t_m$, $c_1, \ldots, c_k$ *(denoted by* $\sigma^{var}(d)$*), or the distinguished variable* `time`,

(6) $\sigma^{var}(d) \subseteq \sigma^{var} \cup \{\texttt{time}\}$, *and*

(7) `time` *does not occur in* $t_1, \ldots t_m$.

The modification in item *(4)* allows costs to dynamically depend on fluent values. Next, we have to redefine the cost of an action instance wrt. the current state (cf. Definition 3.22, p. 43).

Definition 7.2. *Let* $a = p(x_1, \ldots, x_n)$ *be a legal action instance of a declaration* d *of the form (7.5), let* $i \geq 1$ *be a time point,* s *be a state, and let* θ *be a witness substitution for* a. *such that* $\texttt{time}\,\theta = i$. *Then*

$$cost_\theta(p(x_1, \ldots, x_n), s) = \begin{cases} 0, & \text{if the } \texttt{costs}\text{-part of } d \text{ is empty;} \\ val(C\theta), & \text{if } \{c_1\theta, \ldots, c_k\theta\} \subseteq M \cup s; \\ undefined & \text{otherwise.} \end{cases}$$

where M *is the unique answer set of* Π *and* $val : \sigma^{con} \to \mathbb{N}$ *is defined as the integer value for integer constants and undefined for all non-integer constants.*

We also need to modify the definition of well-definedness wrt. the current state (cf. Definition 3.23, p. 43):

Definition 7.3. *Let* $\mathcal{P} = \langle PD, Q?(l)\rangle$ *be a* $\mathcal{K}^c$ *planning problem. A legal action instance* $a = p(x_1, \ldots, x_n)$ *is* well-defined, *wrt.* $\mathcal{P}$ *if it holds that*

(i) for any time point $1 \leq i \leq l$ *and reachable state* s *in* $\mathcal{P}$, *there is some witness substitution* θ *for* a *such that* $\texttt{time}, \theta = i$ *and* $cost_\theta(a)$ *is defined, and*

(ii) $cost_\theta(a, s) = cost_{\theta'}(a, s)$ *holds for any two witness substitutions* θ, θ' *which coincide on* `time` *and have defined costs.*

For any well-defined a, *its unique cost at time point* $i \geq 1$ *in state* s *is given by* $cost_i(a, s) = cost_\theta(a, s)$ *where* θ *is as in (i).*

Note that here we define well-definedness wrt. to any *reachable* state in $\mathcal{P} = \langle PD, Q?(l)\rangle$, i.e. states reachable from a legal initial state within no more than l steps via legal state transitions. It would not make much sense to claim well-definedness wrt. *any* state, i.e. any consistent set of fluent literals: In this case, for instance, fluent-dependent cost values could never be well-defined since the empty set is a state by our definition but does not allow for a witness substitution such that the costs are defined. Therefore, well-definedness

in this context means that the domain has to satisfy that cost definitions are "reasonable" in that action costs do not depend on possibly undefined fluents. We could further loosen the well-definedness restriction by claiming well-definedness only wrt. states where action a is actually executable. On the other hand, then we should define well-definedness wrt. executable *actions sets* rather than for single actions, since executability might depend on other actions (cf. Definition 3.13).

For instance, generalized action declarations with costs only depending on multi-valued, inertial fluents with a domain $D \subseteq \mathbb{N}$ are generally safe in this context: Whenever costs only depend on such fluents, and the value of all these fluents is defined in the initial state, then a unique cost value, and therefore, well-definedness is guaranteed in any state.

We remark that with the extension to fluent-dependent costs, we could drop the special keyword `time`, without loss of expressive power: A time stamp can easily be encoded as an extra (multi-valued) fluent with initial value 1 which increases by one at each time step. Anyway, we keep `time` here, since it is convenient for problem representation.

Next, we will extend our definition of plan costs from Definition 3.24 (see p. 43) wrt. trajectories:

Definition 7.4. *Let $T = \langle t_1, \ldots, t_l \rangle$ be a trajectory, where s_0 is a legal initial state, and $t_j = \langle s_{j-1}, A_j, s_j \rangle$ is a legal state transition from state s_{j-1} to state s_j executing a set of actions A_j $(j = 1, \ldots, l)$. Then, the cost of T wrt. a $\mathcal{K}^c$ planning domain is defined as*

$$cost(T) = \sum_{j=1}^{l} \left(\sum_{a \in A_j} cost_j(a, s_{j-1}) \right).$$

where $cost_j(a, s_{j-1})$ is the cost of action a wrt. state s_{j-1} at time j according to the `costs`*-part of the resp. action declaration of a.*

Now that action costs depend on the state where they are executed we have to change the definition of plan costs accordingly:

Definition 7.5. *Let $\mathcal{P}$ be a $\mathcal{K}^c$ planning problem. Then, for any plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$, where A_i is the set of actions executed at time i, the cost of P is defined as the maximum cost over all supporting trajectories T constituting a successful execution of P (written $T \models P$), i.e.*

$$cost_{\mathcal{P}}(P) = \max_{T \models P} cost(T).$$

An *optimal optimistic plan* is again an optimistic plan with minimal cost. As for secure planning, we define *optimal secure* plans as plans with minimal costs among all secure plans. Analogously, an *admissible (optimistic/secure)* plan wrt. cost c is an (optimistic/secure) plan with $cost_{\mathcal{P}}(P) \leq c$.

Our cautious definition of plan costs by the maximum costs among the supporting trajectories could be in particular important for an estimation of worst case bounds in presence of uncertainty in critical applications. However, also different measures such as the average costs among the supporting trajectories of a plan, or some other aggregate function over the costs of the corresponding trajectories might be of interest; exploring this in detail requires further work.

As for implementation, our current approach for computing optimal plans outlined above is not feasible any longer wrt. this definition. In order to surmount this, a naive approach would be caching all plans and their maximal costs during answer set computation. However, as there might be an exponential number of plans, caching would be highly inefficient. A better strategy is again "intercepting" candidate answer set generation at the same point where checking plan security is performed now: In order to find the optimal plan, consider only those answer sets representing maximum cost trajectories by checking whether there is a more expensive trajectory for the plan at hand.

7.1.4.1 Impact on Complexity Results

The additional expressivity of full dynamic action costs does not come for free. In the following, we will sketch how far the complexity results from Section 3.3 are affected.

Complexity of checking well-definedness The complexity of checking whether a domain *PD* is well-defined increases from Π_2^P to PSPACE in general (unless plan length is polynomially bounded):

Proposition 7.1. *Given a $\mathcal{K}^c$ planning problem $\mathcal{P} = \langle PD, Q(l)? \rangle$ where $PD = \langle \Pi, \langle D, R \rangle \rangle$ with fluent-dependent action costs and the unique model M of Π, checking well-definedness of a given action declaration d of the form (7.5) wrt. PD and well-definedness of PD are (i) Π_2^P complete if plan length l is bound polynomially and (ii)* PSPACE*-complete in general.*

Proof. (Sketch) We deploy similar considerations as for the proof of Theorem 3.10 (see p. 62), i.e. we will guess an action a plus a state s reachable within (at most) l steps and verify well-definedness. Without loss of generality, we now assume states reachable in *exactly* l steps.

From Theorem 3.3 we already know that for plan length fixed to a constant (or polynomially bounded plan length, respectively) we can guess and check a trajectory, i.e. a reachable state s, in polynomial time. We get membership in Π_2^P for (i) by similar considerations as in the proof of Theorem 3.10, since the additional guess for a reachable state s (by means of an NP-oracle) does not increase overall complexity.

Concerning membership in PSPACE for (ii), we have sketched an algorithm REACH(s_0, s, l) in the proof of Theorem 3.3 verifying for a state s whether it is reachable from an initial state s_0 in l steps using polynomial space.[35] Thus, we can guess and verify a reachable state s in NPSPACE = PSPACE. For this s we can again check well-definedness of any action a in $\Pi_2^P \subseteq$ PSPACE, proving membership in PSPACE of the overall problem.

Hardness of (i) has already been shown in the proof of Theorem 3.10 for action costs which do not depend on the current state. As for hardness of (ii), we will reduce STRIPS planning to a well-definedness check. Here, we can use the reduction from the proof of Theorem 3.3 with a small extension: For the STRIPS goal $g = \gamma$? (ℓ) we add new propositional fluents `nogoal` and `goal` (with declarations having empty `requires`-parts) and causation rules:

[35] Recall that if the plan length is not fixed to a constant or polynomially bounded, we can not simply guess the whole trajectory leading to s as in (i) since the representation of this trajectory might be exponential in (the binary representation of) l.

```
always:    caused goal if γ.
always:    caused nogoal if not goal.
```

Next, we add an action `dummy` with declaration

```
always:    dummy costs 0 where nogoal.
```

Now, whenever there is no reachable goal state within l steps, i.e. no STRIPS plan exists, then the domain is well-defined, whereas well-definedness is violated in any goal state for action `dummy`. This proofs hardness for co-PSPACE = PSPACE. □

Note that analogously, checking well-definedness for propositional domains as described in Corollary 3.11 increases to PSPACE in general and to co-NP for polynomially bounded plan length: We again have to check that there exists no reachable state such that the costs of some action a are undefined.

This result has an impact on the encodings for checking well-definedness in Section 4.3.7 which do not work as proposed any longer. Rather, in order to obtain a logic program for checking well-definedness we would need to combine the program from Section 4.3.7 with the translation $lp(\mathcal{P})$, which (when the goal is left empty) can serve to compute the reachable states.

For the following results, we begin with some preliminary thoughts: First, checking whether a supporting trajectory has maximal cost among all supporting trajectories for plan P is in co-NP since a more expensive trajectory for P can be guessed and checked in polynomial time:

Corollary 7.2. *For plan P and trajectory T, deciding whether $cost(T) = cost(P)$, i.e. whether T has maximal cost among all supporting trajectories for P is in co-*NP.

Hardness can also be shown, by adaptions of the proof of Theorem 3.12. Furthermore, by similar considerations as used in the proof of Theorem 3.13 the following can be shown: Let P be a plan for a $\mathcal{K}^c$ planning problem $\mathcal{P}$ with fluent-dependent action costs, then computing $cost(P)$ is $\mathrm{F}\Delta_2^P$-complete, and computing $cost^*_{\mathcal{P}}$ becomes $\mathrm{F}\Delta_3^P$-complete.

Complexity of admissible/optimal planning As for the complexity of admissible optimistic planning the complexity of admissible planning increases from NP (cf. Theorem 3.12) to Σ_2^P due to the need of a co-NP oracle for checking whether the guessed trajectory is maximal. Consequently, optimal planning rises one step in PH from $\mathrm{F}\Delta_2^P$ to $\mathrm{F}\Delta_3^P$. Again, the hardness proof needs adaptions compared with Theorem 3.12.

Complexity of optimal/admissible secure planning Remarkably, complexity does not change for admissible and optimal secure planning, since, informally, the additional check whether no more expensive trajectory for P exists can be "hidden" inside the secure check: As for admissible secure planning, we remain in Σ_3^P since, if we guess plan P and a corresponding trajectory T, we can use the Π_2^P-oracle in the proof of Theorem 3.12 in order to additionally check whether T has maximal cost among the supporting trajectories, letting only those plans/trajectories pass the secure check for which no cheaper trajectory exists. Analogous considerations apply to secure optimal planning, where complexity for computing a secure optimal plan remains in $\mathrm{F}\Delta_4^P$.

7.2 Implementation and Optimization Issues

7.2.1 Integrated Encodings

The reason why we have chosen an interleaved computation for secure planning is that the integrated encodings for $\mathcal{SC}_1$ (resp. $\mathcal{SC}_2$) discussed in the Chapter 4 are not competitive at the current stage wrt. performance.

This might also hold for the general secure check $lp^{SC}(\mathcal{P}, P)$ described in Section 4.3.3 which is not yet implemented. Moreover, we have experienced that for a wide range of problems $\mathcal{SC}_1$ (resp. $\mathcal{SC}_2$) are sufficient and the additional overhead for the general secure check does not pay off for many problems. In order to fruitfully integrate the general check, probably a structural analysis to decide which approach suits best and further optimizations of the integrated encodings wrt. the concrete planning problem at hand are needed for improving performance.

For further details, we refer to Chapter 8 where we compare the interleaved computation of DLV^K with the integrated encodings for conformant planning wrt. to the integrated encoding for secure check $\mathcal{SC}_1$ outlined in Section 4.3.5.

7.2.2 Relaxed Plans and Goal Regression

A basic idea behind many of the most successful classical planners [BF97, HN01] is the computation of so called "relaxed" plans by ignoring delete-lists (in STRIPS-like settings, cf. Section 2.4) and checking beforehand whether the goal actually can be reached in this relaxed setting, thus pruning the forward search space. Such ideas partly have a practical counterpart in our implementation wrt. to the intelligent grounding methods of the underlying DLV system which prunes the search space beforehand by grounding only DLP rules and constraints which can actually be "reached" starting from the facts. For instance, stratified normal logic programs are solved in DLV by the grounding at once. Sophisticated pruning methods like in classical planners such as in the Graphplan algorithm [BF97] are not yet exploited. However, it is questionable how far such techniques which rely on the STRIPS-style framework can be exploited in the more general framework of action languages at all. Still, for instance efforts to adapt and combine these methods for the planning as satisfiability framework [KS99, Bra01] might apply to the answer set planning as well to some extent.

On the other hand, classical planners also often use goal regression in order to search backwards from the goal. A method for "emulating" regression search for queries in LPs, so-called Magic Sets [BMSU86, Ull89], is currently being implemented in DLV. Investigating the applicability to our planning encodings or respective extensions remains part of further research.

Chapter 8

Experimental Evaluation

In this chapter, we give an overview on the experimental evaluation of the described methods which has been conducted in the course of this work.

We will report on several experiments with our implemented system DLV$^{\mathcal{K}}$. Here, we first focus on conformant planning under incomplete knowledge and/or nondeterministic action effects. To this end, we have compared the system with other conformant planning systems on elaborations of the "Bomb in the Toilet" domain from Section 6.3.1. The original data for this experiments stems from our article on the DLV$^{\mathcal{K}}$ system [EFL$^+$03a].

Next, we will draw our attention to planning under action costs. Here we especially focus on problems which can not be represented in other formalisms or existing systems offhand. To get a picture of the system performance on optimal planning problems, we have tested our elaborations of the Blocks World domain on several instances wrt. the optimizations (α) – (δ) from Section 6.4. We have also conducted a series of experiments on the route planning problems with exceptional time-dependent costs from the Traveling Salesperson domain mentioned in Section 6.5. A more detailed discussion of these experiments can be found in our article on planning under action costs [EFL$^+$03c].

Remark 8.1. *Note that, apart from experimenting with* DLV$^{\mathcal{K}}$ *and its underlying Answer Set solver* DLV*, we have also experimented with alternative translations for* SMODELS *and* GNT*. We refrain from detailed discussion here, since the (i) translations mentioned in this thesis are optimized for* DLV *and* SMODELS *performance was worse than* DLV *(e.g., around factor* 10 *for the tested TSP instances with the alternative translation mentioned in Chapter 4) and (ii) there is no integrated planning front-end available for* SMODELS *or* GNT *providing a high-level planning language. Nevertheless, as we have shown, our approach can, with minor modifications, be adopted in a planning system based on these systems or any other efficient Answer Set solver.*

In a further part of this chapter we report on the experimental evaluation of our method for automatically generating integrated encodings of separate "guess" and "check" programs, which was introduced in Section 4.2.3. Here, we will first report on general experiments using the approach on some well-studied Σ_2^P-complete problems compared with existing ad

hoc ASP encodings. Namely, we have tested respective encodings for QBFs and Strategic Companies (cf. Sections 4.2.5.1 and 4.2.5.2). Afterwards, we report on the application of the method to integrated computation of secure plans (cf. Section 4.3.5) and compare it against the interleaved computation implemented in DLV$^{\mathcal{K}}$. These results have partly been reported in [EP03].

We will conclude with a short summary and discussion of the results.

8.1 Overview of Compared Systems

For our experiments we considered the following systems.

CCALC

The *Causal Calculator* (CCALC) is a model checker for the languages of causal theories [MT97]. It translates programs in the action language $\mathcal{C}+$ into the language of causal theories which are in turn transformed into SAT problems using literal completion as described in [MT98]. This approach is based on Satisfiability Planning [KS92], where planning problems are reduced to SAT problems which are then solved by means of an efficient SAT solver. The current version of CCALC uses *mChaff* [MMZ+01] as its default SAT solver, but also other solvers such as like *SATO* [Zha97] or *relsat* [BS97] can be used. Minimal length plans are generated iteratively increasing the plan length up to an upper bound. CCALC is written in Prolog.

Though its input language allows nondeterminism in the initial state and also nondeterministic action effects, CCALC as such is not capable of conformant planning and only computes "optimistic plans" (according to DLV$^{\mathcal{K}}$ terminology). Plan length is fixed, and both sequential and concurrent planning are supported. Still, we mention the system here, due to its close relationship to our approach. A conformant planner based on CCALC is described next.

CCALC is written in Prolog. For our tests, we used version 2.04b of CCALC which we obtained from `<URL:http://www.cs.utexas.edu/users/tag/cc/>` and a trial version of SICStus Prolog 3.9.1.;

CPLAN

Introduced in [Giu00, FG00, CGT02], *CPLAN* is a conformant planner based on CCALC and the $\mathcal{C}$ action language [GL98b, Lif99a, LT99]. This language is similar to $\mathcal{K}$ in many respects, but close to classical logic, while $\mathcal{K}$ is more "logic programming oriented" by the use default negation (see [EFL+03b] for further discussion). CPLAN uses CCALC only to generate a SAT instance and replaces the optional SAT-solvers used by CCALC with an own procedure that extracts conformant plans from these SAT instances. CPLAN implements full conformant planning and supports the computation of both minimal length plans as well as plans of fixed length, by incrementing plan length from a given lower bound until a plan is found or a given upper bound is reached. We set the upper and lower bound equal to the minimal plan length of the specific problems for our experiments to be comparable with DLV$^{\mathcal{K}}$. Sequential and concurrent planning are possible; nondeterminism is allowed in the initial state as well as for action effects.

For our tests, we used CPLAN 1.3.0, which is available at <URL:http://frege.mrg.dist.unige.it/~otto/cplan.html>, together with CCALC 1.90 to produce the input for CPLAN. We had to use the older version of CCALC, since CPLAN is not longer maintained and does not work together with the more current version CCALC 2.0.

CMBP

The *Conformant Model Based Planner* [CR00] is based on the model checking paradigm as well and relies on symbolic techniques such as BDDs. CMBP only allows sequential planning. Its input language is an extension of $\mathcal{AR}$ [GKL97]. Unlike action languages such as $\mathcal{C}$ or $\mathcal{K}$, this language only supports propositional actions. Nondeterminism is allowed in the initial state and for action effects. The length of computed plans is always minimal, but the user has to declare an upper bound using command-line option `-pl`. If `-pl` is set equal to the minimal plan length for the specific problem, this can be used to fix the plan length in advance. We used this method to be comparable with DLV$^{\mathcal{K}}$ which currently can only deal with fixed plan length.

For our tests, we used CMBP 1.0, available at <URL:http://sra.itc.it/people/roveri/cmbp/>.

We remark that in between, with the MBP [BCPT01] system a more powerful extension of CMBP is available. MBP extends CMBP by allowing for conditional planning and furthermore offers a very expressive goal description language. Furthermore, as opposed to CMBP it does not use $\mathcal{AR}$ as input language but a nondeterministic extension of PDDL which is not restricted to propositional actions. We refer to Chapter 10 for further discussion.

GPT

The *General Planning Tool* [BG00] employs heuristic search techniques like A^* to search the belief space. Its input language is a subset of PDDL. Nondeterminism is allowed in the initial state as well as for action effects. GPT only supports sequential planning and calculates plans of minimal length.

We used version GPT 1.14 obtained from <URL:http://www.cs.ucla.edu/~bonet/software/>.

SGP

In addition to conformant planning, *Sensory Graphplan* [WAS98] can also deal with sensing actions. SGP is an extension of the Graphplan algorithm [BF97]. Its input language is an extension of PDDL [GHK+98]. Nondeterminism is allowed only in the initial state. The program always calculates plans of minimal length.[36] SGP does not support sequential planning, but computes concurrent plans automatically recognizing mutually exclusive actions. That means, minimal length plans in terms of SGP are not plans with a minimal number of actions but with a minimal number of steps needed. At each step an arbitrary number of parallel actions are allowed, as long as the preconditions or effects are not mutually exclusive which is automatically detected by the algorithm.

[36] SGP comprises the functionality of another system by Smith and Weld called CGP (Conformant Graphplan, [SW98]), but is slower in general. As CGP is no longer maintained and not available online, we nevertheless decided to choose SGP for our experiments.

SGP is written in LISP and available at <URL:http://www.cs.washington.edu/ai/sgp.html>. For our tests, we used a trial version of Allegro Common Lisp 6.0.

8.1.0.1 Specific System Features

We would also like to point out further specific features of some of these special purpose planning systems:

- SGP automatically recognizes mutually exclusive actions in concurrent plans. It is possible to encode concurrent plans in DLV$^{\mathcal{K}}$ by explicitly describing the mutually exclusive actions, as done in our encodings of the "Bomb in the Toilet" benchmark problems for multiple toilets (see Section 8.2.1). However, the language $\mathcal{K}$ is more complex than PDDL, which makes automatic recognition of possible conflicts of actions much harder in our framework. On the other hand, our notions of executability and nonexecutability allow more flexible encodings of parallel actions than SGP.
- GPT and SGP always compute minimal plans, which is not directly possible in the current version of DLV$^{\mathcal{K}}$ (cf. Sections 6.4 and 7.1.2 for further discussion).
- CMBP and CPLAN optionally compute minimal plans, where the user may specify upper and/or lower bounds for the plan length.

Table 8.1.0.1 provides a comparison of DLV$^{\mathcal{K}}$ and all the systems introduced above. Note that CCALC is not capable of conformant planning, and thus we cannot use it on the respective benchmark problems, but we will reconsider it later for the benchmarks on optimal planning.

	DLV$^{\mathcal{K}}$	CCALC	CPLAN	CMBP	SGP	GPT
Input Language	$\mathcal{K}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{AR}$	PDDL	PDDL
Sequential plans	yes	yes	yes	yes	no	yes
Concurrent plans	yes	yes	yes	no	yes	no
Conformant plans	yes	no	yes	yes	yes	yes
Minimal plan length	no	no	yes	yes	yes	yes
Fixed plan length	yes	yes	yes	no[a]	no	no

[a] An upper bound can be specified, but computed plans are always minimal.

Table 8.1: Overview of System Features

8.1.1 Test Environment

All tests in Sections 8.2 and 8.3 were performed on an Intel Pentium III 733MHz machine with 256MB of main memory running SuSE Linux. The performance results for these tests reflect the state of DLV$^{\mathcal{K}}$ in early 2002 for the results in Section 8.2, and early 2003 for the results in Section 8.3. Note that for CCALC the results include startup time for SICStus Prolog, and loading CCALC while for SGP startup time for Allegro Common Lisp is included (cf. [EFL$^+$03a, EFL$^+$03c] for details).

The more recent tests in Section 8.4 were performed on an AMD Athlon 1200MHz machine with 256MB of main memory running SuSE Linux using DLV$^{\mathcal{K}}$ (or DLV for the respective manual encodings) in a version reflecting the development state of July 2003.

8.2 Conformant Planning

In the following, we will compare DLV$^{\mathcal{K}}$ with several state-of-the-art conformant planning systems, and report about experimental results about the performance of the system in the "Bomb in the Toilet" domain. The results presented here are mainly intended to give a momentary view. To that end, we present extensive benchmark results, and also compare the expressive power and flexibility of the various systems. The underlying data stems from experiments which date back to early 2002 and we emphasize that both DLV$^{\mathcal{K}}$ and other planners have advanced since then. We refer to Section 10.5 for a discussion of latest developments.

8.2.1 Benchmark Problem and Encodings – Bomb in the Toilet

To show the capabilities of DLV$^{\mathcal{K}}$ on planning under incomplete information, and in particular conformant planning, we have chosen different variations of the well-known "Bomb in the Toilet" problem [McD87]. Here, we employ a naming convention from Section 6.3.1.

The respective planning domain comprises actions with nondeterministic effects, the initial state is incomplete and, in more elaborated versions, several actions are available that can be done in parallel. Furthermore it allows for good comparison between knowledge-based and world state encodings wrt. our action language $\mathcal{K}$.

As far as possible, we used the original encodings which come along with the distributions of the compared systems.

CCALC/CPLAN: CCALC is not capable of conformant planning (while CPLAN proved very slow on deterministic domains in preliminary experiments). So, we have used CPLAN for the "Bomb in the Toilet" problems with slight modifications of the $\mathcal{C}$ encodings provided with the CPLAN distribution.

CMBP: For CMBP, we used the "Bomb in the Toilet" encodings which are included in the distribution. BMTUC(p,t) is not included, but only a trivial modification of BMTC(p,t) is needed to obtain an encoding for BMTUC.

GPT: The distribution of GPT provides encodings for various "Bomb in the Toilet" problems; BMTUC(p,t) was not included, but the respective extension of BTUC(p) is trivial.

SGP: For SGP we used the bomb in toilet encodings coming with the distribution. BTUC(p) and BMTUC(p,t) cannot be encoded in SGP which only allows nondeterminism in the initial state. SGP generates concurrent plans (in fact, serializable concurrent plans), so we did not compare the sequential versions of BT(p) and BMTC(p,t).

DLV$^{\mathcal{K}}$: We have tested the "Bomb in the Toilet" problems for the two different encodings introduced in Sections 6.3.1 and 6.3.2. The first one, labeled *ws* in the results, mimics world-state planning as in Section 6.3.1, in which the different completions of the states ("totalizations") to world-states are considered. The second one, labeled *ks*, uses the power of knowledge-state planning by the improved encodings of Section 6.3.2; it does not complete the states right away, but leaves the value of unknown fluents open in accordance with the real knowledge of the planning agent about the state of affairs. In both encodings, we first consider concurrent actions as well as sequential planning.

As discussed in Section 6.3.2, thanks to the knowledge-state representation, the domains become deterministic and have unique initial states, so the security check is trivial and negligible for timing since optimistic and secure planning coincide for these encodings.

On the other hand, the world-state encodings of "Bomb in the Toilet" are clearly not deterministic, so the security check is responsible for a considerable portion of the timings. We have discussed the applicability of DLV$^{\mathcal{K}}$'s default secure check $\mathcal{SC}_1$ for these encodings in Section 6.3.1.

8.2.2 Results and Discussion – Bomb in the Toilet

Tables 8.2.2.3–8.2.2.3 show the results for the various "Bomb in the Toilet" problems. The minimal plan length is reported in the second column of each table. Run-times exceeding 1200 CPU seconds were omitted, which is indicated by a dash in the tables.

In this section, we compare the various systems in terms of representation capabilities and run-time benchmarks.

8.2.2.1 Performance

Under the world-state encodings of the different "Bomb in the Toilet" instances, DLV$^{\mathcal{K}}$ is not competitive except for BT(p) with concurrent dunks, where plan length is always 1, and BMTC(p). This indicates that DLV$^{\mathcal{K}}$'s performance is quite sensitive to (increasing) plan length, especially for sequential planning. Still, DLV$^{\mathcal{K}}$ outperforms SGP, a special purpose planning system, on all comparable instances, and also CPLAN (which is the system most comparable to DLV$^{\mathcal{K}}$ in terms of expressiveness and similar in nature) seems to be within reach.

Under the knowledge-state encodings, DLV$^{\mathcal{K}}$ outperforms its competitors in many of the chosen examples. The sensitivity to increasing plan length/search space can, however, also partly be observed here, where execution times seem to grow drastically from one instance to the next. This can be partly explained by the general heuristics of the underlying DLV system, which which might not scale up well in some cases. For instance, DLV as a general purpose problem solver does not include special heuristics towards plan search. In particular, during the answer set generation process, no distinction is made between actions and fluents, which might be useful for planning tasks to control the generation of answer sets resp. plans; this may be part of further investigations.

8.2.2.2 Effect of Concurrent Actions and Knowledge-State Encodings

Once we also consider concurrent actions (which are not supported by GPT and CMBP), DLV$^{\mathcal{K}}$ performs better than CPLAN on some larger instances of BMTC(p,t) and BMTUC(p,t) (see Tables 8.2.2.3 and 8.2.2.3).

Using the expressive power of default negation to express unknown fluents with the knowledge-state encodings of "Bomb in the Toilet" in $\mathcal{K}$ pays off well: DLV$^{\mathcal{K}}$ outperforms all other systems, including the special purpose conformant planners GPT and CMBP, except on sequential BMTC(p,t) and BMTUC(p,t) with more than two toilets (see Tables 8.2.2.3 and 8.2.2.3), where CMBP is fastest.

8.2.2.3 Summary of Experimental Results

Overall, the results indicate that DLV$^{\mathcal{K}}$ is competitive with state of the art conformant planners, especially when exploiting the $\mathcal{K}$ language features in terms of knowledge-state problem encodings. Recall, however, that some of the systems compute minimal plans, which is (currently) not supported by DLV$^{\mathcal{K}}$. The comparison of DLV$^{\mathcal{K}}$ to CCALC/CPLAN is particularly relevant, since these systems are closest in spirit to DLV$^{\mathcal{K}}$. As we can see, the advanced features of knowledge-state encoding lead to significant performance improvements. However, as discussed in Chapter 6 such knowledge state encodings which exploit the structure of the problem can not always be applied, especially if actions have conditional effects, which is not the case for the "Bomb in the Toilet" problems.

8.3 Optimal Planning

As for planning with action costs, it is harder to find good comparison data on our approach, since most other systems do not allow for optimal plan computation in our sense. Whereas in the International Planning Competition [ea02] numerical domains where some cost function has to be optimized are part of the competition and plan length is a measure for plan quality, optimal solutions are not required, which are much harder to find in general. Problem sizes there include plans with hundreds of steps and objects, which is beyond the capacity of our current approach. In order to emphasize the features of our approach, we therefore decided to test two of the domains introduced in Chapter 6. Namely, we have tested elaborations of Blocks World under various optimization criteria and TSP with exceptional costs which are not easily expressible in other planners. We present some encouraging experimental results for planning with action costs in these two domains.

Where possible, we also report results for CCALC, due to its comparable capabilities of expressing resource constraints and its similar input language $\mathcal{C}+$. We refrained from comparing the other systems above which are mainly tailored for conformant planning, but for instance showed bad performance on tests with many action instances such as in the Blocks World instances which we investigate here.

In the tables below, run-times exceeding 4000 CPU seconds were omitted, which is indicated by a dash again.

BT(p)	steps	DLV$^{\mathcal{K}}$		CPLAN	SGP
		ws	*ks*		
BT(2)	1	0.01s	0.01s	1.38s	0.69s
BT(3)	1	0.02s	0.01s	1.38s	0.80s
BT(4)	1	0.01s	0.01s	1.39s	0.95s
BT(5)	1	0.02s	0.01s	1.42s	1.21s
BT(6)	1	0.02s	0.01s	1.47s	1.55s
BT(7)	1	0.02s	0.01s	1.56s	2.00s
BT(8)	1	0.02s	0.01s	1.79s	2.56s
BT(9)	1	0.01s	0.02s	2.29s	3.32s
BT(10)	1	0.02s	0.02s	3.41s	4.27s
BT(11)	1	0.02s	0.02s	6.04s	5.34s
BT(12)	1	0.02s	0.02s	11.98s	6.66s
BT(13)	1	0.03s	0.02s	25.28s	8.16s
BT(14)	1	0.03s	0.01s	57.71s	9.98s
BT(15)	1	0.03s	0.01s	127.75s	12.11s
BT(16)	1	0.03s	0.01s	294.44s	14.57s
BT(17)	1	0.03s	0.02s	678.19s	17.43s
BT(18)	1	0.03s	0.02s	-	20.74s
BT(19)	1	0.03s	0.02s	-	24.47s
BT(20)	1	0.04s	0.02s	-	28.78s

Table 8.2: Experimental results for BT(p), concurrent dunks

BT(p)	steps	DLV$^{\mathcal{K}}$		CPLAN	CMBP	GPT
		ws	*ks*			
BT(2)	2	0.02s	0.02s	1.37s	0.03s	0.56s
BT(3)	3	0.03s	0.02s	1.39s	0.04s	0.55s
BT(4)	4	0.11s	0.02s	1.39s	0.04s	0.61s
BT(5)	5	1.50s	0.03s	1.45s	0.04s	0.61s
BT(6)	6	28.78s	0.03s	1.81s	0.04s	0.63s
BT(7)	7	593.15s	0.03s	5.12s	0.05s	0.67s
BT(8)	8	-	0.05s	65.85s	0.06s	0.68s
BT(9)	9	-	0.06s	-	0.07s	0.78s
BT(10)	10	-	0.08s	-	0.10s	0.95s
BT(11)	11	-	0.10s	-	0.19s	1.27s
BT(12)	12	-	0.13s	-	0.39s	2.12s
BT(13)	13	-	0.16s	-	0.82s	3.89s
BT(14)	14	-	0.21s	-	1.76s	8.87s
BT(15)	15	-	0.28s	-	4.00s	19.13s
BT(16)	16	-	0.35s	-	8.82s	42.17s
BT(17)	17	-	0.47s	-	19.03s	93.69s
BT(18)	18	-	0.61s	-	38.95s	208.00s
BT(19)	19	-	0.78s	-	91.89s	496.95s
BT(20)	20	-	0.98s	-	199.63s	546.43s

Table 8.3: Experimental results for BT(p) sequential

BTC(p)	steps	DLV$^{\mathcal{K}}$		CPLAN	CMBP	GPT	SGP
		ws	*ks*				
BTC(2)	3	0.02s	0.01s	1.37s	0.04s	0.59s	0.92s
BTC(3)	5	0.08s	0.02s	1.39s	0.04s	0.60s	3.30s
BTC(4)	7	1.56s	0.02s	1.39s	0.05s	0.60s	191.60s
BTC(5)	9	36.28s	0.03s	2.36s	0.05s	0.62s	-
BTC(6)	11	-	0.04s	28.95s	0.06s	0.66s	-
BTC(7)	13	-	0.06s	178.97s	0.07s	0.68s	-
BTC(8)	15	-	0.08s	-	0.12s	0.74s	-
BTC(9)	17	-	0.11s	-	0.21s	0.81s	-
BTC(10)	19	-	0.14s	-	0.39s	1.04s	-
BTC(11)	21	-	0.20s	-	0.81s	1.48s	-
BTC(12)	23	-	0.26s	-	1.72s	2.51s	-
BTC(13)	25	-	0.34s	-	3.79s	4.68s	-
BTC(14)	27	-	0.45s	-	8.82s	10.84s	-
BTC(15)	29	-	0.58s	-	16.92s	23.31s	-
BTC(16)	31	-	0.74s	-	42.92s	51.40s	-
BTC(17)	33	-	0.94s	-	92.03s	114.21s	-
BTC(18)	35	-	1.17s	-	197.85s	273.25s	-
BTC(19)	38	-	1.46s	-	-	374.00s	-
BTC(20)	39	-	1.80s	-	-	-	-

Table 8.4: Experimental results for BTC(p)

BTUC(p)	steps	DLV$^{\mathcal{K}}$		CPLAN	CMBP	GPT
		ws	*ks*			
BTUC(2)	3	0.03s	0.02s	1.35s	0.03s	0.59s
BTUC(3)	5	0.61s	0.02s	1.45s	0.04s	0.60s
BTUC(4)	7	87.54s	0.03s	1.93s	0.04s	0.61s
BTUC(5)	9	-	0.03s	2.48s	0.06s	0.66s
BTUC(6)	11	-	0.04s	-	0.06s	0.65s
BTUC(7)	13	-	0.05s	51.72s	0.07s	0.74s
BTUC(8)	15	-	0.08s	-	0.12s	0.75s
BTUC(9)	17	-	0.10s	-	0.20s	0.88s
BTUC(10)	19	-	0.14s	-	0.39s	1.18s
BTUC(11)	21	-	0.19s	-	0.80s	1.81s
BTUC(12)	23	-	0.25s	-	1.72s	3.18s
BTUC(13)	25	-	0.33s	-	3.79s	6.42s
BTUC(14)	27	-	0.43s	-	8.81s	14.43s
BTUC(15)	29	-	0.55s	-	16.94s	32.25s
BTUC(16)	31	-	0.71s	-	42.93s	71.10s
BTUC(17)	33	-	0.90s	-	92.02s	159.53s
BTUC(18)	35	-	1.15s	-	197.84s	368.12s
BTUC(19)	38	-	1.41s	-	-	-
BTUC(20)	39	-	1.74s	-	-	-

Table 8.5: Experimental results for BTUC(p)

BMTC(p,t)	steps	DLV$^{\mathcal{K}}$		CPLAN	SGP
		ws	*ks*		
BMTC(2, 2)	1	0.02s	0.01s	1.41s	0.95s
BMTC(3, 2)	3	0.04s	0.02s	1.50s	3.40s
BMTC(4, 2)	3	0.11s	0.03s	1.72s	7.17s
BMTC(5, 2)	5	2.79s	0.04s	3.37s	-
BMTC(6, 2)	5	37.04s	0.07s	13.04s	-
BMTC(7, 2)	7	-	0.52s	71.50s	-
BMTC(8, 2)	7	-	10.66s	-	-
BMTC(9, 2)	9	-	206.27s	-	-
BMTC(10, 2)	9	-	-	-	-
BMTC(2, 3)	1	0.02s	0.02s	1.62s	1.15s
BMTC(3, 3)	1	0.02s	0.02s	2.31s	1.76s
BMTC(4, 3)	3	0.08s	0.03s	4.81s	15.01s
BMTC(5, 3)	3	0.35s	0.03s	13.55s	76.28s
BMTC(6, 3)	3	17.81s	0.06s	43.34s	592.41s
BMTC(7, 3)	5	223.31s	0.13s	210.71s	-
BMTC(8, 3)	5	-	0.74s	417.62s	-
BMTC(9, 3)	5	-	5.90s	-	-
BMTC(10, 3)	7	-	389.08s	-	-
BMTC(2, 4)	1	0.02s	0.02s	2.89s	1.52s
BMTC(3, 4)	1	0.02s	0.02s	9.19s	2.34s
BMTC(4, 4)	1	0.03s	0.02s	37.55s	3.71s
BMTC(5, 4)	3	0.18s	0.04s	158.74s	372.74s
BMTC(6, 4)	3	5.29s	0.05s	571.77s	-
BMTC(7, 4)	3	61.73s	0.09s	-	-
BMTC(8, 4)	3	668.74s	0.41s	-	-
BMTC(9, 4)	5	-	1.06s	-	-
BMTC(10, 4)	5	-	12.14s	-	-

Table 8.6: Experimental results for BMTC(p), conc. dunks

BMTC(p,t)	steps	DLV$^{\mathcal{K}}$		CPLAN	CMBP	GPT
		ws	*ks*			
BMTC(2, 2)	2	0.02s	0.02s	1.41s	0.04s	0.76s
BMTC(3, 2)	4	0.07s	0.02s	1.50s	0.05s	0.78s
BMTC(4, 2)	6	2.47s	0.04s	1.64s	0.06s	0.81s
BMTC(5, 2)	8	208.52s	0.05s	2.66s	0.06s	0.82s
BMTC(6, 2)	10	-	0.07s	32.77s	0.09s	0.86s
BMTC(7, 2)	12	-	0.10s	12.46s	0.12s	0.96s
BMTC(8, 2)	14	-	0.13s	-	0.23s	1.11s
BMTC(9, 2)	16	-	0.20s	-	0.48s	1.48s
BMTC(10, 2)	18	-	0.28s	-	0.96s	2.26s
BMTC(2, 3)	2	0.02s	0.02s	1.50s	0.04s	0.76s
BMTC(3, 3)	3	0.03s	0.02s	1.85s	0.04s	0.81s
BMTC(4, 3)	5	1.84s	0.03s	2.86s	0.06s	0.84s
BMTC(5, 3)	7	291.24s	0.06s	5.92s	0.09s	0.90s
BMTC(6, 3)	9	-	0.09s	14.50s	0.14s	0.99s
BMTC(7, 3)	11	-	0.25s	40.41s	0.30s	1.17s
BMTC(8, 3)	13	-	15.42s	-	0.62s	1.66s
BMTC(9, 3)	15	-	-	-	1.44s	2.79s
BMTC(10, 3)	17	-	-	-	3.31s	5.64s
BMTC(2, 4)	2	0.02s	0.02s	2.02s	0.04s	0.81s
BMTC(3, 4)	3	0.41s	0.02s	3.67s	0.05s	0.83s
BMTC(4, 4)	4	0.60s	0.03s	9.03s	0.07s	0.92s
BMTC(5, 4)	6	149.65s	0.06s	30.55s	0.13s	1.01s
BMTC(6, 4)	8	-	0.10s	-	0.23s	1.27s
BMTC(7, 4)	10	-	0.15s	199.73s	0.51s	1.85s
BMTC(8, 4)	12	-	0.47s	-	1.13s	3.34s
BMTC(9, 4)	14	-	67.07s	-	2.94s	7.18s
BMTC(10, 4)	16	-	-	-	6.38s	17.34s

Table 8.7: Experimental results for BMTC(p) sequential

BMTUC(p,t)	steps	DLV$^{\mathcal{K}}$ *ws*	DLV$^{\mathcal{K}}$ *ks*	CPLAN
BMTUC(2, 2)	1	0.02s	0.02s	1.40s
BMTUC(3, 2)	3	0.11s	0.03s	2.06s
BMTUC(4, 2)	3	7.39s	0.03s	3.54s
BMTUC(5, 2)	5	-	0.04s	8.18s
BMTUC(6, 2)	5	-	0.07s	787.58s
BMTUC(7, 2)	7	-	0.80s	-
BMTUC(8, 2)	7	-	23.57s	-
BMTUC(9, 2)	9	-	818.23s	-
BMTUC(10, 2)	9	-	-	-
BMTUC(2, 3)	1	0.02s	0.02s	1.55s
BMTUC(3, 3)	1	0.02s	0.02s	10.27s
BMTUC(4, 3)	3	0.28s	0.03s	41.03s
BMTUC(5, 3)	3	34.09s	0.03s	181.45s
BMTUC(6, 3)	3	-	0.05s	600.66s
BMTUC(7, 3)	5	-	0.10s	-
BMTUC(8, 3)	5	-	0.74s	-
BMTUC(9, 3)	5	-	9.55s	-
BMTUC(10, 3)	7	-	693.99s	-
BMTUC(2, 4)	1	0.02s	0.02s	2.54s
BMTUC(3, 4)	1	0.02s	0.02s	119.18s
BMTUC(4, 4)	1	0.03s	0.02s	582.84s
BMTUC(5, 4)	3	0.84s	0.04s	-
BMTUC(6, 4)	3	748.90s	0.05s	-
BMTUC(7, 4)	3	-	0.08s	-
BMTUC(8, 4)	3	-	0.55s	-
BMTUC(9, 4)	5	-	0.98s	-
BMTUC(10, 4)	5	-	17.89s	-

Table 8.8: Experimental results for BMTUC(p), conc. dunks

BMTUC(p,t)	steps	DLV$^{\mathcal{K}}$ *ws*	DLV$^{\mathcal{K}}$ *ks*	CPLAN	CMBP	GPT
BMTUC(2, 2)	2	0.02s	0.02s	1.39s	0.04s	0.78s
BMTUC(3, 2)	4	0.52s	0.02s	1.96s	0.04s	0.80s
BMTUC(4, 2)	6	264.20s	0.04s	3.37s	0.05s	0.81s
BMTUC(5, 2)	8	-	0.05s	361.64s	0.06s	0.85s
BMTUC(6, 2)	10	-	0.07s	-	0.08s	0.92s
BMTUC(7, 2)	12	-	0.10s	-	0.12s	1.04s
BMTUC(8, 2)	14	-	0.14s	-	0.23s	1.34s
BMTUC(9, 2)	16	-	0.21s	-	0.47s	2.00s
BMTUC(10, 2)	18	-	0.27s	-	0.96s	3.71s
BMTUC(2, 3)	2	0.02s	0.02s	1.49s	0.04s	0.79s
BMTUC(3, 3)	3	0.04s	0.03s	6.47s	0.05s	0.81s
BMTUC(4, 3)	5	71.03s	0.04s	22.07s	0.06s	0.86s
BMTUC(5, 3)	7	-	0.05s	150.72s	0.09s	0.98s
BMTUC(6, 3)	9	-	0.08s	-	0.14s	1.19s
BMTUC(7, 3)	11	-	0.21s	-	0.29s	1.74s
BMTUC(8, 3)	13	-	13.39s	-	0.61s	3.15s
BMTUC(9, 3)	15	-	-	-	1.45s	6.69s
BMTUC(10, 3)	17	-	-	-	3.31s	15.57s
BMTUC(2, 4)	2	0.01s	0.02s	1.93s	0.04s	0.79s
BMTUC(3, 4)	3	0.78s	0.02s	41.70s	0.05s	0.86s
BMTUC(4, 4)	4	5.81s	0.04s	182.92s	0.07s	0.97s
BMTUC(5, 4)	6	-	0.06s	837.33s	0.12s	1.33s
BMTUC(6, 4)	8	-	0.09s	-	0.23s	2.23s
BMTUC(7, 4)	10	-	0.13s	-	0.51s	4.79s
BMTUC(8, 4)	12	-	0.42s	-	1.13s	11.37s
BMTUC(9, 4)	14	-	64.02s	-	2.94s	28.07s
BMTUC(10, 4)	16	-	-	-	6.37s	68.26s

Table 8.9: Experimental results for BMTUC(p) sequential

8.3.1 Benchmark Problem and Encodings – Blocks World

We used encodings taken from [LL01] for parallel Blocks World adapted for CCALC 2.0. These encodings are included in the current download version of the system. For sequential Blocks World we adapted the encodings by adding the $\mathcal{C}+$ command "`noConcurrency`." which resembles the respective $\mathcal{K}$ command.

Tables 8.10–8.13 show the results for our different Blocks World encodings in Sections 6.2 and 6.4 on several configurations: P0 denotes our simple instance from Figure 6.2, while P1–P5 are instances used in previous work [EFL+03a, Erd99].

Table 8.10 shows the results for finding a shortest sequential plan. The second and third column show the number of blocks and the length of a shortest plan (i.e., the least number of moves) solving the respective Blocks World instance. The execution time for solving the problem using the shortest-plan encoding $\mathcal{P}_\beta$ in Section 6.4 is shown in column five, using the upper bound shown in the fourth column on the plan length. Column six shows the execution time for finding the shortest plan in an incremental plan length search starting from 0, similar to the method used for CCALC. The last column shows the results for CCALC.

Table 8.11 shows the execution times for parallel blocks world with fixed plan length where the number of moves is minimized, i.e. problem (α) in Section 4.2.2. We used the encoding in Figure 6.1, which generates parallel serializable plans. As CCALC does not allow for optimizing other criteria than plan length, we only have results for $\texttt{DLV}^{\mathcal{K}}$ here.

Next, Table 8.12 shows some results for finding a shortest parallel plan, i.e. problem (β) in Section 4.2.2. First, the minimal possible number of steps is given. We processed each instance (i) using the encoding $\mathcal{P}_\beta$ from Section 6.4, (ii) without costs by iteratively increasing the plan length and (iii) using CCALC, by iteratively increasing the plan length until a plan is found. For every result, the number of moves of the first plan computed is reported separately.

Finally, Table 8.13 shows the results for the combined optimizations (γ) and (δ) for parallel Blocks World as outlined in Section 6.4. The second column again contains the upper bound for the plan length of the respective instance. The following three columns present the results on finding a shortest among the cheapest plans, i.e. problem (γ) in Section 4.2.2:

$\texttt{DLV}^{\mathcal{K}}$ refers to the results for our combined minimal encoding $\mathcal{P}_\gamma$ and as described in Section 6.4;

$\texttt{DLV}^{\mathcal{K}}_{inc}$ refers to the results for incrementally searching for the shortest among the cheapest

Problem	#blocks	min. #moves (=#steps)	upper bound #steps	$\texttt{DLV}^{\mathcal{K}}$	$\texttt{DLV}^{\mathcal{K}}_{inc}$	CCALC
P0	6	5	6	0.48s	0.29s	4.65s
P1	4	4	4	0.05s	0.08s	3.02s
P2	5	6	7	0.24s	0.27s	4.02s
P3	8	8	10	25.32s	2.33s	10.07s
P4	11	9	16	-	8.28s	27.19s
P5	11	11	16	-	12.63s	32.27s

Table 8.10: Sequential Blocks World - shortest plans

Problem	#blocks	#steps(fixed)	min. #moves	DLV$^{\mathcal{K}}$
P0	6	2	6	0.05s
P0	6	3	5	0.09s
P1	4	3	4	0.04s
P2	5	5	6	0.10s
P3	8	4	9	0.21s
P4	11	5	13	0.81s
P5	11	7	15	327s

Table 8.11: Parallel Blocks World – minimal number of moves at fixed plan length (α)

	upper bound	min. #steps	DLV$^{\mathcal{K}}$		DLV$^{\mathcal{K}}_{inc}$		CCALC	
			#moves	time	#moves	time	#moves	time
P0	6	2	6	0.52s	6	0.09s	6	4.05s
P1	4	3	5	0.07s	5	0.08s	4	2.95s
P2	7	5	9	0.39s	9	0.21s	6	3.70s
P3	10	4	-	-	12	0.43s	9	7.69s
P4	16	5	-	-	18	1.54s	13	20.45s
P5	16	7	-	-	26	3.45s	15	23.22s

Table 8.12: Parallel Blocks World – shortest plan (β)

plans: This is done by means of the `-costbound=`i command-line option taking the minimal sequential costs (i.e., the shortest sequential plan length as computed in Table 8.10) as an upper cost limit. As our encodings compute serializable plans, the minimal sequential length can be used as cost limit in this special case.

CCALC A similar technique can be used with CCALC when encoding bound costs through "additive fluents" [LL01].

Note that the incremental strategy (used by DLV$^{\mathcal{K}}_{inc}$ and CCALC) takes advantage of our specific formulation of the parallel Blocks World problem: In general, when allowing parallel actions which are not necessarily serializable and have arbitrary costs, the optimal parallel cost might differ from the optimal sequential solution. In particular, plans which are longer than the cheapest sequential plans (which, in this example, coincide with the shortest sequential plans) may need to be considered. This makes incremental search for a solution of problem (γ) infeasible in general.

The last test is finding a cheapest among the shortest plans, that is, problem (δ) in Section 4.2.2. Again we have tested the integrated encoding with an upper bound ($\mathcal{P}_\delta$) resp. incrementally finding the shortest plan. Unlike for problem (γ), we cannot derive a fixed cost limit from the sequential solution here; we really need to optimize costs, which makes an encoding in CCALC infeasible.

8.3.2 Results and Discussion – Blocks World

The Blocks World experiments show that DLV$^{\mathcal{K}}$ can solve various optimization tasks in a more effective and flexible way than the system compared. On the other hand, as already

		(γ)				(δ)		
	upper bound	steps/moves	DLV$^{\mathcal{K}}$	DLV$^{\mathcal{K}}_{inc}$	CCALC	steps/moves	DLV$^{\mathcal{K}}$	DLV$^{\mathcal{K}}_{inc}$
P0	6	3/5	38.5s	0.18s	5.89s	2/6	0.26s	0.09s
P1	4	3/4	0.07s	0.11s	3.47s	3/4	0.08s	0.08s
P2	7	5/6	2.08s	0.21s	5.65s	5/6	0.78s	0.28s
P3	10	5/8	-	1.57s	15.73s	4/9	177s	0.45s
P4	16	9/9	-	-	73.64s	5/13	-	1.86s
P5	16	11/11	-	-	167s	7/15	-	323s

Table 8.13: Parallel Blocks World – (γ),(δ)

stated above, for the minimal plan length encodings in Section 6.4, we can only solve the problems where a tight upper bound for the plan length is known. Iteratively increasing the plan length is more effective, especially if the upper bound is much higher than the actual optimal solution. This becomes drastically apparent when execution times seem to explode from one instance to the next, in a highly non-linear manner as in Table 8.10 where a solution for P3 can be found in reasonable time whereas P4 and P5 could not be solved within the time limit of 4000 seconds. This observation is also confirmed in the other tables (instance P5 in Table 8.11, etc.) and is partly explained by the behavior of the underlying DLV system, which is not geared towards plan search, and as a general purpose problem solver uses heuristics which might not work out well in some cases.

Interestingly, CCALC finds "better quality" parallel solutions for problem (β) (cf. Table 8.12), i.e. solutions with fewer moves, although it is significantly slower than our system on these instances. For the incremental encoding of problem (γ), CCALC seems even more effective than our system. However, CCALC offers no means of optimization; it allows for admissible but not for optimal planning. This makes our approach more flexible and general. As stated above, we could fortunately exploit the fixed cost bound in this particular example for CCALC, which is not possible in general instances of problem (γ).

Problem (γ) is also intuitively harder than simply finding a shortest plan or a cheapest among all shortest plans in general: While these problems can always be solved incrementally, for (γ) we must consider all plans of all lengths. A longer plan may be cheaper, so we cannot freeze the plan length once a (shortest) plan has been incrementally found.

8.3.3 Benchmark Problem and Encodings –TSP

Some experimental results on TSP with variable costs are reported in Tables 8.14 and 8.15. Unlike for blocks world, no comparable systems were available; none of the systems from above supports cost optimal planning as needed for solving this problem. Here, the plan length is always given by the number of cities.

Table 8.14 shows the results for our TSP instance on the Austrian province capitals as in Figure 6.7 (nine cities, 18 connections), with and without the exceptional costs as in Section 6.5 (with and without subscript *exc* in the table). Further instances reported in this table with different cost exceptions (*we*, *lwe*, *rnd*) are described below.

Results for some bigger TSP instances, given by the capitals of the 15 members of the European Union (EU) with varying connection graphs and exceptional costs are shown in Table 8.15. We have used the flight distances (km) between the cities as connection costs.

Instance	#cost exceptions	cost/time
$TSP_{Austria}$	0	15/0.31s
$TSP_{Austria,exc}$	2	15/0.32s
$TSP_{Austria,we}$	36	12/0.34s
$TSP_{Austria,lwe}$	54	11/0.35s
$TSP_{Austria,rnd}$	10	14/0.30s
$TSP_{Austria,rnd}$	50	15/0.31s
$TSP_{Austria,rnd}$	100	23/0.35s
$TSP_{Austria,rnd}$	200	36/0.37s

Table 8.14: TSP – Results for $TSP_{Austria}$ with varying exceptions

Instance	#conn.	#except.	cost/time
TSP_{EU1}	30	0	-/9.11s
$TSP_{EU1,we}$	30	60	-/11.93s
$TSP_{EU1,lwe}$	30	90	-/13.82s
$TSP_{EU1,rnd}$	30	100	-/11.52s
$TSP_{EU1,rnd}$	30	200	-/12.79s
$TSP_{EU1,rnd}$	30	300	-/14.64s
$TSP_{EU1,rnd}$	30	400	-/16.26s
TSP_{EU2}	30	0	16213/13.27s
$TSP_{EU2,we}$	30	60	13195/16.41s
$TSP_{EU2,lwe}$	30	90	11738/18.53s
$TSP_{EU2,rnd}$	30	100	15190/15.54s
$TSP_{EU2,rnd}$	30	200	13433/16.31s
$TSP_{EU2,rnd}$	30	300	13829/18.34s
$TSP_{EU2,rnd}$	30	400	13895/20.59s
TSP_{EU3}	35	0	18576/24.11s
$TSP_{EU3,we}$	35	70	15689/28.02s
$TSP_{EU3,lwe}$	35	105	14589/30.39s
$TSP_{EU3,rnd}$	35	100	19410/26.75s
$TSP_{EU3,rnd}$	35	200	22055/29.64s
$TSP_{EU3,rnd}$	35	300	18354/31.54s
$TSP_{EU3,rnd}$	35	400	17285/32.66s
TSP_{EU4}	35	0	16533/36.63s
$TSP_{EU4,we}$	35	70	12747/41.72s
$TSP_{EU4,lwe}$	35	105	11812/43.12s
$TSP_{EU4,rnd}$	35	100	15553/39.17s
$TSP_{EU4,rnd}$	35	200	13216/41.19s
$TSP_{EU4,rnd}$	35	300	16413/43.51s
$TSP_{EU4,rnd}$	35	400	13782/45.69s
TSP_{EU5}	40	0	15716/91.83s
$TSP_{EU5,we}$	40	80	12875/97.73s
$TSP_{EU5,lwe}$	40	120	12009/100.14s
$TSP_{EU5,rnd}$	40	100	13146/85.69s
$TSP_{EU5,rnd}$	40	200	12162/83.44s
$TSP_{EU5,rnd}$	40	300	12074/76.81s
$TSP_{EU5,rnd}$	40	400	12226/82.97s
$TSP_{EU5,rnd}$	40	500	13212/82.53s
TSP_{EU6}	40	0	17483/142.7s
$TSP_{EU6,we}$	40	80	14336/150.3s
$TSP_{EU6,lwe}$	40	120	13244/154.7s
$TSP_{EU6,rnd}$	40	100	15630/142.5s
$TSP_{EU6,rnd}$	40	200	14258/137.2s
$TSP_{EU6,rnd}$	40	300	11754/120.5s
$TSP_{EU6,rnd}$	40	400	11695/111.4s
$TSP_{EU6,rnd}$	40	500	12976/120.8s
TSP_{EU7}	55	0	15022/102.6s
$TSP_{EU7,we}$	55	110	12917/112.2s
$TSP_{EU7,lwe}$	55	165	11498/116.2s
$TSP_{EU7,rnd}$	55	100	13990/104.2s
$TSP_{EU7,rnd}$	55	200	12461/100.8s
$TSP_{EU7,rnd}$	55	300	13838/106.9s
$TSP_{EU7,rnd}$	55	400	12251/96.58s
$TSP_{EU7,rnd}$	55	500	16103/109.2s
$TSP_{EU7,rnd}$	55	600	14890/110.3s
$TSP_{EU7,rnd}$	55	700	17070/110.7s
TSP_{EU8}	64	0	10858/3872s
$TSP_{EU8,we}$	64	128	9035/3685s
$TSP_{EU8,lwe}$	64	192	8340/3324s
$TSP_{EU8,rnd}$	64	100	10283/2603s
$TSP_{EU8,rnd}$	64	200	9926/1372s
$TSP_{EU8,rnd}$	64	300	10028/1621s
$TSP_{EU8,rnd}$	64	400	8133/597.7s
$TSP_{EU8,rnd}$	64	500	8770/573.3s
$TSP_{EU8,rnd}$	64	600	8220/360.7s
$TSP_{EU8,rnd}$	64	700	6787/324.6s
$TSP_{EU8,rnd}$	64	800	11597/509.5s

Table 8.15: TSP – Various instances for the capitals of the 15 EU members

Instances TSP_{EU1}–TSP_{EU6} have been generated by randomly choosing a given number of connections from all possible connections between the 15 cities. Note that TSP_{EU1} has no solution; the time reported here is until $\texttt{DLV}^{\mathcal{K}}$ terminated, and for all other instances until the first optimal plan was found.

We have also tested some instances of more practical relevance than simply randomly choosing connections: TSP_{EU7} is an instance where we have taken the flight connections of three carriers (namely, Star Alliance, Alitalia, and Luxair), and in TSP_{EU8} we have included only direct connections of at most 1500km. Such a "capital hopping" is of interest for a small airplane with limited range, for instance.

For each instance in Tables 8.14–8.15 we have measured the execution time:

- without exceptional costs,
- with 50% costs for all connections on Saturdays and Sundays (weekends, *we*)
- with 50% costs for all connections on Fridays, Saturdays and Sundays (long weekends, *lwe*),
- for some random cost exceptions (*rnd*): We have added a number of randomly generated exceptions with costs between 0 and 10 for $\text{TSP}_{Austria}$ and between 0 and 3000 for the instances *EU1* to *EU8*.

8.3.4 Results and Discussion – TSP

Instance TSP_{EU8} shows the limits of our system: the given data allows for many possible tours, so finding an optimal one gets very tricky. On the other hand, a realistic instance like TSP_{EU7} with real airline connections is solved rather quickly, which is not very surprising: Most airlines have a central airport (for instance Vienna for Austrian Airlines) and few direct connections between the destinations served. This allows for much fewer candidate answer sets, when (as in reality) the number of airlines we consider is limited. E.g., TSP_{EU7} has no solution at all if only two out of Star Alliance, Alitalia, and Luxair are allowed. Of course, we cannot compete with dedicated TSP solvers/algorithms, which are able to solve much bigger TSP instances and have not been considered here. However, to our knowledge, none of these solvers can deal with features such as incomplete knowledge, defaults, time dependent exceptional costs, etc. directly. Our results even show that execution times are stable yet in case of many exceptions. In contrast, instance TSP_{EU8} shows that exceptions can also cause a significant speedup. This is due to the heuristics used by the underlying DLV system, which can single out better solutions faster if costs are not spread evenly like in TSP_{EU8} without exceptional costs.

8.4 Integrated Encodings

In this section, we evaluate the automatic generation of integrated (single-program) encodings to evaluate Σ_2^P-complete problems which has been presented in Section 4.2.3. Recall that this approach merges two HEDLPs, Π_{guess} and Π_{check}, where the answer sets of Π_{guess} serve as input for Π_{check}. Our approach merges these two programs into a single DLP Π_{solve} such that the answer sets of Π_{guess} for which Π_{check} has no answer set are computed. We

have conducted a series of experiments using this method for the problems outlined in the Section 4.2.5, and for the integrated computation of secure plans.

8.4.1 Benchmark Problem and Encodings – QBF, Strategic Companies

First, we investigated QBFs with one quantifier alternation and the Strategic Companies problem as introduced in Section 4.2.5. Both these problems are well-studied in Answer Set Programming. We were mainly interested in the following question:

- *What is the performance impact of our automatically generated, integrated encoding compared with existing ad hoc encodings of Σ_2^P problems?*

We have therefore compared our automatically generated integrated encodings of QBFs and Strategic Companies against the following ad hoc encodings:

(i) The integrated encoding obtained from our QBF "guess" and "check" programs from Section 4.2.5.1 have been compared against the ad hoc encoding for QBFs described in [LPF+02] (this encoding coincides with the one used in the proof of Theorem 2.2, p. 16).

(ii) The integrated encoding obtained from our "guess" and "check" formulation of Strategic Companies from Section 4.2.5.2 has been compared against the two ad hoc encodings for the Strategic Companies Problem from [EFLP00].

These two encodings significantly differ: The first encoding, which is very concise, integrates guessing and checking in only two rules; it is an illustrative example of the power of disjunctive rules:

```
strat(Y)  v  strat(Z) :- prod_by(X,Y,Z).
strat(W)  :-  contr_by(W,X,Y,Z),
              strat(X), strat(Y), strat(Z).
```

We will denote this encoding as $adhoc_1$ in the following. Here, predicate names are as in Section 4.2.5.2.

The second ad hoc encoding, denoted as $adhoc_2$, has a more obvious separate structure of the guessing and checking parts of the problem at the cost of some extra rules:

```
strat(X)  v -strat(X) :- company(X).                                  } Guess

:- prod_by(X,Y,Z), not strat(Y), not strat(Z).                        )
:- contr_by(W,X,Y,Z), not strat(W),                                   )
   strat(X), strat(Y), strat(Z).                                      )
:- not min(X), strat(X).                                              )
:- strat1(X,Y), -strat(Y).                                            )
:- strat1(X,X).                                                       } Check
min(X)  v  strat1(X,Y)  v  strat1(X,Z) :- prod_by(G,Y,Z),             )
                                          strat(X).                   )
min(X)  v  strat1(X,C) :- contr_by(C,W,Y,Z), strat(X),                )
                          strat1(X,W), strat1(X,Y), strat1(X,Z).      )
strat1(X,Y) :- min(X), strat(X), strat(Y), X<>Y.                      )
```

However, in our opinion, none of these two ad hoc encodings is obvious at first sight compared with the separate guess and check programs shown in Section 4.2.5.2.

Concerning (i) we have tested randomly generated QBF instances with n existentially and n universally quantified variables (QBF-n), and three literals per clause. In these instances, the number of clauses equals the overall number of variables (that is, $|X| + |Y|$, i.e. $2n$), as suggested by Cadoli et al. [CGS97].

Concerning (ii) we have chosen randomly generated instances involving n companies (STRATCOMP-n). Here, we have used a random instance generator kindly provided by Gerald Pfeifer (cf. [LPF+02]).

8.4.2 Benchmark Problem and Encodings – Secure Planning

Here, we were mainly interested in the following question:

- *What is the performance impact of the automatically generated, integrated encoding compared with interleaved computation of guess and check programs?*

To this end, we have tested the performance on solving some secure planning problems with integrated encodings compared with interleaved computation as implemented in $\texttt{DLV}^{\mathcal{K}}$ (cf. Chapter 5) For our experiments we have used elaborations of "Bomb in the Toilet" as described in Section 6.3.1, namely "Bomb in the Toilet with clogging" BTC(i), and "Bomb in the Toilet with uncertain clogging" BTUC(i), where we focused on world-state encodings. We remark that since secure and optimistic plans coincide for the knowledge state encodings of these examples, the integration of secure checks does not pay off.

For comparison with $\texttt{DLV}^{\mathcal{K}}$, we have tested BTC($i$) and BTUC($i$) with the integrated encoding for secure check $\mathcal{SC}_1$, $\Pi^{\mathcal{P}}_{sec,\mathcal{SC}_1}$, sketched in Proposition 4.20. Here, we used the alternative "guess" program $\Pi^{\mathcal{P}}_{guessOpt}$, which returns optimistic plans. This reflects the interleaved computation of $\texttt{DLV}^{\mathcal{K}}$ and proved to be faster than guessing arbitrary action sequences (as in $\Pi^{\mathcal{P}}_{guess}$, cf. Section 4.3.5 for details).

8.4.3 Results and Discussion

Since our method works on ground programs, we had to ground all instances (i.e. the corresponding guess and check programs) beforehand whenever dealing with non-ground programs. Here, we have used DLV grounding with most optimizations turned off[37]: Some optimizations during DLV grounding rewrite the program, adding new predicate symbols, etc. which we turned off in order to obtain correct input for the meta-interpreters.

In order to assess the effect of various optimizations and improvements to the transformation $tr(\cdot)$, we have also conducted the above experiments with the integrated encodings based on different optimized versions of $tr(\cdot)$.

The results of our experiments are shown in Tables 8.16-8.18. In theses tables, we report the following variations for the integrated encodings that we have tested:

- *meta* indicates the unoptimized meta-interpreter Π_{meta}
- *mod* indicates the non-modular optimization (**OPT**$_{mod}$) including the refinement for constraints.

[37] Respective ground instances have been produced with command `dlv -OR- -instantiate` (cf. the DLV-Manual for details [FP96]) which turns off most of the grounding optimizations.

- *dep* indicates the optimization ($\mathbf{OPT}_{dep}$) where `phi` is only guessed for literals mutually depending on each other through positive recursion.
- *opt* indicates both optimizations ($\mathbf{OPT}_{mod}$) and ($\mathbf{OPT}_{dep}$) turned on.

Remark 8.2. *We did not consider optimization ($\boldsymbol{OPT}_{pa}$) in our experiments, since the additional overhead for computing unfounded rules in the check programs we used did not pay off. In fact, ($\boldsymbol{OPT}_{pa}$) is irrelevant for QBF and Strategic Companies):*

(1) *($\boldsymbol{OPT}_{pa}$) does not play a role in our QBF encoding; the only rules in QBF_{check} with non-empty heads are always potentially applicable as their bodies are empty.*

(2) *As for Strategic companies, again all rules with non-empty heads are either possibly applicable or "switched off" by SC_{guess}. Since there are no positive dependencies among the rules,* `pa`$(\cdot)$ *does not play a role.*

(3) *Regarding the integrated encodings for the "Bomb in the Toilet" instances from Section 6.3.1, though there might be rules which are* not *possible applicable wrt. to a guessed plan, the additional overhead for computing unfounded sets did not pay off experimentally. Preliminary experiments with other well-known planning domains (e.g. SQUARE) showed similar results.*

We remark that the computation of predicate `pa`$(\cdot)$ *comes at a cost: Informally, a profit of optimization ($\boldsymbol{OPT}_{pa}$) might only be expected in domains with a reasonable number of rules which positively depend on each other and might on the other hand likely be "switched off" by particular guesses in Π_{guess}.*

All times reported in the tables represent the execution times for finding the first answer set where we set a time limit of 600 CPU seconds for QBFs and Strategic Companies, and 4000 CPU seconds for the "Bomb in the Toilet" instances. Moreover, we set a limit of 256 MB on memory consumption (in order to avoid swapping) for all the tests. A dash in the tables indicates again that one or more instances exceeded these limits.

The results in Tables 8.17-8.18 show that the "guess and saturate" strategy in our approach benefits a lot from optimizations for all problems considered. However, we emphasize that it might depend on the structure of Π_{guess} and Π_{check} which optimizations apply. We strongly believe that there is room for further improvements and optimizations both on the translation and for the underlying `DLV` engine.

Interestingly, for the QBF problem, the performance of our optimized translation stays within reach of the ad hoc encoding in [LPF+02] (*adhoc*) for small instances. Overall, the performance shown in Table 8.16 is within roughly a factor of 5-6 (with few exceptions for small instances), and thus scales similarly.

For Strategic companies, we have in Table 8.17 an even more interesting picture. Unsurprisingly, the automatically generated encoding is inferior to the first, succinct ad hoc encoding ($adhoc_1$); it is more than an order of magnitude slower and scales worse. However, while it is slower by a small factor than the second, more involved ad hoc encoding ($adhoc_2$) on small instances, it scales much better and quickly outperforms this encoding.

For the planning problems, the integrated encodings tested still stay behind the interleaved calls of $\texttt{DLV}^{\mathcal{K}}$. One might argue that the results of table 8.18 show that the integrated

	adhoc [LPF+02]		*meta*		*mod*		*dep*		*opt*	
	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
QBF-4	0.01s	0.02s	0.16s	0.18s	0.10s	0.15s	0.09s	0.11s	0.07s	0.09s
QBF-6	0.01s	0.02s	1.11s	1.40s	0.25s	1.12s	0.17s	0.21s	0.08s	0.12s
QBF-8	0.01s	0.06s	10.4s	16.3s	1.18s	7.99s	0.49s	0.87s	0.10s	0.23s
QBF-10	0.02s	0.09s	82.7s	165s	4.34s	30.7s	1.74s	3.67s	0.12s	0.36s
QBF-12	0.02s	0.16s	-	-	-	-	-	-	0.15s	0.79s
QBF-14	0.06s	1.21s	-	-	-	-	-	-	0.34s	5.87s
QBF-16	0.08s	1.85s	-	-	-	-	-	-	0.44s	10.3s
QBF-18	0.19s	7.12s	-	-	-	-	-	-	1.04s	38.8s
QBF-20	1.49s	21.3s	-	-	-	-	-	-	7.14s	101s

Average and maximum times for 50 randomly chosen instances per size.

Table 8.16: Experiments for QBF

	$adhoc_1$ [EFLP00]		$adhoc_2$ [EFLP00]		*meta*		*mod*		*dep*		*opt*	
	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX	AVG	MAX
STRATCOMP-10	0.01s	0.02s	0.05s	0.05s	0.66s	0.69s	0.49s	0.51s	0.36s	0.38s	0.13s	0.15s
STRATCOMP-15	0.01s	0.02s	0.11s	0.13s	1.82s	3.23s	1.50s	3.12s	0.64s	0.68s	0.20s	0.22s
STRATCOMP-20	0.02s	0.02s	0.26	0.27s	3.75s	3.90s	3.34s	3.61s	1.07s	1.13s	0.26s	0.27s
STRATCOMP-25	0.02s	0.02s	0.51s	0.54s	-	-	-	-	1.63s	1.68s	0.33s	0.35s
STRATCOMP-30	0.02s	0.03s	0.91s	0.97s	-	-	-	-	2.35s	2.47s	0.42s	0.44s
STRATCOMP-35	0.02s	0.03s	1.50s	1.60s	-	-	-	-	3.17s	3.27s	0.54s	0.56s
STRATCOMP-40	0.03s	0.03s	2.52s	2.70s	-	-	-	-	4.25s	4.43s	0.68s	0.71s
STRATCOMP-45	0.03s	0.04s	4.503	4.97s	-	-	-	-	5.46s	5.77s	0.84s	0.90s
STRATCOMP-50	0.03s	0.04s	8.38s	8.68s	-	-	-	-	6.73s	6.86s	1.00s	1.02s
STRATCOMP-60	0.04s	0.05s	22.6s	24.3s	-	-	-	-	10.2s	10.6s	1.47s	1.53s
STRATCOMP-70	0.04s	0.05s	44.2s	48.1s	-	-	-	-	14.7s	15.4s	2.05s	2.10s
STRATCOMP-80	0.04s	0.05s	75.9s	82.5s	-	-	-	-	19.7s	21.0s	2.78s	3.05s
STRATCOMP-90	0.05s	0.06s	125s	130s	-	-	-	-	26.8s	27.6s	3.67s	3.85s
STRATCOMP-100	0.06s	0.08s	196s	208s	-	-	-	-	34.8s	36.3s	4.70s	4.80s

Average and maximum times for 10 randomly chosen instances per size.

Table 8.17: Experiments for Strategic Companies

	DLV$^{\mathcal{C}}$	*meta*	*mod*	*dep*	*opt*
BTC(2)	0.01s	1.16s	0.80s	0.15s	0.08s
BTC(3)	0.11s	9.33s	9.25s	8.18s	4.95s
BTC(4)	4.68s	71.3s	67.8s	333s	256s
BTUC(2)	0.01s	6.38s	6.26s	0.22s	0.17s
BTUC(3)	1.78s	-	-	28.1s	13.0s
BTUC(4)	577s	-	-	-	2322s

BTC, BTUC with 2,3 and 4 packages.

Table 8.18: Experiments for Integrated Secure Planning – Bomb in the Toilet

encoding grows slower than the interleaved computation, but this conclusion could not be verified: Since the results grow exponentially with the number of packages for world state encodings in either case, the number of solvable instances is too small and more extensive tests will be necessary. In all cases, time limit of 4000 CPU seconds was exceeded earlier (for smaller instances) than memory, but especially for bigger instances of "Bomb in the Toilet" (and also for "Strategic Companies") there were also cases where memory exceeded within the time limits for some encodings (for instance BTUC(5), even with the optimized version of our transformation).

Clearly, the performance of the automatic integrated encodings was expected to stay behind the best ad hoc encodings or the interleaved methods. However, the strength of our approach outcrops if an integrated ad hoc encoding is non-obvious. Then, by our method such an encoding can be generated automatically from separate guess and check programs, which are often easy to formalize, while a manual integrated encoding may be difficult to find (as in the case of conformant planning or minimal update answer sets [EFST02] where such integrated encodings did not exist before for the general case).

8.5 Final Remarks

As for $\texttt{DLV}^{\mathcal{K}}$, in all tables great sensitivity to increasing plan length/search space can be observed, where execution times seem to grow drastically from one instance to the next. This can be partly explained by the general heuristics of the underlying DLV system, which which might not scale up well in some cases. For instance, DLV as a general purpose problem solver does not include special heuristics towards plan search. In particular, during the answer set generation process, no distinction is made between actions and fluents, which might be useful for planning tasks to control the generation of answer sets resp. plans; this may be part of further investigations.

We have seen that integrated encodings showed some promising results for hard problems, for instance on the Strategic Companies problem where an ad hoc guess and check encoding from the literature [EFLP00] was even outperformed. As for planning problems structural analysis of the domain beforehand and further optimizations might successfully improve our methods also for planning problems: For example, we might profit from the observation that the check programs for the cheap secure checks $\mathcal{SC}_1$ and $\mathcal{SC}_2$ already have a certain form. E.g., in the check program for $\mathcal{SC}_1$, there are no constraints for timestamps greater than 0, etc. An in-depth analysis of such properties is crucial for future improvements.

Chapter 9

A Practical Scenario – Planning as Monitoring

This chapter shall give a practical scenario where we apply our planning formalism in the field of agent monitoring as an application domain for planning. This chapter is somehow independent of the rest of this work in that the monitoring approach outlined here is not restricted to our particular approach to planning but we will introduce a general method for applying planning techniques in the fields of monitoring and design of multi-agent systems in order to give a clearer idea of a possible more realistic application domain for planning techniques. It shall also provide a perspective for further research directions.

9.1 Overview

Multi-Agent systems have been recognized as a promising paradigm for distributed problem solving, and numerous multi-agent platforms and frameworks have been proposed, which allow to program agents in languages ranging from imperative over object-oriented to logic-based ones [LMPG02]. A major problem which agent developers face with many platforms is verifying that a suite of implemented agents collaborate well to reach a certain goal (e.g., in supply chain management). Tools for automatic verification[38] are rare. Thus, common practice is geared towards extensive agent testing, employing tracing and simulation tools (if available).

We suggest a *monitoring* approach which aids in automatically detecting that agents do not collaborate properly. In the spirit of Popper's *principle of falsification*, it aims at refuting from (possibly incomplete) information at hand that an agent system works properly, rather than proving its correctness. In our approach, agent collaboration is described at an abstract level, and the single steps in runs of the system are examined to see whether the agents behave "reasonable," i.e., "compatible" to a collaboration plan for reaching a goal.

Even if the internal structure of some agents is unknown, we may get hold of the messages exchanged among them. A given message protocol allows us to draw conclusions about the

[38] By well-known results, this is impossible in general but often also in simple cases if details of some agents (e.g., in a heterogenous environment) are missing.

correctness of the agent collaboration. Our monitoring approach hinges on this fact and involves the following steps:

(1) The intended collaborative behavior of the agents is modeled as a planning problem. More precisely, knowledge about the agent actions (specifically, messaging) and their effects is formalized in an *action theory*, T, which can be reasoned about to automatically construct *plans* as sequences of actions to reach a given goal.

(2) From T and the collaborative goal G, a set of intended plans, *I-Plans*, for reaching G is generated via a planner. For this purpose we will employ $\text{DLV}^{\mathcal{K}}$.

(3) The observed agent behavior, i.e., the message actions from a message log, is then compared to the plans in *I-Plans*.

(4) In case an incompatibility is detected, an error is flagged , pinpointing to the last action causing the failure so that further steps might be taken by the developer or user, respectively.

Steps 2-4 can be done by a special *monitoring agent*, which is added to the agent system providing support both in testing, and in the operational phase of the system. Among the benefits of this approach are the following:

- It allows to deal with collaboration behavior regardless of the implementation language(s) used for single agents.
- Depending on the planner used in step 2, different kinds of plans (optimal, conformant, admissible ...), might be considered, reflecting different agent attitudes and collaboration objectives.
- Changes to the agent messaging by the system designer may be transparently incorporated to the action theory T, without further need to adjust the monitoring process.
- Furthermore, T adds to a formal system specification, which may be reasoned about and used in other contexts.
- As a by-product, the method may also be used for automatic *protocol generation*, i.e., determine the messages needed and their order, in a (simple) collaboration.

The next section describes the basic agent framework that we build upon and presents a simplified version of a multi-agent system in the postal services domain. After that, in Section 9.3 we describe how to model the intended behavior of a multi-agent system as an abstract planning problem, and instantiate this for our example system using action language $\mathcal{K}$. Our approach to agent monitoring is then discussed in Section 9.4, where we also investigate some fundamental properties. After a brief discussion of the implementation in Section 9.5 and a review of related work on planning in multi-agent systems in Section 9.6, we conclude this chapter with an outlook on further research.

9.2 Message Flow in a Multi-Agent System

In a multi-agent system (*MAS*), a set of autonomous agents are collaborating to reach a certain goal. Our aim is to monitor (some aspects of) the behavior of the agents in order to detect inconsistencies and help debugging the whole system.

As opposed to verification, monitoring a *MAS* does not require a complete specification of the behavior of the particular agents. Rather, we adopt a more general (and in practice much more realistic) view: We do not have access to the (entire) internal state of a single autonomous agent, but we are able *to observe the communication between agents* of the system. By means of its communication capabilities, an agent can potentially control another agent. Our aim is to draw conclusions about the state of a multi-agent system by monitoring the message protocol and its correspondence to reasonable plans for the agent collaboration.

Basic Framework

We consider multi-agent systems consisting of a finite set $\mathsf{A} = \{\mathfrak{a}_1, \ldots, \mathfrak{a}_n\}$ of collaborating agents $\mathfrak{a}_i$. Although agents may perform a number of different (internal) actions, we assume that only one action is externally observable, namely an action called `send_msg`(m), which allows an agent to send a message, m, to another agent in the system. Every `send_msg` action is given a timestamp and recorded in a message-log file containing the history of messages sent. The following definitions do not assume a sophisticated messaging framework and apply to almost any *MAS*. Thus, our framework is not bound to a particular *MAS*.

Definition 9.1 (Message, $\mathcal{M}_{log}$ file). *A* message *is a quadruple* $m = \langle \mathsf{s}, \mathsf{r}, c, d \rangle$, *where* $\mathsf{s}, \mathsf{r} \in \mathsf{A}$ *are the identifiers of the* sending *and the* receiving *agents, respectively;* $c \in \mathsf{C}$ *is from a finite set* C *of* message commands*;* d *is a list of constants representing the* message data*. A* message-log file *is an ordered sequence* $\mathcal{M}_{log} = t_1{:}m_1, t_2{:}m_2, \ldots, t_k{:}m_k$ *of messages* m_i *with timestamps* t_i*, where* $t_i \leq t_{i+1}$*,* $i < k$*.*

The set C constitutes a set of message *performatives* specifying the intended meaning of a message. In other words, it is the type of a message according to speech act theory: the illocutionary force of an utterance. These commands may range from `ask`/`tell` primitives to application specific commands fixed during system specification.

Often, an agent $\mathfrak{a}_i$ will not send every kind of message, but use a message repertoire $\mathsf{C}_i \subseteq \mathsf{C}$. Moreover, only particular agents might be message recipients (allowing for simplified formats). Given that the repertoires C_i are pairwise disjoint and each message type c has a unique recipient, we use $\langle \mathsf{c}, d \rangle$ in place of $m = \langle \mathsf{s}, \mathsf{r}, \mathsf{c}, d \rangle$.

Finally, we assume a fixed bound on the time within the next action should happen in the *MAS*, i.e., a timeout for each action (which may depend on previous actions), which allows to see from $\mathcal{M}_{log}$ whether the *MAS* is stuck or still idle.

Gofish Post Office

We consider an example *MAS* called *Gofish Post Office* for postal services. Its goal is to improve postal product areas by mail tracking, customer notifications, and advanced quality control. The following scenario is our running example:

Example scenario: Pat drops a package, p_1, for a friend, Sue, at the post office. In the evening, Sue gets a phone call that a package has been sent. The next day, Sue decides to pick up the package herself at the post office on her way to work. Unfortunately, the clerk has to tell her that the package is already on a truck on its way to her home.

The overall design of the *Gofish MAS* is depicted in Figure 9.1. An *event dispatcher agent* (disp) communicates system relevant (external) events to an *event management agent* (em) which maintains an event database. Information about packages is stored in a package database manipulated by a *package agent* (pa). The *notification agent* (notify) notifies customers about package status and expected delivery time, for which it maintains a statistics database. Finally, a *zip agent* (zip) informs responsible managers, stored in a manager database, about zip codes not being well served.

Example 9.1 (Simple *Gofish*). To keep things simple and illustrative, we restrict the *Gofish MAS* to the package agent, pa, the event management agent, em, and the event dispatcher agent, disp; thus, $\mathsf{A} = \{\mathtt{pa}, \mathtt{em}, \mathtt{disp}\}$.

The event dispatcher informs the event manager agent about the drop off of a package (identified by a unique identifier), its arrival at the distribution center, its loading on a truck, its successful delivery, or when a recipient shows up at the distribution center to pick up a package by herself: $\mathsf{C}_{\mathtt{disp}} = \{\mathsf{dropOff}, \mathsf{distCenter}, \mathsf{truck}, \mathsf{delivery}, \mathsf{pickup}\}$. The event manager agent instructs the package agent to add a package to the package database after drop off, as well as to update the delivery time after delivery or customer pickup: $\mathsf{C}_{\mathtt{em}} = \{\mathsf{addPackage}, \mathsf{setDelivTime}\}$. The package agent here only receives messages, thus $\mathsf{C}_{pa} = \{\}$.
Running scenario: The message-log $\mathcal{M}_{log}$ contains the messages: $m_1 = \langle \mathsf{dropOff}, p_1 \rangle$, $m_2 = \langle \mathsf{addPackage}, p_1 \rangle$, $m_3 = \langle \mathsf{distCenter}, p_1 \rangle$, $m_4 = \langle \mathsf{truck}, p_1 \rangle$, and $m_5 = \langle \mathsf{pickup}, p_1 \rangle$. The corresponding (time-stamped) entries are

0: $\langle \mathtt{disp}, \mathtt{em}, \mathsf{dropOff}, p_1 \rangle$,
5: $\langle \mathtt{em}, \mathtt{pa}, \mathsf{addPackage}, p_1 \rangle$,
13: $\langle \mathtt{disp}, \mathtt{em}, \mathsf{distCenter}, p_1 \rangle$,
19: $\langle \mathtt{disp}, \mathtt{em}, \mathsf{truck}, p_1 \rangle$, and
20: $\langle \mathtt{disp}, \mathtt{em}, \mathsf{pickup}, p_1 \rangle$.

◇

9.3 Modeling Agent Behavior via Declarative Planning

We now discuss how to formalize the intended collaborative agent behavior as an action theory for planning in $\mathcal{K}$, which encodes the legal message flow. In it, actions correspond to messages and fluents represent assumptions about the current state of the world.

We remark that this approach is not limited to $\mathcal{K}$. Under suitable encodings, we could use planning formalisms like STRIPS [FN71], PDDL [GHK+98] or HTN [EHN94] based planners to model simple agent environments. In fact, HTN planning has recently been incorporated in a *MAS* [DMANZ02]. Another powerful language suitable for modeling control knowledge and plans for agents is GOLOG [LRL+97]. However, due to its high expressive power (loop, conditionals) automated plan generation is limited in this formalism. In the following we give a generic formulation of our approach, independent of a particular planning mechanism. Then, we instantiate this high-level description using our action language $\mathcal{K}$.

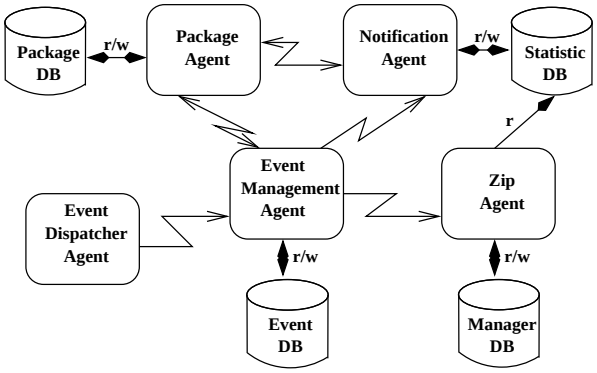

Figure 9.1: The *Gofish* post office system.

Modeling Intended Behavior of a *MAS*

Our approach to formalize the intended collaborative behavior of a *MAS* consisting of agents $\mathsf{A} = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$ as a planning problem $\mathcal{P}$ comprises three steps:

Step 1: *Actions (Act).* Declare a corresponding *action* for each message $m = \langle \mathsf{s}, \mathsf{r}, \mathsf{c}, d \rangle$ in our domain, i.e., we have $\mathsf{c}(\mathsf{s}, \mathsf{r}, d) \in Act$ (see Def. 9.1). Again, if the message repertoires C_i are pairwise disjoint and each message type c has a unique recipient, we use in short $\mathsf{c}(d)$. These actions might have effects on the states of the agents involved and will change the properties that hold on them.

Step 2: *Fluents (Fl).* Define properties, *fluents*, of the "world" that are used to describe action effects. We distinguish between the sets of *internal* fluents, Fl_{a}, of a particular agent a[39] and *external* fluents, Fl_{ext}, which cover properties not related to specific agents. These fluents are often closely related to the message performatives C_i of the agents.

Step 3: *Theory (T)* and *Goal (G).* Using the fluents and actions from above, state various axioms about the collaborative behavior of the agents as a *planning theory* T. The axioms describe how the various actions change the state and under which assumptions they are executable. Finally, state the ultimate *Goal* G (in the running scenario: to deliver the package) suitable for the chosen planning formalism.

We end up with a *planning problem* $\mathcal{P}$ whose *solutions* are a set of $\mathcal{P}$*-Plans*. Note that the precise formulation of these notions depends on the underlying planning formalism. For example, in HTN planning one has to specify *operators* and *methods* and their effects (this is closely related to *Act* and *Fl* above), as well as a domain description and a task list (which corresponds to T and G above): we refer to [DKN02b] for a full discussion. The above is a generic formulation suitable for many planning frameworks. Here, we will exemplify the approach for the $\mathtt{DLV}^{\mathcal{K}}$ planning framework.

[39] Internal fluents especially can describe private values which might be inaccessible by an external observer.

Using Action Language $\mathcal{K}$

In this section, we instantiate the planning problem described above to a problem $\mathcal{P}$ formulated in action language $\mathcal{K}$.

In $\mathcal{K}$, as for Steps 1 and 2 we first have to identify and declare the relevant actions *Act* and fluents *Fl* from our *MAS*. Concerning Step 3, in the $\mathcal{K}$ framework T comprises background knowledge Π and all axioms of the domain formalized as causation rules and executability conditions as well as a description of the initial state, i.e. a $\mathcal{K}$ planning domain *PD*. These steps yield a $\mathcal{K}$ planning domain where finally the goal G, can be added as a set of ground fluent literals in order to obtain the $\mathcal{K}$ formulation of the planning problem $\mathcal{P} = \langle PD, G \rangle$ corresponding to our intended *MAS* collaboration.

Example 9.2 (Simple *Gofish* cont'd). In the *Gofish* example, the following $\mathcal{K}$ actions and fluents are declared:

```
actions : dropOff(P) requires pkg(P).
          addPkg(P) requires pkg(P).
          distCenter(P) requires pkg(P).
          truck(P) requires pkg(P).
          delivery(P) requires pkg(P).
          pickup(P) requires pkg(P).
          setDelivTime(P) requires pkg(P).
```

} *Act*

```
fluents : pkgAt(P,Loc) requires pkg(P),loc(Loc).
          delivered(P) requires pkg(P).
          recipAtHome(P) requires pkg(P).
          added(P) requires pkg(P).
          delivTimeSet(P) requires pkg(P).
```

} *Fl*

Actions simply correspond to the defined messages in our *MAS*. The first three external fluents describe the current location of a package, whether it has successfully been delivered, and whether its recipient is at home, respectively. The last two fluents are internal fluents about the state of agent pa; whether the package has already been added to the package database resp. whether the delivery time has been set properly.

A possible package (e.g., a generic p_1) and its locations are background knowledge represented by the set of facts $\Pi_{Gofish} = \{pkg(p_1), loc(drop), loc(dist), loc(truck)\}$. Now we specify further axioms for PD_{Gofish} as follows:

```
initially : recipAtHome(p1).
always :    noConcurrency.
 inertial pkgAt(P,L).       inertial delivered(P).
 inertial recipAtHome(P).   inertial added(P).

 executable dropOff(P) if not added(P).
 caused pkgAt(P,drop) after dropOff(P).
 nonexecutable dropOff(P) if pkgAt(P,drop).

 executable addPkg(P) if pkgAt(P,drop),not added(P).
 caused added(P) after addPkg(P).
```

```
executable distCenter(P) if added(P), pkgAt(P, drop).
caused pkgAt(P, dist) after distCenter(P).
caused -pkgAt(P, drop) after distCenter(P).

executable truck(P) if pkgAt(P, dist), not delivered(P).
caused pkgAt(P, truck) after truck(P).
caused -pkgAt(P, dist) after truck(P).

executable delivery(P) if pkgAt(P, truck), not delivered(P).
caused delivered(P) after delivery(P), recipAtHome(P).

executable setDelivTime(P, DTime) if delivered(P).
caused delivTimeSet(P) after setDelivTime(P).

executable pickup(P) if pkgAt(P, dist), not delivered(P).
executable pickup(P) if pkgAt(P, truck), not delivered(P).
caused delivered(P) after pkgAt(P, dist), pickup(P).
total recipAtHome(P) after pickup(P).
```

Most of the theory is self-explanatory. The recipient is at home initially. The keyword `noConcurrency` specifies that concurrent actions/messages are disallowed.[40]

An important aspect is modeled by the final `total` statement. It expresses uncertainty whether after a pickup attempt at the distribution center, the recipient will be back home, in particular in time before the truck arrives to deliver the package, if it was already on the way. Finally, the goal is $G = \texttt{delivTimeSet}(p_1)$ or

$$\texttt{delivTimeSet}(p_1)\ ?(l)$$

in $\mathcal{K}$ notation for plan length l.

The following (optimistic) plans reach G in 7, 6 and 5 steps, respectively:

$$\begin{aligned}
P_1 &= \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1); \\
&\quad \texttt{pickup}(p_1); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle \\
P_2 &= \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1); \\
&\quad \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle \\
P_3 &= \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{pickup}(p_1); \texttt{setDelivTime}(p_1)\rangle
\end{aligned}$$

In P_1, the recipient shows up at the distribution center after the package is loaded on the truck and the truck is on its way. In P_2, the package is successfully delivered before the recipient comes to pick it up herself, whereas in P_3, she picks up the package before it has been loaded on the truck.

Running scenario: According to the message history in $\mathcal{M}_{log}$, we see that plan P_2 is infeasible, as well as P_3 since the package can not be handed over to Sue at the distribution center. Thus, only P_1 remains for successful task completion. ◇

[40] In this particular application we might even want to ensure that exactly one action occurs per time (since naturally $\mathcal{M}_{log}$ is sequential and does not contain empty entries) which can be ensured by $\texttt{DLV}^{\mathcal{K}}$'s options `-planminactions=1` and `-planmaxactions=1`.

9.4 Agent Monitoring

The overall aim of adding a monitoring agent (monitor) is to *aid debugging a given MAS*. We can distinguish between two principal types of errors: *(1) design errors*, and *(2) implementation (or coding) errors*. While the first type means that the model of the system is wrong (i.e., the *MAS* behaves correctly to the model of the designer of the *MAS*, but this model is faulty and does not yield the desired result in the application), the second type points to more mundane mistakes in the actual code of the agents: the code does not implement the formal model of the system (i.e., the actions are not implemented correctly).

Note that often it is very difficult, if not impossible at all, to distinguish between design and implementation errors. But even before the system is deployed, the planning problem $\mathcal{P}$ can be given to a planner and thus the overall existence of a solution can be checked. If there is no solution, this is clearly a design error and the monitoring agent can pinpoint where exactly the planning fails (assuming the underlying planner has this ability). If there are solutions, the agent designer can check them and thus critically examine the intended model.

However, for most applications the bugs in the system become apparent only at runtime. Our proposed monitoring agent has the following structure.

Definition 9.2 (Structure of the monitoring agent). *The agent* monitor *loops through the following steps:*

1. *Read and parse the message log $\mathcal{M}_{log}$. If $\mathcal{M}_{log} = \emptyset$, the set of plans for $\mathcal{P}$ may be cached for later reuse.*
2. *Check whether an action timeout has occurred.*
3. *If this is not the case, compute the current* intended plans *(according to the planning problem description and additional info from the designer) compatible with the actions as executed by the* MAS.
4. *If no compatible plans survive, or the system is no more idle, then inform the agent designer about this situation.*
5. *Sleep for some pre-specified time.*

We now elaborate more deeply into these tasks.

Checking *MAS* behavior: monitor continually keeps track of the *messages sent between the agents*. They are stored in the message-log, $\mathcal{M}_{log}$, which is accessible by monitor. Thus for monitor, the behavior of the *MAS* is completely determined by $\mathcal{M}_{log}$. We think this is a realistic abstraction from internal agent states. Rather than describing all the details of each agent (which might be unknown, e.g. if legacy agents are involved), the kinds of messages sent by an agent can be chosen so as to give a declarative high-level view of it. In the simplified *Gofish* example, these messages for agents em, disp, pa are given by $\mathsf{C}_{\mathtt{em}}$, $\mathsf{C}_{\mathtt{disp}}$, and $\mathsf{C}_{\mathtt{pa}}$ (see Section 9.2).

Intended behavior and compatibility: The desired collaborative *MAS* behavior is formalized as a planning problem $\mathcal{P}$ (e.g., in language $\mathcal{K}$, cf. Section 9.3). Thus, even before

the *MAS* is in operation, problem $\mathcal{P}$ can be fed into a planner which computes potential plans to reach a goal. Agent monitor is exactly doing that.

In general, not all $\mathcal{P}$-Plans may be admissible, as constraints may apply (derived from the intended collaborative behavior). [41] E.g., some actions ought to be taken in fixed order, or actions may be penalized with costs whose sum must stay within a limit. We thus distinguish a set *I-Plans*($\mathcal{P}$) $\subseteq$*P-Plans* as *intended plans* (of the *MAS* designer).

It is perfectly possible that the original problem has successful plans, yet after some actions executed by the *MAS*, these plans are no longer valid. This is the interesting case for the agent designer since it clearly shows that something has gone wrong: monitor can pinpoint to the precise place indicating which messages have when caused the plan to collapse. Because these messages are related to actions executed by the agents, information about them will help to debug the *MAS*. In general, it is difficult to decide whether the faulty behavior is due to a coding or design error. However, the info given by monitor will aid the agent designer to detect the real cause.

Messages from monitor: Agent monitor continually checks and compares the actions taken so far for compatibility with all current plans. Once a situation has arisen in which no successful plan exists (detected by the planner employed), monitor writes a message into a separate file containing (1) the first action that caused the *MAS* to go into a state where the goal is not reached, (2) the sequence of actions taken up to this action, and (3) all the possible plans *before* the action in 1) was executed (these are all plans compatible with the *MAS* behavior up to it).

In the above description, we made heavily use of the notion of a *compatible* plan. Before giving a formal definition, we consider our running scenario. In *Gofish*, all three plans P_1, P_2, P_3 generated from the initial problem coincide on the first three steps: dropOff(p_1), addPkg(p_1), and distCenter(p_1).

Running scenario (coding error): Suppose on a preliminary run of our scenario, $\mathcal{M}_{log}$ shows m_1=dropOff(p_1). This is compatible with each plan P_i, $i \in \{1,2,3\}$. Next, m_2=distCenter(p_1). This is incompatible with each plan; monitor detects this and gives a warning. Inspection of the actual code may show that the command for adding the package to the database is wrong. While this doesn't result in a livelock (the *MAS* is still idle), the database was not updated. Informed by monitor, this is detected at this stage already.

After correction of this coding error, the *MAS* may be started again and another error shows up:

Running scenario (design error): Instead of waiting at home (as in the "standard" plan P_2), Sue shows up at the distribution center and made a pickup attempt. This "external" event may have been unforeseen by the designer (problematic events could also arise from *MAS* actions). We can expect this in many agent scenarios: we have no complete knowledge about the world, unexpected events may happen, and action effects may not fully determine the next state.

Only plan P_1 remains to reach the goal. However, there is *no guarantee of success*, if Sue is not back home in time for delivery. This situation can be easily captured in the framework of $\mathcal{K}$ by secure plans. As can be easily seen, P_1 is not secure. Thus, a design

[41] This might depend on the capabilities of the underlying planning formalism to model constraints such as cost bounds or optimality wrt. resource consumption etc.

error is detected, if delivering the package must be guaranteed under any circumstances. Based on a generic planning problem $\mathcal{P}$, we now define compatible plans as follows.

Definition 9.3 ($\mathcal{M}_{log}$ compatible plans). *Let the planning problem $\mathcal{P}$ model the intended behavior of a* MAS*, which is given by a set I-Plans$(\mathcal{P}) \subseteq$ P-Plans. Then, for any message log $\mathcal{M}_{log} = t_1{:}m_1, \ldots, t_k{:}m_k$, we denote by C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n)$, $n \geq 0$, the set of plans from I-Plans$(\mathcal{P})$ which comply on the first n steps with the actions $m_1, \ldots, m_n$.*

Respecting that DLV$^{\mathcal{K}}$, is capable of computing optimistic and secure plans, we denote for any $\mathcal{K}$ planning problem $\mathcal{P}^{\mathcal{K}}$ by X-Plans$^o(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n)$ (resp. X-Plans^s $(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n)$ the set of all optimistic (resp. secure) plans for $\mathcal{P}^{\mathcal{K}}$ with the above property, $X \in \{\mathsf{I}, \mathsf{C}\}$.

Definition 9.4 (Culprit$(\mathcal{M}_{log}, \mathcal{P})$). *Let $t_n{:}m_n$ be the first entry of $\mathcal{M}_{log}$ such that either (i) C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n) = \emptyset$ or (ii) a timeout is detected. Then, Culprit$(\mathcal{M}_{log}, \mathcal{P})$ is the pair $\langle t_n{:}m_n, idle\rangle$ if (i) applies and $\langle t_n{:}m_n, timeout\rangle$ otherwise.*

Initially, $\mathcal{M}_{log}$ is empty and thus C-Plans$(\mathcal{P})$ = I-Plans$(\mathcal{P})$. As more and more actions are executed by the *MAS*, they are recorded in $\mathcal{M}_{log}$ and the set C-Plans$(\mathcal{P})$ shrinks. monitor can thus compare at any point in time whether C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n)$ is empty or not. Whenever this happens, Culprit$(\mathcal{M}_{log}, \mathcal{P})$ is computed and pinpoints to the problematic action.

Running scenario: Under guaranteed delivery (i.e., secure planning), monitor writes Culprit$(\mathcal{M}_{log}, \mathcal{P}) = \langle 20{:}m_5, idle\rangle$ (the *pickup*(p_1) message) on a file, and thus clearly points to a situation missed in the *MAS* design. Note that there are also situations where everything is fine; if pickup would not occur, agent monitor would not detect a problem at this stage.

9.4.1 Properties

We can show that the agent monitoring approach has desirable properties. The first result concerns its soundness.

Theorem 9.1 (Soundness). *Let the planning problem $\mathcal{P}$ model the intended collaborative* MAS *behavior, given by I-Plans$(\mathcal{P}) \subseteq$ P-Plans. Let $\mathcal{M}_{log}$ be a message log. Then, the* MAS *is implemented incorrectly if Culprit$(\mathcal{M}_{log}, \mathcal{P})$ exists.*

Semantically, the intended collaborative *MAS* behavior (described in any formalism) may manifest in a set of trajectories as described for $\mathcal{K}$ planning problems, where trajectories correspond to possible runs of the *MAS* (sequences of states and executed actions). On the other hand, optimistic plans for a $\mathcal{K}$ planning problem $\mathcal{P}$ are projected trajectories. We say that a set OP of such plans *covers* the intended collaborative *MAS* behavior, if each run of the *MAS* corresponds to some trajectory whose projection is in OP. For example, this holds if OP is the set of all optimistic plans for a $\mathcal{K}$ planning problem $\mathcal{P}$ and the intended collaborative *MAS* behavior is given by a secure plan in $\mathcal{K}$.

We have:

Theorem 9.2 (Soundness of $\mathcal{P}$ Cover). *Let $\mathcal{P}$ be a $\mathcal{K}$ planning problem, such that I-Plans$^o(\mathcal{P})$ covers the intended collaborative* MAS *behavior. Let $\mathcal{M}_{log}$ be a message log. Then,* MAS *is implemented incorrectly if Culprit$(\mathcal{M}_{log}, \mathcal{P})$ exists.*

As for completeness, we need the assertion that plans can not grow arbitrarily long, i.e., have an upper bound on their length [42] otherwise, an error may not be detected in finite time.

Theorem 9.3 (Completeness). *Let the planning problem $\mathcal{P}$ model the intended collaborative* MAS *behavior, given by I-Plans($\mathcal{P}$) $\subseteq$ $\mathcal{P}$-Plans where plans are bounded. If the* MAS *is implemented incorrectly, then there is some message log $\mathcal{M}_{log}$ such that either (i) C-Plans($\mathcal{P}, \mathcal{M}_{log}, 0$) $= \emptyset$, or (ii) Culprit($\mathcal{M}_{log}, \mathcal{P}$) exists.*

In (i), we can conclude a design error, while in (ii) a design or coding error may be present. There is no similar completeness result for $\mathcal{P}$ covers, i.e. completeness holds of course only iff the planning formalism at hand provides exactly the intended plans and not only covers them: note that in our running scenario, a design error is detected for secure plans as *MAS* collaborative behavior formalism. However, the culprit vanishes if the cover contains plan P_1, which is compatible with $\mathcal{M}_{log}$.

As for complexity, we mention that in expressive planning formalisms like $\mathcal{K}$, deciding whether C-Plans($\mathcal{P}$, $\mathcal{M}_{log}$, n) $\neq \emptyset$ or Culprit($\mathcal{M}_{log}, \mathcal{P}$) exists from $\mathcal{P}$, $\mathcal{M}_{log}$ and n is at least NP-hard in general, which is inherited from the expressiveness of the underlying planning language.

In particular, if the intended behavior is expressed by optimistic/secure plans in $\mathcal{K}$, deciding C-Plans($\mathcal{P}$, $\mathcal{M}_{log}$, n) $\neq \emptyset$ tantamounts to optimistic/secure plan existence; the plan prefix given by $\mathcal{M}_{log}$, n can be easily encoded in the planning problem itself by adding respective constraints.

Let $\langle m_1, m_2, \ldots, m_n \rangle$ be a sequence of actions, and $\mathcal{P}$ =$\langle PD$,q$\rangle$ a $\mathcal{K}$ planning problem. Let $\mathcal{P}$' be the problem obtained from $\mathcal{P}$ by extending PD as follows:

```
initially: f1.
always:    caused false after not m_1, f1.
           caused f2 after m_1.
           caused false after not m_2, f2.
           caused f3 after m_2.
           ⋮
           caused false after not m_n, fn.
```

where f1, ..., fn are new distinct propositional fluents. We obtain:

Proposition 9.4. *Let I-Plans^o (or I-Plans^s, resp.) be the intended plans of $\mathcal{P}$. Then, C-Plans($\mathcal{P}$, $\mathcal{M}_{log}$, n) $\neq \emptyset$ iff $\mathcal{P}$' has an optimistic (or secure, resp.) plan.*

The corresponding complexity results for $\mathcal{K}$ can be found in Section 3.3. Similar considerations apply for the existence of Culprit($\mathcal{M}_{log}, \mathcal{P}$).

We remind that, NP-hardness (or even worse, if secure plans are required) is a theoretical worst-case measure, though, and still solutions for many instances can be found quickly, especially if optimistic planning is required only. Moreover, there are instance classes which

[42] for plans computed by DLV$^{\mathcal{K}}$ this is trivially true due to the restriction to given plan length as part of the input; all the possible plans up to an upper bound of plan length can be iteratively computed as discussed in Section 7.1.2

are polynomial time solvable and for which $\texttt{DLV}^{\mathcal{K}}$ is guaranteed to compute plans in polynomial time. This highly depends on which requirements for intended plans we have and how complicated the corresponding planning problem is.

For small domains, where the number of plans is moderate, I-Plans (or C-Plans, resp.) might simply be cached such that checking against $\mathcal{M}_{log}$ becomes trivial.

9.5 Implementation

To demonstrate the proposed approach, a running example has been implemented. The *Gofish MAS* and Agent monitor is developed within IMPACT (*Interactive Maryland Platform for Agents Collaborating Together*). Note that in principle our approach is completely independent of any specific agent system. We refer to [SBD+00] for the details of IMPACT.

Each agent consists of a set of *data types*, *API functions*, *actions*, and an *agent program* that includes some rules prescribing its behaviors. Since we use $\texttt{DLV}^{\mathcal{K}}$ [EFL+03a] as the planner, a new connection module has been created within Agent monitor so that monitor can access the $\texttt{DLV}^{\mathcal{K}}$ planner. In this way, before the *Gofish MAS* operates, we feed $\mathcal{P}^{\mathcal{K}}_{Gofish}$ into monitor, which then exploits $\texttt{DLV}^{\mathcal{K}}$ to compute all potential plans including both secure and optimistic plans.

Running scenario: The *Gofish* post office guarantees the package delivery within 24 hours of dropOff (time 0). Consider the case that Sue wanted to pick up her package (p_1=0x00fe6206c.1) at the distribution center. Unfortunately, it has been loaded on the truck. Sue did not come back home in time, therefore the package wasn't delivered in time. Thus after the "pickup" action at time 20, the *MAS* was keeping idle till a timeout (24 in this example) was detected by monitor. In the end, monitor generated a log file as follows (see also the project web page [43]):

> Problematic action:
> $\texttt{20:pickup}(0x00fe6206c.1), \texttt{timeout}$
>
> Actions executed:
> $\texttt{0:dropOff}(0x00fe6206c.1); \texttt{5:addPkg}(0x00fe6206c.1);$
> $\texttt{13:distCenter}(0x00fe6206c.1); \texttt{19:truck}(0x00fe6206c.1)$
>
> Possible plans before problematic action:
>
> $\langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1);$
> $\texttt{pickup}(p_1); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1) \rangle$
>
> $\langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{truck}(p_1);$
> $\texttt{delivery}(p_1); \texttt{setDelivTime}(p_1) \rangle$

9.6 Related Work

In contrast to research on plan generation as such, there has been relatively little work on the use of plans to monitor execution. Plans can be executed by one agent or by many collaborative agents. In this section, we review previous work on monitoring problem in (1) single-agent settings, and (2) multi-agent environments.

[43] <URL:http://www.cs.man.ac.uk/~zhangy/project/monitor/>

Monitoring in Single-Agent Settings

Interleaving monitoring with plan execution has been addressed in the context of single agent environment in [GRS98], where the authors presented a situation calculus-based account of execution monitoring for robot programs written in Golog. A situation calculus specification is given for the behavior of Golog programs. Combined with the interpretation of Golog programs, an execution monitor detects the discrepancy after each execution of a primitive action. Once a discrepancy is found, the execution monitor checks whether it is *relevant* in the current state, that is, whether preconditions of next action still hold with the effect of exogenous action. If this exogenous action does matter, a *recover* mechanism will be invoked. The method of recovering is based on planning. A new plan (or program) is computed whose execution will make things right by way of leading the current state to the desired situation, had the exogenous action not occurred. In their work, declarative representations have been proposed for the entire process of plan-execution, -monitoring and -recovery. Similar to our method, their approach is completely formal and works for monitoring arbitrary programs. They addressed the problem of recovering from failure, which we haven't included in our system for the moment. However, their approach must know in advance all exogenous events in order to specify appropriate *Relevant* and *Recover* mechanism. In addition, they don't explore in-depth how to properly define *Relevant* and *Recover*.

To enable generation of plans in dynamic environments, Veloso et al. [VPC98] introduced *Rationale-Based Monitoring* based on the idea of planning as decision making. Rationale-based monitors encode the features that are associated with each planning decision. The method is used for sensing relevant (or potentially relevant) features of the world that likely affect the plan. Moreover, it investigates the balance between sensitivity to changes in the world and stability of the plans. Although their approach provides the planner opportunities to optimize the plans in a dynamic environment during plan generation, in contract with our approach, they haven't studied the issue of execution monitoring.

As the methods mentioned above address the problem of a *single agent* acting in an uncertain environment, the techniques focus on monitoring of environment and verifying plans. While our approach could be directly applied to single agent domains, they need extra work in order to handle monitoring the collaboration of multiple agents.

Monitoring of Multi-Agent Coordination

Teamwork monitoring has been recognized as a crucial problem in multi-agent coordination. Jennings [Jen93] proposed two foundations of multi-agent coordination: *commitments* and *conventions*. Agents make commitments. Conventions are a means of monitoring of the commitments. The monitoring rules, i.e. what kind of information monitored and the way to perform monitoring, are decided by conventions. Jennings illustrates the method by some examples. However, he doesn't investigate how to select such conventions. Different from his idea, our approach avoids the problem of monitoring selectivity.

Myers [Mye99] introduced a continuous planning and execution framework (CPEF). The system's central component is a plan manager, directing the processes of plan-generation, -execution, -monitoring, and -repair. Monitoring of the environment is carried out at all time during plan generation and execution. Furthermore, execution is tracked by the plan

manager by comparing reports of individual actions' outcomes with the temporal ordering relationships of actions. Several types of event-response rule have been concerned: (1)*failure monitors* encode suitable responses to potential failures during plan execution, (2)*knowledge monitors* detect the availability of information required for decision making, and (3)*assumption monitors* check whether assumptions that a given plan relies still hold. The idea of assumption monitors helps early detection of potential problems before any failure occurs, which can also be fulfilled in our system with a different approach. Based upon CPEF, Wilkins et al. presented a system in [WLB03]. The execution monitoring of agent teams is performed based on communicating state information among team members that could be any combination of humans and/or machines. Humans make the final decision, therefore, even if unreliable communications exist, the monitoring performance may not be degraded much with the help of humans experience.

Another interesting monitoring approach in multi-agent coordination is based on *plan-recognition*, by Huber [HD95], Tambe [Tam96], Intille and Bobick [IB99], Devaney and Ram [DR98], Kaminka et al. [KPT01, KT00]. In this approach, an agent's intentions (goals and plans), beliefs or future actions are inferred through observations of another agent's ongoing behavior.

Devaney and Ram [DR98] described the plan recognition problem in a complex multi-agent domain involving hundreds of agents acting over large space and time scales. They use pattern matching to recognize team tactics in military operations. The team-plan library stores several strategic patterns which the system needs to recognize during the military operation. In order to make computation efficient, they utilize representations of agent-pair relationships for team behaviors recognition.

Intille and Bobick [IB99] constructed a probabilistic framework that can represent and recognize complex actions based on visual evidence. Complex multi-agent action is inferred using a multi-agent belief network. The network integrates the likelihood values generated by several visual goal networks at each time and returns a likelihood that a given action has been observed. The network explicitly represents the logical and temporal relationships between agents, and its structure is similar to a naive Bayesian classifier network structure, reflecting the temporal structure of a particular complex action. Their approach relies on all *coordination constraints* among the agents. Once an agent fails, it may not be able to recognize the plans.

Another line of work has been pursued in ISI. Gal Kaminka et al. [KPT01, KT00] developed the *OVERSEER* monitoring system, which builds upon work on multi-agent plan-recognition by [IB99] and [Tam96]. They address the problem of many geographically distributed team members collaborating in a dynamic environment. The system employs plan recognition to infer the current state of agents based on the observed messages exchanged between them. The basic component is a *probabilistic plan-recognition algorithm* which underlies the monitoring of a single agent and runs separately for each agent. This algorithm is built under a Markovian assumption and allows linear-time inference. To monitor multiple agents, they utilize *social knowledge*, i.e. relationships and interactions among agents, to better predict the behavior of team members and detect coordination failures. *OVERSEER* supports reasoning about uncertainty and time, and allows to answer queries related to the likelihood of current and future team plans.

While our objective is (1) to debug *offline* an implemented MAS, and (2) to monitor *online* the collaboration of multiple agents, the plan-recognition approaches described above

mainly aim to inferring (sub-)team plans and future actions of agents. They do not address the MAS debugging issue. Furthermore, we point out that our method might be used in the MAS design phase to support *protocol generation*, i.e., determine at design time the messages needed and their order, for a (simple) agent collaboration. More precisely, possible plans $P = \langle m_1, \ldots, m_k \rangle$ for a goal encode sequences of messages $m_1, \ldots, m_k$ that are exchanged in this order in a successful cooperation achieving the goal. The agent developer may select one of the possible plans, e.g. according to optimality criteria such as least cost, P^*, and program the individual agents to obey the corresponding protocol. In subsequent monitoring and testing, P^* is then the (single) intended plan.

Plan recognition is suitable for various situations: if communication is *not* possible, agents exchanging messages are not reliable, or communications must be secure. It significantly differs from our approach in the following points:

(1) If a multi-agent system has already been deployed, or it consists of legacy code, the plan-recognition approach can do monitoring without modifications on the deployed system. Our method entirely relies on an agent message log file.

(2) The algorithms developed in [KT00] and [DR98] have low computational complexity. Especially the former is a linear-time plan recognition algorithm.

(3) Our model is not yet capable of reasoning about uncertainty, time and space.

(4) In some tasks, agents do not frequently communicate with others during task execution. In addition, communication is not always reliable and messages may be incorrect or get lost.

We believe the first three points can be taken into account in our framework. (1) Adding an agent actions log file explicitly for a given MAS should not be too difficult. (2) While the developed algorithms are of linear complexity, the whole framework needs to deal with uncertainty or probabilistic reasoning which can be very expensive. While our approach is NP-hard in the worst case, we did not encounter any difficulties in the scenarios we have dealt with. (3) Although IMPACT does not yet have implemented capabilities for dealing with probabilistic, temporal and spatial reasoning, such extensions have been developed and are currently being implemented.

Among the advantages of our method are the following:

- Our method can be more easily extended to do *plan repair* than the methods above. Merely Kaminka et al. mentioned the idea of dealing with failure actions.

- The approach we have chosen includes protocol generation in a very intuitive sense relying on the underlying planner while the cited approaches model agent behavior at an abstract level which can not be used to derive intended message protocols directly.

- Since ascertaining the intentions and beliefs of the other agents will result in uncertainty with respect to that information, some powerful means of reasoning under uncertainty are required for the plan recognition method.

9.7 Conclusion

We have described a method to support testing of a multi-agent system, based on monitoring the agents' message exchange using planning methods. This can be seen as a very useful debugging tool for detecting coding and design errors. We also presented some soundness and completeness results for our approach, and touched its complexity.

Our approach works for arbitrary agent systems and can be tailored to any planning formalism that is able to express the collaborative behavior of the *MAS*. We have briefly discussed (and implemented) how to couple a specific planner, $\texttt{DLV}^{\mathcal{K}}$, which is based on the language $\mathcal{K}$, to a particular *MAS* platform, viz. IMPACT. A web page for further information and detailed documentation has been set up (see footnote 43).

There are many extensions to our approach. We mention just some:

(1) Cost based planning: To keep the exposition simple, we have omitted that $\texttt{DLV}^{\mathcal{K}}$'s capabilities of computing admissible plans and optimal plans over optimistic and secure plans which could also be exploited in our monitoring approach. For instance, in the *Gofish* example we might prefer plans where the customer picks up the package herself, which is cheaper than sending a truck. Thus, in the realization of our approach, also economic behavior of agents in a *MAS* under cost aspects can be easily monitored, such as obedience to smallest number of message exchanges or least total communication cost.

(2) Dynamic planning: We assumed an *a priori* chosen collaboration plan for $\mathcal{M}_{log}$ compatibility. This implies $\mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n') \subseteq \mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n)$, for all $n' \geq n \geq 0$. However, this no longer holds if the plan may be dynamically revised. Checking $\mathcal{M}_{log}$ compatibility then amounts to a new planning problem whose initial states are the states reached after the actions in $\mathcal{M}_{log}$.

(3) At the beginning of monitoring, all potentially interesting plans for the goal are generated, and they can be cached for later reuse. We have shown the advantages of this method. However, if a very large number of intended plans exists up front, the method may become infeasible. In this case, we might just check, similar as above, whether from the state reached and the actions in $\mathcal{M}_{log}$, the goal can be reached.

Investing the above issues is part of ongoing and planned future research.

Chapter 10

Comparison With Related Works and Systems

In this chapter, we will on the one hand compare our language $\mathcal{K}^c$ against other planning and action languages and on the other hand compare the proposed logic programming method for planning against previous logic-based approaches to planning. In more detail, in this comparison we will only focus on directly related research concerning the following topics:

- action languages,
- related results on planning complexity,
- other logic-based approaches to planning using answer set programming and SAT-based approaches,
- approaches to conformant planning and planning under incomplete knowledge in general, and
- approaches to optimal planning.

10.1 Comparison with Other Action Languages

As stated above the language $\mathcal{K}$ proposed here builds on earlier work on action languages (cf. Section 2.4). We will now give a detailed overview on how those languages are related to $\mathcal{K}$.

Most importantly, their underlying semantics is not based on knowledge states, so fluents can not be undefined in any state. As a consequence, totality of fluents cannot be expressed in any of the languages $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$, as each fluent is implicitly total, and default negation is not supported. As we have seen, an advantage of representation in $\mathcal{K}$ is the possibility of representing only what we need to know, and of forgetting about superfluous knowledge. The price for this flexibility is that one has to be aware of what knowledge is needed or where to apply "totalization" when encoding a special problem.

Furthermore, none of the languages mentioned explicitly supports action cost.[44]

10.1.0.1 Correspondence to Language $\mathcal{A}$

It is easy to see that the language $\mathcal{A}$ [GL93] roughly corresponds to $\mathcal{K}$ with complete states in which

- all rules r are of the form (3.2) of Section 3.1.1 with $\mathsf{h}(r) \neq \texttt{false}$, $\mathsf{post}^-(r) = \mathsf{pre}^-(r) = \emptyset$, $\mathsf{pre}(r) \subseteq \mathcal{L}^+_{act}$ and $|\mathsf{pre}(r)| = 1$, i.e. $\mathsf{pre}(r)$ consists of a single action.
- all executability conditions e are of the form (3.3) with $\mathsf{pre}^-(r) = \emptyset$ and $\mathsf{pre}(r) \subseteq \mathcal{L}_{fl}$
- all fluents are inertial and in order to ensure completeness, we add initial totality on all fluents.

10.1.0.2 Correspondence to Language $\mathcal{B}$

Language $\mathcal{B}$ [GL98a] can also be mapped to $\mathcal{K}$ by relaxing the restrictions on $\mathcal{A}$ in also allowing rules r of the form (3.2) with $\mathsf{pre}(r) = \emptyset$ additionally, allowing for the formulation of ramifications.

10.1.0.3 Correspondence to Language $\mathcal{A}_K$

The action language $\mathcal{A}_K$ [SB01] is a variation of the language $\mathcal{A}$, developed for incorporating sensing actions and to support reasoning about conditional plans. $\mathcal{A}_K$ provides value, effect, and executability propositions, which correspond to restrictions of initial state constraints, causal rules, and executability conditions in $\mathcal{K}$, respectively, where most noticeably the $\mathsf{post}(\cdot)$-parts are empty and no default negation occurs. Furthermore, $\mathcal{A}_K$ provides knowledge determining propositions of form a `determines` f, which intuitively means that after executing action a, the value of fluent f is known; this corresponds to a conditionalized form of totalization, which can be expressed easily in $\mathcal{K}$. Using this language, particular temporal projection problems to the state reached after executing a conditional plan are considered, namely, whether a fluent (or formula) is known, or whether it is decided, i.e., either known to be true or known to be false. For that, a transition-based semantics for $\mathcal{A}_K$ is developed, both in a 2-valued and 3-valued setting. In the latter, states are modeled as 3-valued interpretations in which fluents can be true, false, or unknown. State transitions are defined in increasingly sophisticated refinements, by taking into account both fluent values which can definitely be derived from effect propositions and which can *possibly* be derived, by an effect proposition whose body is not contradicted by the current state. A fluent literal is kept in or added to the current state only if there is no danger of a possible contradiction; in the worst case, the state is emptied out, and all fluents become unknown.

The view of state transitions in $\mathcal{A}_K$, which aims at handling reasoning by cases in possible worlds, is different from the view in $\mathcal{K}$, where a new knowledge state is determined just by the sanctioned knowledge about the current state, without considering possible world extensions. To model this in (an extension of) $\mathcal{K}$, we might complete the knowledge states

[44] However, to some extent, action costs can be emulated by additive fluents in $\mathcal{C}+$ [GLLT01, GLL+03] or functions in PDDL if supported.

and consider a set of (evolving) knowledge states rather than a single one, and reason about them. This is beyond the current scope of language $\mathcal{K}$, which is conceived for planning in terms of reaching goal states rather than for reasoning about actions in general.

10.1.0.4 Correspondence to Language $\mathcal{C}$

The language $\mathcal{C}$ [GL98b] is the action language which is closest to $\mathcal{K}$ among those discussed in Section 2.4. In $\mathcal{C}$ not all fluents are automatically inertial – just as in $\mathcal{K}$ it must be explicitly declared if a fluent has the property of being inertial. As in $\mathcal{K}$, this is achieved by a macro `inertial F.` which is defined in $\mathcal{C}$ as `caused F if F after F.` whereas in $\mathcal{K}$ it is defined as `caused F if not -F after F.` Furthermore, $\mathcal{C}$ has (like $\mathcal{K}$) a macro `default F.` for declaring that a property holds by default. In $\mathcal{C}$, it stands for `caused F if F`, while in $\mathcal{K}$, it is defined as `caused F if not -F.` The difference in macro expansion is due to the slightly different semantic definition of causation (discussed in Subsection 10.1.0.4 below) and also due to the lack of default negation in $\mathcal{C}$. In particular, `default F.` means in $\mathcal{C}$ that `F` is true without the need of further causal support. Finally, $\mathcal{C}$ also provides a way to specify nondeterministic action effects.

Despite those differences, there is a principal fragment of $\mathcal{C}$ action descriptions which correspond to similar $\mathcal{K}$ action descriptions, and allow to semantically embed this fragment of $\mathcal{C}$ efficiently into $\mathcal{K}$. Namely, any propositional definite $\mathcal{C}$ action description $AD_\mathcal{C}$, i.e, set of causal rules having only fluent literals in the heads, where rule bodies are conjunctions of fluent literals is semantically equivalent to the $\mathcal{K}$ action description $AD_\mathcal{K} = tr_\mathcal{K}(AD_\mathcal{C})$ which contains:

(i) fluent and action declarations, for each fluent symbol f and action symbol a in $AD_\mathcal{C}$, respectively;

(ii) `executable` a, for every action symbol a in $AD_\mathcal{C}$, i.e. all actions are executable;

(iii) `initially total` f, for each fluent symbol f in $AD_\mathcal{C}$;

(iv) a causation rule `caused l if not` $\neg$`.`$\mathtt{b}_1$`,...,not` $\neg$`.`$\mathtt{b}_k$ `after` H, for every rule `caused l if` $\mathtt{b}_1$`,...,`$\mathtt{b}_m$ `after` H in $AD_\mathcal{C}$; and,

(v) a constraint `forbidden not` f, `not` $\neg f$, for every fluent symbol f in $AD_\mathcal{C}$.

The fluent declarations in (i) and executability conditions in (ii) are required by the conventions of $\mathcal{K}$. The statements in (iii) effect $\mathcal{C}$'s exogenous assignment of values to the fluents in the initial state, which are exempted from causation (but must comply with all static rules); the legal initial states of $AD_\mathcal{K}$ and $AD_\mathcal{C}$ coincide. The rewriting of the causation rules in (iv) serves to emulate $\mathcal{C}$'s notion of causation, while the constraints in (v) enforce completeness of a state. The mapping $tr_\mathcal{K}(\cdot)$ amounts, apart from minor variations, via the translation of $\mathcal{K}$ to answer set programming (see Chapter 4) to the translation of the above fragment of language $\mathcal{C}$ to answer set programming given in [LT99]. From the results in [LT99], we thus obtain the following correspondence:

Proposition 10.1. *For any complete state s, the legal state transitions $\langle s, A, s' \rangle$ in the planning domain $\langle \emptyset, tr_\mathcal{K}(AD_\mathcal{C}) \rangle$ correspond 1-1 to the causally explained (i.e. possible) transitions from s to complete state s' in $AD_\mathcal{C}$ executing the actions in A.*

The above translation can be easily generalized to arbitrary definite $\mathcal{C}$ action descriptions $AD_{\mathcal{C}}$, in which bodies of causal rules r : `caused f if` E `after` H may be arbitrary propositional expressions E. By using disjunctive normal forms $E = E_1 \vee \cdots \vee E_n$ and $H = H_1 \vee \cdots \vee H_m$, we can easily split up r into an equivalent set of rules $r_{i,j}$: `caused f if` E_i `after` H_j, $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,m\}$. While this transformation is, due to disjunctive normal form conversion, exponential in general, we remark that by the use of auxiliary fluents for labeling subexpressions of E and H in a standard way, one can polynomially translate any definite $\mathcal{C}$ action description into a $\mathcal{K}$ action description which is equivalent modulo the auxiliary fluents. Thus, in summary, planning in $\mathcal{C}$ using definite action descriptions is naturally and efficiently embeddable into $\mathcal{K}$. We can view this syntactic class of $\mathcal{C}$ as a semantic fragment of $\mathcal{K}$, and any $\mathcal{K}$ planning system can be easily utilized for planning in it as well.

On the other hand, $\mathcal{K}$ action descriptions seem not amenable to a simple translation into $\mathcal{C}$. The reason is a semantic difference between the notion of causation in $\mathcal{C}$ and in $\mathcal{K}$, which is a consequence of a stronger foundedness principle for causation that is implemented in $\mathcal{K}$, and is in analogy to minimal models versus supported models of a logic program. In $\mathcal{K}$, only transitions between states are legal which are "foundedly supported" by the respective causation rules; in more detail, any causation of a fluent must, by starting from unconditional facts, be derivable by applying causation rules which are recursively founded. On the other hand, $\mathcal{C}$ defines causally explained transitions where supportedness but no minimality aspects play a role. This is exemplified by the encoding of a default `caused` f `if` f. considered above. In $\mathcal{C}$, the state $\{f\}$ is causally explained by this rule, while it is not in $\mathcal{K}$: f is concluded from the assumption of its truth "by default;" using negation as failure, this is more familiarly expressed in $\mathcal{K}$ by `not` $-f$. Since $\mathcal{C}$ adheres in spirit to supported models and $\mathcal{K}$ to minimal models, encoding $\mathcal{K}$ action descriptions in $\mathcal{C}$ is obviously more involved: For instance, expressing transitive closure of a graph is simple in $\mathcal{K}$, while is more involved in $\mathcal{C}$, which is shown by the example in Section 6.6.1: The two $\mathcal{K}$ rules in the example are not sufficient for describing causally explained state transitions in $\mathcal{C}$, as negative knowledge has to be given explicitly for instance by adding a rule

```
default -above(X,Y).
```

in order to guarantee for causally explained transitions while we can just leave the negative information open in $\mathcal{K}$. Lee and Lifschitz describe the extension $\mathcal{C}+$ of the action language $\mathcal{C}$ which allows for an intuitive encoding of resources and costs by means of so called "additive fluents" [LL01]. By constraining the values of these fluents, admissible planning can be realized to some extent, but optimization has not been considered in that framework so far. For a more detailed discussion on new features of the enhanced language $\mathcal{C}+$ [GLLT01, GLL+03] such as multi-valued and additive fluents we refer to the respective remarks in Sections 6.5.1 and 7.1.3 above.

10.1.0.5 Correspondence to Languages STRIPS, ADL, and PDDL

With the above mentioned correspondence to language $\mathcal{A}$ in mind, STRIPS [FN71] can analogously be embedded into $\mathcal{K}$ by similar considerations. (Remember that $\mathcal{A}$ is in fact as expressive as propositional ADL [Ped89], i.e. an extension of pure STRIPS.) Such an en-

coding is shown in detail in the proof of Theorem 3.3 in Section 3.3. The extension by conditional effects like in ADL is straightforward.

The last decade has witnessed a dramatic progress in the variety and performance of techniques and tools for classical planning. The existence of a de-facto standard modeling language for classical planning, PDDL [GHK+98], has played a relevant role here. PDDL significantly differs from the above-mentioned languages in that PDDL stands for a modular family of languages rather than a single clearly defined language, which is defined by so called *requirements*, Any system accepting PDDL might or might not implement these requirements.

Clearly, as mentioned above $\mathcal{K}$ is capable of expressing the STRIPS and ADL fragments of PDDL. Furthermore, $\mathcal{K}$ features typing of variables in a comparable way and many other PDDL requirements of Version 1.2 such as domain axioms, disjunctive preconditions of actions, quantified preconditions, etc. [GHK+98] can be emulated in $\mathcal{K}$. Arithmetic expression evaluation in PDDL can to some extent also be emulated in $\mathcal{K}$ with the restrictions of $\texttt{DLV}^{\mathcal{K}}$ integer arithmetics. However, compound tasks as defined with the `:action-expansions` in PDDL are beyond the scope of $\mathcal{K}$. The techniques proposed by Dix et al. [DKN02a] to encode Hierarchical Task Network (HTN) Planning in Answer Set Programming might serve as a starting point here.

We remark that the `:open-world` and `:true-negation` requirements in PDDL version 1.2, which however have been removed in the current version (2.1), have a natural representation in $\mathcal{K}$ where the user can flexibly decide whether CWA should be applied or not for a particular fluent.

Meanwhile, PDDL appears in version 2.1 [FL03] which adds additional levels introducing for instance durative actions, continuous and/or conditional effects, etc. This is currently not expressible in $\mathcal{K}^c$ in a straightforward way, and needs to be further investigated.

Still, action languages like $\mathcal{K}^c$ often offer a more flexible framework for modeling actions and transitions than the strict framework of PDDL; we conceive the rule based, almost natural language concept behind action languages in order to describe transitions more general and flexible than the strict operator-based syntax of PDDL (and its predecessors). Furthermore, incomplete knowledge and nondeterminism are not addressed in the current version of PDDL. However, one has to bear in mind the different objectives of these languages. Based on ADL, PDDL originally has been designed as a domain specification language for the International Planning Competition (IPC). PDDL is conceived to serve as a generic language representing the features of various special-purpose planners. The further development of the language is driven by the IPC and extensions are made very cautiously in order to provide a widely accepted standard. The strict corset of an operator-based STRIPS or PDDL like syntax has without doubt the advantage of allowing for easier structural analysis of the domain at hand, wrt. optimizations and planning specific heuristics as opposed to the more general concept of action languages like $\mathcal{K}$ or $\mathcal{C}$.

Remarkably, also steps in the directions of extensions for incomplete initial state specifications and nondeterminism have been recently made in the last "Workshop on PDDL" held at the "International Conference on Automated Planning and Scheduling (ICAPS'03)": For instance, the language NPDDL [BCLP03], the input language of the successful MBP [BCPT01] planner includes such extensions. Interestingly, NPDDL also includes an extended formalism for expressing goals specified in a Computation Tree Logic (CTL) [CE82] like manner.

10.2 Related Results on Planning Complexity

Our results on the complexity of planning in $\mathcal{K}$ are related to several results in the planning literature. First and foremost, planning in STRIPS can be easily emulated in $\mathcal{K}$ planning domains, and thus results for STRIPS planning carry over to respective planning problems in $\mathcal{K}$, in particular Optimistic Planning, which by the results in [Byl94, ENS95] is PSPACE-complete.

As for finding secure plans (alias conformant or valid plans), there have been interesting results in the recent literature. Turner [Tur02] has analyzed in a recent paper the effect of various assumptions on different planning problems, including conformant planning and conditional planning under domain representation based on classical propositional logic. In particular, Turner reports that deciding the existence of a classical (i.e. optimistic) plan of polynomial length is NP-complete, and NP-hard already for length 1 where actions are always executable. Furthermore, he reports that deciding the existence of a conformant (i.e. secure) plan of polynomial length is Σ_3^P-complete, and Σ_3^P-hard already for length 1. Furthermore, the problem is reported Σ_2^P-complete if, in our terminology, the planning domain is proper, and Σ_2^P-hard for length 1 in deterministic planning domains. Turner's results match our complexity results, announced in [EFL$^+$00a]; this is intuitively sound, since answer set semantics and classical logic, which underlies ours and his framework, respectively, have the same computational complexity.

Enrico Giunchiglia et al. [CGT02] considered conformant planning in the action language $\mathcal{C}$, where concurrent actions, constraints on the action effects, and nondeterminism on both the initial state and effects of the actions are allowed – all these features are provided in our language $\mathcal{K}$ as well. In their approach they use SAT solvers for computing, in our terminology, optimistic and secure plans following a two step approach. For this purpose, transformations of finding optimistic plans and security checking into SAT instances and QBFs are provided. The same approach is studied in [FG00] for an extension of STRIPS in which part of the action effects may be nondeterministic. While not explicitly analyzed, the structures of the QBFs emerging in [FG00, CGT02] reflect our complexity results for Optimistic Planning and Security Checking.

Haslum and Jonsson [HJ99] showed that conformant planning in a closely related formalism, where no plan length is given in the input, is in general EXPSPACE-complete, which reflects the result of Theorem 3.8. Note that we obtain a proof by a generic reduction from Turing machines to conformant planning, rather than by a reduction from the universality problem for regular expressions with exponentiation as in [HJ99]. Rintanen [Rin03] describes a reduction similar to ours and derives, by further generalizations, complexity results for planning with partial observability.

From our results on the complexity of planning in the language $\mathcal{K}$, similar complexity results may be derived for other declarative planning languages, such as STRIPS-like formalisms as in [Rin99a] and the language $\mathcal{A}$ [GL93], or the fragment of $\mathcal{C}$ restricted to causation of literals (cf. [Giu00]), by adaptations of our complexity proofs. The intuitive reason is that in all these formalisms, state transitions are similar in spirit and have similar complexity characteristics. In particular, our results on Secure Planning should be easily transferred to these formalisms by adapting our proofs for the appropriate problem setting.

As for optimal optimistic and secure planning, we are not aware of previous in-depth complexity investigations in related formalisms, but our results for $\mathcal{K}^c$ may analogously

map to optimizations wrt. to particular fluents for instance in functional STRIPS [Gef00], where numerical fluent values are allowed. Note that, however complexity results for finding optimal plans in nondeterministic domains slightly change in case costs do not only depend on actions but also on fluent values in the current state. For further details on this topic, we refer to Section 7.1.4 below, where such extensions for $\mathcal{K}^c$ are discussed.

10.3 Answer Set Planning – Previous Approaches

One of the main targets of this work was defining transformations to logic programming for planning problems encoded in our action language $\mathcal{K}$.

These transformations allowed for solving planning problems directly be use of existing answer set solvers such as DLV, GNT, or SMODELS such that the answer sets of the generated logic programs correspond to plans. This idea is not completely novel; logic programming and answer set programming in particular has been widely accepted as a useful tool for solving classical planning problems by means suitable transformations, cf. [Lif02, Bar03]. Some previous approaches for using answer set programming in the area of planning and more general for reasoning about actions and change shall be shortly reviewed in this section and compared against our contributions.

First approaches to formalize actions and planning in logic programming date back to the early 90's. Action language $\mathcal{A}$ has served as a basis for several previous approaches of encoding reasoning about actions by means of logic programming techniques: Gelfond and Lifschitz [GL93] expressed the semantics of their language $\mathcal{A}$ in terms of answer set semantics by means of a translation to extended (normal) logic programs with function symbols, where their translation resembles the Situation Calculus representation of states by using function $res(Action, Situation)$. This work employs a convenient non-monotonic formulation of frame axioms using negation as failure, but their translation is rather meant to justify the semantics of $\mathcal{A}$ than for solving planning problems as such. Due to the use of function symbols, this approach is not directly applicable in current Answer Set Programming systems which only allow for function-free (Datalog) programs, such as DLV. Although SMODELS allows for limited use of function symbols (cf. [Syr02]), it does not accept for full, unbound recursion, by imposing domain restriction on any used variable in order to keep the ground program finite (cf. Section 2.1.5.2). Therefore, the translation proposed in [GL93] is inapplicable for SMODELS, either. In this context, we remark that Bonatti [Bon01b] recently proposed subclasses of programs with function symbols, so-called *finitary* programs, where ground queries are decidable, even if the universe of the program is infinite. In [Bon01a] also a prototype implementation of this approach is described and investigating the applicability of this method for particular translations of planning problems might be worthwhile.

The seminal works by Subrahmanian and Zaniolo [SZ95] first related logic programs under the answer set semantics to STRIPS-like planning problems, where plans exactly correspond to the answer sets of their logic programming encodings. In these encodings, states are not encoded by function symbols like in Situation Calculus based encodings, but associated with time stamps, i.e. each fluent/action predicate is augmented with an additional timestamp parameter over a finite time range like in our approach. This idea is borrowed

from the planning as SAT approach [KS92] and fundamental for our translations. However, this restricts planning to fixed plan length. We remark that similar ideas for using time-stamped predicates in Datalog to model states and change have also been proposed by Lausen and Ludäscher [LL94] in the context of database updates.

Dimopoulos et al. [DNK97] discuss similar encodings and suggest several possible optimizations. These encodings have been further refined and simplified by later works:

For instance, [Nie99] discusses similar encodings for blocks world problems tailored for SMODELS.

The term "Answer Set Planning" has actually been introduced by Lifschitz et al. [LT99, Lif99a, Lif99b, EL99, Lif02] who have extended these encodings to handle more complex planning languages such as action language $\mathcal{C}$.

Finally, Leone et al. [LRS01] have first shown some integrated ad hoc encodings for conformant planning and simple forms of conditional planning. These encodings can be considered as a starting point for the general method shown in Section 4.3.5.

We have built upon the above-mentioned work for our basic translation, particularly on the translations proposed by Lifschitz et al. [LT99, Lif99a] and extended them with respect to planning under uncertainty, incomplete information, and action costs, adopting useful concepts of Answer Set Programming such as negation as failure and weak constraints. In particular, optimal planning by exploiting advanced capabilities of solvers like DLV and SMODELS has to our knowledge not been addressed in Answer Set Programming based approaches before. Furthermore, we have discussed how to remedy the restrictions to fixed plan length under certain circumstances in order to compute shortest plans.

Son and Pontelli [SP02] propose to translate action language $\mathcal{B}$ to prioritized default theory and answer set programming. They allow to express preferences between actions and rules at the object level in an interpreter, but not as a part of the input language. However, these preferences are orthogonal to our approach as they model qualitative preferences as opposed to our overall cost-based value function for plans/trajectories.

We remark here that also Hierarchical Task Network (HTN) planning has recently been formulated in Answer Set Programming by Dix et al. [DKN02a] which, however, has not been covered here.

10.4 Planning as Satisfiability

Closely related to our approach and preliminary to it is the method of Planning as Satisfiability introduced by Kautz and Selman [KS92]. There, (STRIPS-like) planning problems are translated to SAT instances, where also models correspond to plans, similar to the Answer Set Programming approach. Improvements and comparison against classical planning techniques have been suggested by Brafman [Bra01].

The Blackbox planning system [KS99], or CPLAN [Giu00, FG00, CGT02] mark further advances in the Planning as SAT paradigm. CPLAN even deals with conformant planning in nondeterministic domains. While direct encodings of conformant planning problems to SAT in polynomial time are infeasible (cf. Section 3.3), they use an interleaved approach similar to $\text{DLV}^{\mathcal{K}}$, where guessing the plan and checking security are interleaved by separate calls to a SAT-solver.

Also CCALC [McC99a], the causal calculator, which allows for optimistic planning and also more general reasoning about actions in the action language $\mathcal{C}+$, relies on reductions to SAT, via the "detour" of translations to causal theories [MT97, MT98]. As mentioned above, by the features of the language $\mathcal{C}+$ admissible planning can be performed and CCALC is also capable of iteratively computing shortest plans. Optimization of arbitrary criteria is not possible within CCALC. and conformant planning is not yet supported.

The general restriction to bounded plan length applies to Planning as Satisfiability approaches as well, in that underlying SAT solvers can also only deal with a finite structures.

10.5 Planning under Incomplete Knowledge

Planning under incomplete knowledge has been widely investigated in the AI literature. Most works extend algorithms/systems for classical planning, rather than using deduction techniques for solving planning tasks as proposed in our work. The systems Buridan [KHW95], UDTPOP [Peo98], SGP [WAS98], CNLP [PS92], GPT [BG00], CASSANDRA [PC96], and Conformant-FF[BH03] fall in this class. In particular, Buridan, UDTPOP, GPT and SGP can solve secure planning (also called conformant planning), like DLV$^{\mathcal{K}}$. On the other hand, the systems CNLP and CASSANDRA deal with conditional planning (where the sequence of actions to be executed depends on dynamic conditions). While extension to conditional planning are an interesting perspective also for our future work, in this comparison we want to focus on systems and approaches for conformant planning. Here, particularly SGP, GPT and Conformant-FF, as the most recent of those systems should be mentioned:

SGP is an extension of the Graphplan algorithm [BF97]. Its input language is an extension of PDDL. Nondeterminism is allowed only in the initial state. By definition, the Graphplan algorithm allows for parallel actions, in particular, parallelizes serializable actions automatically. Furthermore, SGP can also deal with sensing actions.

GPT on the other hand employs heuristic search techniques like A^* to search the belief space. Its input language is also an extension of PDDL. Nondeterminism is allowed in the initial state as well as for action effects. GPT supports sequential planning and calculates plans of minimal length. The system offers apart from conformant planning several notable features such as (conditional) planning under partial observability, or probabilistic uncertainty.

A recent new conformant planner is conformant-FF [BH03] by Brafman and Hoffmann. In the current version, it does not allow for nondeterminism apart from the initial state, but the authors claim that the extension is straight-forward. They report that checking security is co-NP in their formalism, which also reflects our complexity results wrt. their less expressible ADL-based input language. This allows them to employ SAT methods for security checking which they interleave with a classical planning algorithm. Here, they also rely on heuristic search methods, building up on the successful FF planner [HN01].

Recent works propose the use of automated reasoning techniques for planning under incomplete knowledge. The above-mentioned CPLAN falls in this class. In [Rin99a] a technique for encoding conditional planning problems in terms of 2-QBF formulae is proposed. The work in [FPR00] proposes a technique based on regression for solving secure planning problems in the framework of the Situation Calculus, and presents a Prolog implementation of such a technique. In [MT98], sufficient syntactic conditions ensuring security of every

(optimistic) plan are singled out. While sharing their logic-based nature, our work presented differs considerably from such proposals, since it is based on a different formalism.

Over the last years, especially methods from the field of model-checking have successfully been applied to classical planning and also planning under incomplete knowledge. The model checking paradigm exploits symbolic Boolean function representation techniques such as Binary Decision Diagrams [Bry86] for compactly representing belief states and transitions between them. Two successful planners in this area provide similar features: MBP [BCPT01] and UMOP [JV00], which both accept nondeterministic extensions of PDDL as input language. These planners allow for strong (conformant planning) planning, strong cyclic planning (conditional planning with loop-constructs) and optimistic planning. MBP is the successor of the CMBP planner mentioned in our experiments (cf. Chapter 8). It extends CMBP by allowing for conditional planning and furthermore offers a very expressive goal description language based on Computation Tree Logic (CTL) [CE82]. UMOP also implements strong cyclic adversarial planning and optimistic adversarial planning [JVB01]. Adversarial Planning takes additional environmental actions into account, which can be seen as exogenous, concurrent actions of other agents in a multi-agent environment.

10.6 Planning with Action Costs

The PYRRHUS system [WH94] is an extension of UCPOP planning which allows for optimal planning with resources and durations. Domain-dependent knowledge can be added to direct the heuristic search. A "utility model" has to be defined for a planning problem which can be used to express an optimization function. This system supports a language extension of ADL. The algorithm is a synthesis of branch-and-bound optimization with a least-commitment, plan-space planner.

Other approaches based on heuristic search include the use of an A* strategy together with action costs in the heuristics [EPM96] and work by Refanidis and Vlahavas who use multi-criteria heuristics to obtain near-optimal plans, considering multiple criteria apart from plan length alone [RV01]. However, the described heuristics is not fully admissible, and only guarantees optimal plans under certain restrictions [HG00].

In fact, most heuristic state-space planners are not able to guarantee optimality. Whereas strict cost and also resource bounds (admissible planning in our terminology) is expressible in all planners which allow for numeric domain axioms in some form, systems which actually compute optimal plans are rare since many planners only aim at finding "fairly good" solutions. This applies to most planners capable of expressing numeric functions in PDDL, such as LPG [GSS03], Sapa [DK03], Metric-FF [Hof03], and MIPS [Ede03], which participated in the numeric track of the 3rd International Planning Competition [ea02]. All these planners are discussed in detail in an upcoming special issue of the "Journal of Artificial Intelligence Research (JAIR)".

A powerful approach has been suggested by Nareyek, who describes planning with resources as a structural constraint satisfaction problem (SCSP), and then solves that problem by local search combined with global control. However, this work promotes the inclusion of domain-dependent knowledge; the general problem has an unlimited search space, and no declarative high-level language is provided [Nar01].

Among other related approaches, Kautz and Walser generalize the "Planning as Satisfia-

bility" approach to use integer optimization techniques for encoding optimal planning under resource production/consumption [KW99]. First, they recall that integer logic programming generalizes SAT, as a SAT formula can be translated to a system of inequalities. Second, they extend effects and preconditions of actions similar to a STRIPS extension proposed by Köhler for modeling resource consumption/production [Koe98]. Kautz and Walser allow for arbitrary optimization functions but they use a non-declarative, low-level representation based on the algebraic modeling language AMPL [FGK93]. They mention that Köhler's STRIPS-like formalization can be mapped to their approach. However, they can not express nondeterminism or incomplete knowledge. There is an implementation of this approach called ILPPLAN, which uses the AMPL package.[45] Although this approach seems to be flexible enough to express optimization problems like the ones we have shown, AMPL is not freely available, so we could not compare the system with our approach experimentally.

[45] <URL:http://www.ampl.com/>)

Chapter 11

Conclusions and Outlook

The main results of this thesis are twofold. On the one hand, we showed the feasibility of building a planning system with advanced features based on Answer Set Programming, an evolving paradigm in logic programming. On the other hand, when looking at existing planning languages, we came to the conclusion that, especially for planning with incomplete information and nondeterminism, logic programming features are valuable within action languages.

To this end, we have come up with the novel action language $\mathcal{K}^c$. It includes features borrowed from logic programming such as a distinction between "negation as failure" and "classical" negation of state variables (i.e. fluents). The benefit of these additional language features has been explored by means of illustrative examples from several well-known but also novel planning domains, where our language allows for a concise declarative formulation of such problems.

We have shown how to build a planning system based on existing Answer Set Programming solvers by reasonable transformations of planning problems to logic programs. Planning in general is PSPACE-hard, even for STRIPS-like domains. However, when assuming restrictions such as constant upper bounds for the plan length which may be safely assumed in many domains, we showed that several planning problems can be decently solved employing such translations.

In this context, our approach may be seen in close relation to "Planning as Satisfiability" (coined by Kautz and Selman), where planning problems are transformed to propositional satisfiability problems which can be solved by existing, efficient SAT checkers. SAT solvers had been developed already for years before the first ASP solvers appeared and clearly have the advantage of long-term fundamental research on algorithms and heuristics. Nevertheless, ASP solvers such as DLV and SMODELS can already compete performance-wise on particular problems, last but not least profiting from the heuristics developed by the SAT community (cf. [Sim00, Fab02]). ASP solvers are steadily improving and often have the edge in terms of knowledge representation and ease of prototyping: Action languages such as $\mathcal{K}^c$ or $\mathcal{C}$ describe state transitions by causation rules and our translations to ASP, which reflect this rule-based structure per se, seem more intuitive than translations to SAT here.

In analogy to the term "Planning as Satisfiability" our approach may well be conceived as "Planning as Answer Set Programming" or even "Answer Set Programming as Planning",

to some extent.

Whereas our translations of classical (optimistic) planning build up on previous approaches, the translations introduced for conformant (secure) planning problems and optimal planning are completely novel and have – to our knowledge – not been tackled with logic programming techniques so far. Furthermore, we have shown that such translations are not feasible by reductions to SAT. Moreover, the general method we have deployed in this context can be applied to a variety of other hard problems as well.

Besides, we have shown the applicability of our method for modeling a problem in the area of design and monitoring of multi-agent-systems. We remark that the newly introduced monitoring approach demonstrated on the basis of this application may be generalized to other planning frameworks as well.

Outlook and Open Issues

The research in Action Languages, Planning and Answer Set Programming is steadily evolving and there is a number of interesting parallel research directions in these fields of which we have only explored a few in this work. For instance, we have pinpointed to several possible language extensions in Chapter 7. Particularly, we mention dynamic (i.e. state-dependent) action costs. It seems that there is no agreement yet upon how to deal with the combination of numeric costs of a plan and uncertainty in the planning community. An open question to be answered is the following: Which plans are interesting in this context? On the one hand, we can employ a cautious view of expecting worst-case scenarios, where everything goes wrong. However, also different measures such as plans which have least average costs or any other cost aggregation might be interesting in specific scenarios and application domains.

Concerning incomplete information and nondeterminism, we have only considered conformant planning in the course of this work, i.e., plans which reach the goal under any contingencies. This is quasi computing a plan which may be executed "blindfold", with no observability. However, whereas such plans do not always exist, there may be conditional solutions, where the agent branches on results of current observations. Apart from that, exogenous, unforeseen events might change the environment during plan execution, invalidating pre-computed plans. In this context, we think that considering methods of diagnosis [EFLP99, EFL$^+$00b], belief revision [EFST01b] and knowledge base updates [EFST01a, EFST02], which have already been investigated in the context of Answer Set Programming, might be worthwhile. Combined with our research in Answer Set Planning, we believe that these methods can be exploited in a general framework for monitoring, reactive planning, and re-planning. Here, our monitoring approach discussed in Chapter 9 is just a starting point.

As we have seen, there may also be further potential in fundamental research on Answer Set Programming itself. We conceive our proposed method for integrating separate programs as an initial step for further research on automatic integration of "guess" and "check" encodings which exploit the full potential of disjunctive logic programming: Integrated encodings like those we have considered are infeasible in less expressive frameworks such as propositional SAT solving or normal logic programming. However, while ad hoc encodings for specific problems have already been available, general methods in this direction were still missing. Several issues remain for further work. Our rewriting method currently applies to

propositional programs. Thus, before transformation, the program should be instantiated. A more efficient extension of the method to non-ground programs is needed, as well as further improvements to the current transformations. Experimental results suggest that structural analysis of the "guess" and "check" programs might be valuable for this purpose. A further issue are alternative transformations, possibly tailored for certain classes of programs. We strongly believe that there is room for further optimizations and improvements both on the translation and for the underlying DLV engine.

Bibliography

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Minker [Min88], pages 89–148.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[Bab02] Yulia Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[Bar03] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[BCLP03] Piergiorgio Bertoli, Alessandro Cimatti, Ugo Dal Lago, and Marco Pistore. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. In *Proceedings of ICAPS'03 Workshop on PDDL*, Trento, Italy, June 2003.

[BCPT01] Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, and Paolo Traverso. MBP: A model based planner. In Alessandro Cimatti, Hector Geffner, Enrico Giunchiglia, and Jussi Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, August 2001.

[BDH99] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[BED94] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[BEL00] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming. In *Proceedings of the Eighth International Workshop on Non-Monotonic Reasoning*, 2000.

[BF97] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90:281–300, 1997.

[BG00] Blai Bonet and Héctor Geffner. Planning with Incomplete Information as Heuristic Search in Belief Space. In Steve Chien, Subbarao Kambhampati, and

Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 52–61, Breckenridge, Colorado, USA, April 2000.

[BG01] Blai Bonet and Héctor Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[BH03] Ronan I. Brafman and Jörg Hoffmann. Conformant planning via heuristic forward search. In *Proceedings of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, pages 8–17, Trento, Italy, June 2003.

[BLR97] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[BLR00] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[BMSU86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. Int. Symposium on Principles of Database Systems*, pages 1–16, 1986.

[Bon01a] Piero A. Bonatti. Prototypes for Reasoning with Infinite Stable Models and Function Symbols. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 416–419. Springer, 2001.

[Bon01b] Piero A. Bonatti. Reasoning with Infinite Stable Models. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 603–610, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[Bra01] Ronen I. Brafman. On reachability, relevance and resolution in the planning as satisfiability approach. *Journal of Artificial Intelligence Research*, 14:1–28, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS97] Roberto Bayardo and Robert Schrag. Using CSP Look-back Techniques to Solve Real-world SAT Instances. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[Byl94] T. Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69:165–204, 1994.

[Cad92] Marco Cadoli. The Complexity of Model Checking for Circumscriptive Formulae. *Information Processing Letters*, 44:113–118, 1992.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1982.

[CEG97] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[CGGT97] Alessandro Cimatti, Enrico Giunchiglia, Fausto Giunchiglia, and Paolo Traverso. Planning via Model Checking: A Decision Procedure for AR. In *Proceedings of the 4th Eutopean Conference on Planning (ECP-97)*, pages 130–142. Springer Verlag, 1997.

[CGS97] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. Experimental analysis of the computational cost of evaluating quantified boolean formulae. In *Proceedings of the 5th Congress of the Italian Association for Artificial Intelligence (AI*IA 97)*, number 1321 in LNCS, pages 207–218, Rome, Italy, 1997. Springer.

[CGS98] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings AAAI/IAAI-98*, pages 262–267, 1998.

[CGT02] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. SAT-Based Planning in Complex Domains: Concurrency, Constraints and Nondeterminism. *Artificial Intelligence*, ?, 2002. To appear.

[CR99] Alessandro Cimatti and Marco Roveri. Conformant Planning via Model Checking. In Susanne Biundo and Maria Fox, editors, *Proceedings of the Fifth European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Computer Science*, pages 21–34, Durham, UK, September 1999.

[CR00] Alessandro Cimatti and Marco Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.

[CRT98] Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceedings National Conference on AI (AAAI '98)*, pages 875–881, Madison, Wisconsin, July 26–30 1998. AAAI Press.

[DEF+03] Jrgen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang. Monitoring agents using planning. To appear in Proc. of the 26th German Conference on Artificial Intelligence (KI2003), 2003.

[DEGV97] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. In *Proceedings of the Twelfth Annual IEEE conference on Computational Complexity, June 24–27, 1997, Ulm, Germany, (CCC'97)*, pages 82–101. Computer Society Press, June 1997. Extended paper in *ACM Computing Surveys*, 33(3), September 2001.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[DFI+03] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.

[DHW94] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic Planning with Information Gathering and Contingent Execution. In Kristian J. Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 31–36. AAAI Press, June 1994.

[DK03] Minh B. Do and Subbarao Kambhampati. A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research*, Special Issue on the Third International Planning Competition, 2003. to appear.

[DKN02a] Jürgen Dix, Ugur Kuter, and Dana Nau. HTN Planning in Answer Set Programming. Technical Report CS-TR-4332 (UMIACS-TR-2002-14), Dept. of CS, University of Maryland, College Park, MD 20752, February 2002. To appear in *Theory and Practice of Logic Programming.*

[DKN02b] Jürgen Dix, Ugur Kuter, and Dana Nau. Planning in Answer Set Programming using Ordered Task Decomposition. *Journal of the Theory and Practice of Logic Programming*, October 2002. Revised version under submission. Short paper to appear in KI 2003 (German National Conference on Artificial Intelligence).

[DMANZ02] Jürgen Dix, Hector Munoz-Avila, Dana Nau, and Lingling Zhang. Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA '02)*, LNCS 2424, pages 173–185. Springer, 2002.

[DNK97] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *Proceedings of the European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer Verlag, 1997.

[DR98] Mark Devaney and Ashwin Ram. Needles in a haystack: Plan recognition in large spatial domains involving multiple agents. In *AAAI/IAAI*, pages 942–947, 1998.

[DST01] Jim Delgrande, Torsten Schaub, and Hans Tompits. plp: A Generic Compiler for Ordered Logic Programs. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 411–415. Springer, 2001.

[Dun92] Phan Minh Dung. On the Relations between Stable and Well-Founded Semantics of Logic Programs. *Theoretical Computer Science*, 105(1):7–25, 1992.

[ea02] Derek Long et al. The third international planning competition, 2002. `http://www.dur.ac.uk/d.p.long/competition.html`.

[Ede03] Stefan Edelkamp. Taming numbers and duration in the model checking integrated planning system. *Journal of Artificial Intelligence Research*, Special Issue on the Third International Planning Competition, 2003. to appear.

[EFL+00a] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under Incomplete Knowledge. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, Proceedings*, number 1861 in Lecture Notes in AI (LNAI), pages 807–821, London, UK, July 2000. Springer Verlag.

[EFL+00b] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Using the `dlv` System for Planning and Diagnostic Reasoning. In François Bry, Ulrich Geske, and Dietmar Seipel, editors, *Proceedings of the 14th Workshop on Logic Programming (WLP'99)*, pages 125–134. GMD – Forschungszentrum Informationstechnik GmbH, Berlin, January 2000. ISSN 1435-2702.

[EFL+01a] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. System Description: The DLV$^{\mathcal{K}}$ Planning System. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 413–416. Springer Verlag, September 2001.

[EFL+01b] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. The DLV$^{\mathcal{K}}$ Planning System. In Alessandro Cimatti, Héctor Geffner, Enrico Giunchiglia, and Jussi Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 76–81, August 2001.

[EFL+02a] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer Set Planning under Action Costs. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, number 2424 in

Lecture Notes in Computer Science, pages 186–197, Cosenza, Italy, September 2002.

[EFL+02b] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. The DLV$^{\mathcal{K}}$ Planning System: Progress Report. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, number 2424 in Lecture Notes in Computer Science, pages 541–544, Cosenza, Italy, September 2002. (System Description).

[EFL+03a] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1–2):157–211, March 2003.

[EFL+03b] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. To appear in ACM Transactions on Computational Logic, 2003.

[EFL+03c] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer Set Planning under Action Costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2003.

[EFLP99] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[EFLP03] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. *Journal of the Theory and Practice of Logic Programming*, 3:463–498, July/September 2003.

[EFST01a] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. A Framework for Declarative Update Specifications in Logic Programs. In Bernhard Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654. Morgan Kaufmann, 2001. Extended version Technical Report INFSYS RR-1843-02-07, TU Wien, 2002.

[EFST01b] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. An Update Front-End for Extended Logic Programs. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 397–401. Springer, 2001. System Description (abstract).

[EFST02] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. On Properties of Update Sequences Based on Causal Rejection. *Journal of the Theory and Practice of Logic Programming*, 2(6):721–777, 2002.

[EG95] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[EH00] Stefan Edelkamp and Malte Helmert. On the Implementation of MIPS. In *Proceedings of the AIPS-Workshop on Model-Theoretic Approaches to Planning*, pages 18–25, 2000.

[EHN94] Kutluhan Erol, James A. Hendler, and Dana S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In Kristian J. Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 249–254. AAAI Press, June 1994.

[EL99] Esra Erdem and Vladimir Lifschitz. Transformations of Logic Programs Related to Causality and Planning. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 107–116, El Paso, Texas, USA, December 1999. Springer Verlag.

[ELM+98a] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. In Troels Andreasen, Henning Christiansen, and Henrik Legind Larsen, editors, *Proceedings International Conference on Flexible Query Answering Systems (FQAS'98)*, number 1495 in Lecture Notes in AI (LNAI), pages 148–163, Roskilde University, Denmark, May 1998. Springer.

[ELM+98b] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[ENS95] Kutluhan Erol, Dana S. Nau, and V.S. Subrahmanian. Complexity, Decidability and Undecidability Results for Domain-independent Planning. *Artificial Intelligence*, 76(1-2):75–88, July 1995.

[EP03] Thomas Eiter and Axel Polleres. Transforming coNP checks to answer set computation by meta-interpretation. In *Proceedings of the 2003 Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, Reggio Calabria, Italy, September 2003.

[EPM96] Eithan Ephrati, Martha E. Pollack, and Marina Mihlstein. A Cost-directed Planner: Preliminary Report. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1223 – 1228. AAAI Press, August 1996.

[Erd99] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[Fab02] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems.* PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[Fag94] François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[FG00] Paolo Ferraris and Enrico Giunchiglia. Planning as Satisfiability in Nondeterministic Domains. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 748–753. AAAI Press / The MIT Press, 2000.

[FGK93] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 1993.

[FL03] M. Fox and D. Long. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. Manuscript, `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`, April 2003.

[FLMP99] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[FMS00] Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings National Conference on AI (AAAI'00)*, pages 285–290, Austin, Texas, July 30-August 3 2000. AAAI Press.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[FP96] Wolfgang Faber and Gerald Pfeifer. `DLV` homepage, since 1996. `http://www.dlvsystem.com/`.

[FPR00] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open World Planning in the Situation Calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 754–760. AAAI Press / The MIT Press, 2000.

[GB96] R. Goldman and M. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings AIPS-96*, pages 110–117. AAAI Press, 1996.

[Gef00] Héctor Geffner. Functional strips: A more flexible language for planning and problem solving. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 187–209. Kluwer Academic Publishers, 2000.

[GHK+98] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL — The Planning Domain Definition language. Technical report, Yale Center for Computational Vision and Control, October 1998. Available at http://www.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz.

[Gin89] Matthew L. Ginsberg. Universal Planning: An (Almost) Universally Bad Idea. *AI Magazine*, 10(4):40–44, 1989.

[Giu00] Enrico Giunchiglia. Planning as Satisfiability with Expressive Action Languages: Concurrency, Constraints and Nondeterminism. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 657–666. Morgan Kaufmann, 2000.

[GKL97] Enrico Giunchiglia, G. Neelakantan Kartha, and Vladimir Lifschitz. Representing Action: Indeterminacy and Ramifications. *Artificial Intelligence*, 95:409–443, 1997.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[GL90] M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In *Proceedings of the 7th International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. MIT Press.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[GL93] Michael Gelfond and Vladimir Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[GL98a] Michael Gelfond and Vladimir Lifschitz. Action Languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.

[GL98b] Enrico Giunchiglia and Vladimir Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[GLL+03] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, Special Issue on Logical Formalizations of Commonsense Reasoning, 2003. To appear.

[GLLT01] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Causal Laws and Multi-Valued Fluents. In *Working Notes of the Fourth Workshop on Nonmonotonic Reasoning, Action and Change*, 2001.

[GLV99] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a Source of Expression Complexity. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.

[GRS98] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning*, pages 453–465, 1998.

[GSS03] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research*, Special Issue on the Third International Planning Competition, 2003. to appear.

[HD95] Marcus J. Huber and Edmund H. Durfee. On Acting Together: Without Communication. In *Spring Symposium Working Notes on Representing Mental States and Mechanisms*, pages 60–71. American Association for Artificial Intelligence, Stanford, California, 1995.

[HG00] Patrik Haslum and Héctor Geffner. Admissible Heuristics for Optimal Planning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 140–149, Breckenridge, Colorado, USA, April 2000. AAAI Press.

[Hin62] J. Hintikka. *Knowledge and Belief.* Cornell University Press Ithaca, NY, 1962.

[HJ99] Patrik Haslum and Peter Jonsson. Some results on the complexity of planning with incomplete information. In Susanne Biundo and Maria Fox, editors, *Proceedings of the Fifth European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Computer Science*, pages 308–318, Durham, UK, September 1999.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hof03] Jörg Hoffmann. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, Special Issue on the Third International Planning Competition, 2003. to appear.

[IB99] Stephen S. Intille and Aaron F. Bobick. A framework for recognizing multi-agent action from visual evidence. In *AAAI/IAAI*, pages 518–525, 1999.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, August 1987.

[Jan00] Tomi Janhunen. Comparing the expressive powers of some syntactically restricted classes of logic porgrams. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, Proceedings*, number 1861 in Lecture Notes in AI (LNAI), pages 852–866, London, UK, July 2000. Springer Verlag.

[Jan01] Tomi Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 93–106. Springer Verlag, September 2001.

[Jen93] Nicholas R. Jennings. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250, 1993.

[JNSY00] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Partiality and Disjunctions in Stable Model Semantics. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

[JV00] Rune M. Jensen and Manuela M. Veloso. OBDD-based Universal Planning for Multiple Synchronized Agents in Non-Deterministic Domains. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, pages 167–176, Breckenridge, Colorado, USA, April 2000. AAAI Press.

[JVB01] Rune M. Jensen, Manuela M. Veloso, and Michael H. Bowling. OBDD-based optimistic and strong cyclic adversarial planning. In *Proceedings of the 6th European Conference on Planning (ECP-01)*, Toledo, Spain, 2001.

[KHW95] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.

[Koe98] Jana Koehler. Planning Under Resource Constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98)*, pages 489–493, 1998.

[KPT01] Gal A. Kaminka, David V. Pynadath, and Milind Tambe. Monitoring deployed agent teams. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, pages 308–315. ACM, 2001.

[Kre92] Mark Krentel. Generalizations of Opt P to the Polynomial Hierarchy. *Theoretical Computer Science*, 97(2):183–198, 1992.

[KS92] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, 1992.

[KS99] Henry Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *The International Joint Conferences on Artificial Intelligence (IJCAI) 1999*, pages 318–325, Stockholm, Sweden, August 1999.

[KT00] Gal A. Kaminka and Milind Tambe. Robust agent teams via socially-attentive monitoring. *Journal of Artificial Intelligence Research*, 12:105–147, 2000.

[KW99] Henry Kautz and Joachim P. Walser. State-space planning by integer optimization. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 526–533, July 1999.

[LA97] C.L. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI) 1997*, pages 366–371, Nagoya, Japan, August 1997.

[Lif96] Vladimir Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lif97] Vladimir Lifschitz. On the Logic of Causal Explanation. *Artificial Intelligence*, 96:451–465, 1997.

[Lif99a] Vladimir Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[Lif99b] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[Lif02] Vladimir Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138:39–54, 2002.

[Lin95] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In Chris S. Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, pages 1985–1993. Morgan Kaufmann Publishers, 1995.

[LL94] Georg Lausen and Bertram Ludäscher. Updates by reasoning about states. Technical Report Bericht 59, Institut für Informatik, Universität Freiburg, September 1994.

[LL01] Joohyung Lee and Vladimir Lifschitz. Additive Fluents. In Alessandro Provetti and Son Tran Cao, editors, *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 116–123, Stanford, CA, 2001. AAAI Press.

[LL03] Joohyung Lee and Vladimir Lifschitz. Describing additive fluents in action language $\mathcal{C}+$. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, 2003. To appear.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1984.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. second edition.

[LMPG02] M. Luck, P. McBurney, C. Preist, and C. Guilfoyle. The agentlink agent technology roadmap draft. AgentLink, available at `http://www.agentlink.org/roadmap/index.html`, 2002.

[LPF+02] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. Technical Report cs.AI/0211004, arXiv.org, November 2002. Submitted to ACM TOCL.

[LRL+97] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1–3):59–83, 1997.

[LRS01] Nicola Leone, Riccardo Rosati, and Francesco Scarcello. Enhancing Answer Set Planning. In Alessandro Cimatti, Héctor Geffner, Enrico Giunchiglia, and Jussi Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 33–42, August 2001.

[LT94] V. Lifschitz and H. Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[LT99] V. Lifschitz and H. Turner. Representing Transition Systems by Logic Programs. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 92–106, El Paso, Texas, USA, December 1999. Springer Verlag.

[Luk90] Witold Lukaszewicz. *Non-Monotonic Reasoning, Formalization of Commonsense Reasoning.* Ellis Horwood Limited, 1990.

[LW92] Vladimir Lifschitz and Thomas Y. C. Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William R. Swartout, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 603–614. Morgan Kaufmann Publishers, October 1992.

[LZ02] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[McC58] John McCarthy. Programs with common sense. In *Symposium on the Mechanization of Thought Processes*, pages 77–84, 1958.

[McC90] John McCarthy. *Formalization of Common Sense, papers by John McCarthy edited by V. Lifschitz.* Ablex, 1990.

[McC99a] Norman McCain. The Causal Calculator Homepage, 1999. `http://www.cs.utexas.edu/users/tag/cc/`.

[McC99b] John McCarthy. Elaboration Tolerance. Available at `http://www-formal.stanford.edu/jmc/elaboration.html`., 1999.

[McD87] Drew McDermott. A Critique of Pure Reason. *Computational Intelligence*, 3:151–237, 1987. Cited in [CR00].

[MH69] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. reprinted in [McC90].

[Min88] Jack Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, June 2001.

[MR01] V.W. Marek and J.B. Remmel. On the Expressibility of Stable Logic Programming. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 107–120. Springer, 2001.

[MT91] Victor W. Marek and Mirosław Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[MT97] Norman McCain and Hudson Turner. Causal Theories of Actions and Change. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97)*, pages 460–465, 1997.

[MT98] Norman McCain and Hudson Turner. Satisfiability Planning with Causal Theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[Mye99] Karen Myers. FPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–69, 1999.

[Nar01] Alexander Nareyek. Beyond the Plan-Length Criterion. In *Local Search for Planning and Scheduling, ECAI 2000 Workshop*, volume 2148 of *Lecture Notes in Computer Science*, pages 55–78. Springer, August 2001.

[NCLMA99] Dana Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999*, pages 968–973, Stockholm, Sweden, August 1999. Morgan Kaufmann Publishers.

[Nie99] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[NS63] Alan Newell and Herbert A. Simon. Gps, a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. McGraw-Hill, 1963.

[NSS59] Alan Newell, J.C. Shaw, and Herbert A. Simon. Report on a general problem-solving program for a computer. *Computers and Automation*, 8(7):10–16, 1959.

[Pap85] Christos H. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, 1985.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PC96] Louise Pryor and Gregg Collins. Planning for Contingencies: A Decision-based Approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[PC01] Alessandro Provetti and Son Tran Cao, editors. *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*. AAAI Press, Stanford, CA, March 2001.

[Ped89] Edwin P. D. Pednault. Exploring the Middle Ground between STRIPS and the Situation Calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, Toronto, Canada, May 1989. Morgan Kaufmann Publishers, Inc.

[Peo98] Mark Alan Peot. *Decision-Theoretic Planning*. PhD thesis, Stanford University, Stanford, CA, USA, 1998.

[Pfe00] Gerald Pfeifer. *DLV: Evaluation Algorithms and Efficient Implementation*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich, 2000.

[Pol01] Axel Polleres. The DLV$^{\mathcal{K}}$ System for Planning with Incomplete Knowledge. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich, 2001.

[Pol03] Axel Polleres. The declarative planning system DLV^{K}: Progress and extensions. In Jeremy Frank and Susanne Biundo, editors, *Printed Notes of the ICAPS-03 Doctoral Consortium*, pages 94–98, June 2003.

[PR95] R. Parr and Stuart Russel. Approximating optimal policies for partially observable stochastic domains. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI '95)*, pages 1088–1094. Morgan Kaufmann Publishers, 1995.

[Prz88] Theodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[Prz89] T. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[Prz91] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[PS92] Mark A. Peot and David E. Smith. Conditional Nonlinear Planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197. AAAI Press, 1992.

[PS94] Christos H. Papadimitriou and Martha Sideri. Default Theories that always have Extensions. *Artificial Intelligence*, 69:347–357, 1994.

[Rei80] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[Rin99a] Jussi Rintanen. Constructing Conditional Plans by a Theorem-Prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[Rin99b] Jussi Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999*, pages 1192–1197, Stockholm, Sweden, August 1999. Morgan Kaufmann Publishers.

[Rin03] Jussi Rintanen. Complexity of planning with partial observability. In *ICAPS-03 Workshop on Planning under Uncertainty and Incomplete Information*, pages 103–109, Trento, Italy, June 2003.

[RN95] Stuart J. Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice-Hall, Inc., 1995.

[RV01] Ioannis Refanidis and Ioannis Vlahavas. A Framework for Multi-Criteria Plan Evaluation in Heuristic State-Space Planning. In *IJCAI-01 Workshop on Planning with Resources*, August 2001.

[SB01] Tran Cao Son and Chitta Baral. Formalizing Sensing Actions – A Transition Function Based Approach. *Artificial Intelligence*, 125(1–2):19–91, 2001.

[SBD+00] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation.* MIT Press, 2000.

[Sch87] Marcel J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI) 1987*, pages 1039–1046, Milan, Italy, August 1987. Morgan Kaufmann Publishers, Inc.

[Sel94] Alan L. Selman. A Taxonomy of Complexity Classes of Functions. *Journal of Computer and System Sciences*, 48(2):357–381, 1994.

[Sim00] Patrik Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Finland, 2000.

[SNS02] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[SP02] Tran Cao Son and Enrico Pontelli. Reasoning About Actions in Prioritized Default Theory. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, number 2424 in Lecture Notes in Computer Science, pages 369–381, Cosenza, Italy, September 2002.

[SW98] David E. Smith and Daniel S. Weld. Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, (AAAI'98)*, pages 889–896. AAAI Press / The MIT Press, July 1998.

[Syr02] Tommi Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[SZ95] V.S. Subrahmanian and Carlo Zaniolo. Relating Stable Models and AI Planning Domains. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 233–247, Tokyo, Japan, June 1995. MIT Press.

[Tam96] Milind Tambe. Tracking dynamic team activity. In *National Conference on Artificial Intelligence (AAAI-96)*, pages 80–87, 1996.

[Tur02] Hudson Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Proceedings of the 8th European Conference on Artificial Intelligence*

(JELIA), number 2424 in Lecture Notes in Computer Science, pages 111–124, Cosenza, Italy, September 2002.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, 1989.

[van88] A. van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In Minker [Min88], pages 1149–1176.

[Var82] Moshe Y. Vardi. Complexity of relational query languages. In *Proceedings of the 14th Symposium on Theory of Computation (STOC)*, pages 137–146, 1982.

[VPC98] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-Based Monitoring for Planning in Dynamic Environments. In Reid G. Simmons, Manuela M. Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 171–180, Pittsburgh Pennsylvania, USA, 1998. AAAI Press.

[vRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[WAS98] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending Graphplan to Handle Uncertainty & Sensing Actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, (AAAI'98)*, pages 897–904. AAAI Press / The MIT Press, July 1998.

[Wel94] Daniel S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 15(4):27–61, 1994.

[WH94] Mike Williams and Steve Hanks. Optimal Planning with a Goal-Directed Utility Model. In Kristian J. Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 176–181. AAAI Press, June 1994.

[WLB03] David Wilkins, Thomas Lee, and Pauline Berry. Interactive execution monitoring of agent teams. *Journal of Artificial Intelligence Research*, 18:217–261, 2003.

[Zha97] Hantao Zhang. SATO: An Efficient Propositional Prover. In *Proceedings of the International Conference on Automated Deduction (CADE'1997)*, pages 272–275, 1997.

Appendix A

Encodings

A.1 Quantified Boolean Formulae

This appendix contains the full integrated encoding for solving a sample QBF with one quantifier alternation as sketched in Section 4.2.5.1. Consider the following QBF:

$$\exists x_0 x_1 \forall y_0 y_1 (\neg x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_0) \vee (y_1 \wedge x_0 \wedge \neg y_0) \vee (y_0 \wedge \neg x_1 \wedge \neg y_0)$$

This QBF evaluates to true; for the assignments $x_0 = 0, x_1 = 0$ and $x_0 = 0, x_1 = 1$, the subformula $\forall y_0 y_1 (\cdots)$ is a tautology.

The integrated program $QBF_{solve} = QBF_{guess} \cup QBF'_{check}$ under use of the optimized transformation $tr_{Opt}()$ of tr as discussed is shown below with optimization $(\mathbf{OPT}_{mod})$ and $(\mathbf{OPT}_{dep})$ applied. It has two answer sets, namely $S_1 = \{\texttt{x0}, \texttt{-x1}, \ldots, \}$ and $S_2 = \{\texttt{x0}, \texttt{x1}, \ldots, \}$.

```
%%%% GUESS PART

  x0 v -x0.  x1 v -x1.

%%%% REWRITTEN CHECK PART
%% 1. Create dynamically the facts for the program

% y0 v -y0.
  lit(h,"y0",1). lit(h,"-y0",1).
  atom("y0","y0"). atom("-y0","y0").

% y1 v -y1.
  lit(h,"y1",2). lit(h,"-y1",2).
  atom("y1","y1"). atom("-y1","y1").

 % :- -y0, -x0.
 % :-  y0, -x0.
 % :- -y0, y1, x0.
 % :- -y0, y0, -x1.

%% 2. Optimized Meta-interpreter
%% 2.1 -- program dependent part

  notok :- ninS("y0"),ninS("-y0").
  notok :- ninS("y1"),ninS("-y1").
  notok :- inS("-y0"), -x0.
```

```
  notok :- inS("y0"),-x0.
  notok :- inS("y1"),inS("-y0"),x0.
  notok :- inS("y0"),inS("-y0"),-x1.

%% 2.2 -- fixed rules

% Iterate only over rules which contain L in the head:
  rule(L,R) :- lit(h,L,R), not lit(p,L,R),  not lit(n,L,R).
  ruleBefore(L,R) :- rule(L,R), rule(L,R1), R1 < R.
  ruleAfter(L,R) :- rule(L,R), rule(L,R1), R < R1.
  ruleBetween(L,R1,R2) :- rule(L,R1), rule(L,R2), rule(L,R3), R1 < R3, R3 < R2.
  firstRule(L,R) :- rule(L,R),  not ruleBefore(L,R).
  lastRule(L,R) :- rule(L,R),  not ruleAfter(L,R).
  nextRule(L,R1,R2) :- rule(L,R1), rule(L,R2), R1 < R2,  not ruleBetween(L,R1,R2).

% hlits are only those from active rules:
  hlit(L) :- rule(L,R).

  inS(L) v ninS(L) :- hlit(L).
  ninS(L) :- lit(HPN,L,R),  not hlit(L).
% Consistency check could be skipped for programs without class. negation:
  notok :- inS(L), inS(NL), L != NL, atom(L,A), atom(NL,A).

  dep(L,L1) :- rule(L,R),lit(p,L1,R),inS(L1), inS(L).
  dep(L,L2) :- rule(L,R),lit(p,L1,R),dep(L1,L2),inS(L).
  cyclic :- dep(L,L1), dep(L1,L).
  phi(L,L1) v phi(L1,L) :- dep(L,L1), dep(L1,L), L < L1, cyclic.
  phi(L,L2) :- phi(L,L1),phi(L1,L2), cyclic.
  failsToProve(L,R) :- rule(L,R), lit(p,L1,R), ninS(L1).
  failsToProve(L,R) :- rule(L,R), lit(n,L1,R), inS(L1).
  failsToProve(L,R) :- rule(L,R), rule(L1,R), inS(L1), L1 != L.
  failsToProve(L,R) :- lit(p,L1,R), rule(L,R), phi(L1,L), cyclic.
  allFailUpto(L,R) :- failsToProve(L,R), firstRule(L,R).
  allFailUpto(L,R1) :- failsToProve(L,R1), allFailUpto(L,R), nextRule(L,R,R1).
  notok :-  allFailUpto(L,R), lastRule(L,R), inS(L).
  phi(L,L1) :- notok, hlit(L), hlit(L1), cyclic.
  inS(L)     :- notok, hlit(L).
  ninS(L)    :- notok, hlit(L).

%%% 3. constraint
   :- not notok.
```

A.2 $\mathrm{lp}(\mathcal{P}_{QBridge})$

This Appendix contains the full translation $\mathrm{lp}(\mathcal{P}_{QBridge})$ from Section 3.2.3.

```
%%%%%%%%%%% Step 1 - Background Knowledge %%%%%%%%%%

person(joe). person(jack). person(william). person(averell).
walk(joe,1). walk(jack,2). walk(william,5). walk(averell,10).
side(here). side(across).
otherSide(X,Y) :- side(X), side(Y), X != Y.

%%%%%%%%%%% Step 2 - Auxiliary Predicates %%%%%%%%%%

time(0). time(1). time(2). time(3).
time(4). time(5). time(6). time(7).
nexttime(0,1). nexttime(1,2). nexttime(2,3). nexttime(3,4).
nexttime(4,5). nexttime(5,6). nexttime(6,7).

%%%%%%%%%%% Step 3 - Causation Rules %%%%%%%%%%

%    nonexecutable takeLamp(X) if hasLamp(X).
:- hasLamp(X,T0), takeLamp(X,T0), nexttime(T0,T1).
```

```
%    caused at(X,S1) after crossTogether(X,Y), at(X,S), otherSide(S,S1) .
at(X,S1,T1) :- person(X), side(S1), crossTogether(X,Y,T0),
               at(X,S,T0), otherSide(S,S1), nexttime(T0,T1).

%    caused at(Y,S1) after crossTogether(X,Y), at(Y,S), otherSide(S,S1) .
at(Y,S1,T1) :- person(Y), side(S1), crossTogether(X,Y,T0),
               at(Y,S,T0), otherSide(S,S1), nexttime(T0,T1).

%    caused -at(X,S) after crossTogether(X,Y), at(X,S).
-at(X,S,T1) :- person(X), side(S), crossTogether(X,Y,T0), at(X,S,T0),
               nexttime(T0,T1).

%    caused -at(Y,S) after crossTogether(X,Y), at(Y,S).
-at(Y,S,T1) :- person(Y), side(S), crossTogether(X,Y,T0), at(Y,S,T0),
               nexttime(T0,T1).

%    caused at(X,S1) after cross(X), at(X,S), otherSide(S,S1).
at(X,S1,T1) :- person(X), side(S1), cross(X,T0), at(X,S,T0), otherSide(S,S1),
               nexttime(T0,T1).

%    caused -at(X,S) after cross(X), at(X,S).
-at(X,S,T1) :- person(X), side(S), cross(X,T0), at(X,S,T0), nexttime(T0,T1).

%    caused hasLamp(X) after takeLamp(X).
hasLamp(X,T1) :- person(X), takeLamp(X,T0), nexttime(T0,T1).

%    caused -hasLamp(X) after takeLamp(Y), hasLamp(X), X != Y.
-hasLamp(X,T1) :- person(X), takeLamp(Y,T0), hasLamp(X,T0), X != Y,
                  nexttime(T0,T1).

%    inertial at(X,S).
at(X,S,T1) :- not -at(X,S,T1), person(X), side(S), person(X), side(S), at(X,S,T0),
              nexttime(T0,T1).

%    inertial hasLamp(X).
hasLamp(X,T1) :- not -hasLamp(X,T1), person(X), person(X), hasLamp(X,T0),
                 nexttime(T0,T1).

%    noConcurrency.
:- cross(X2,T0), cross(X1,T0), X2 != X1.
:- cross(X2,T0), crossTogether(X4, X3, T0).
:- cross(X2,T0), takeLamp(X3, T0).
:- crossTogether(X4,X2,T0), crossTogether(X3,X1,T0), X4 != X3.
:- crossTogether(X4,X2,T0), crossTogether(X3,X1,T0), X2 != X1.
:- crossTogether(X4,X2,T0), takeLamp(X5,T0).
:- takeLamp(X2,T0), takeLamp(X1,T0), X2 != X1.


%%%%%%%%%% Step 4 - Executability Conditions %%%%%%%%%%

%    executable crossTogether(X,Y) if hasLamp(X), at(X,S), at(Y,S).
crossTogether(X,Y,T0) v -crossTogether(X,Y,T0) :-
                  hasLamp(X,T0), at(X,S,T0), at(Y,S,T0),
                  walk(X,WX), walk(Y,WY), X != Y,
                  WX <= WY,next(T0,T1).
%    executable crossTogether(X,Y) if hasLamp(Y), at(X,S), at(Y,S).
crossTogether(X,Y,T0) v -crossTogether(X,Y,T0) :-
                  hasLamp(Y,T0), at(X,S,T0), at(Y,S,T0),
                  walk(X,WX), walk(Y,WY), X != Y,
                  WX <= WY,next(T0,T1).
%    executable cross(X) if hasLamp(X).
cross(X,T0) v -cross(X,T0) :- hasLamp(X,T0), walk(X,WX), next(T0,T1).

%    executable takeLamp(X) if hasLamp(Y), at(X,S), at(Y,S).
takeLamp(X,T0) v -takeLamp(X,T0) :- hasLamp(Y,T0), at(X,S,T0), at(Y,S,T0),
                                    person(X), next(T0,T1).
```

```
%%%%%%%%%%% Step 5 - Initial State Constraints %%%%%%%%%%%

%   caused at(X,here).
at(X,here,0) :- person(X), side(here), time(0).

%   caused hasLamp(joe).
hasLamp(joe,0) :- person(joe), time(0).


%%%%%%%%%%% Step 6 - Goal Query %%%%%%%%%%%

% goal: at(joe, across), at(jack, across),
%       at(william, across), at(averell,across)? (7)
goal :- at(joe,across,7), at(jack,across,7),
        at(william,across,7), at(averell,across,7).
:- not goal.


%%%%%%%%%%% Step 7 - Action Costs %%%%%%%%%%%

%    cross(X) requires walk(X,WX) costs WX.
costs_cross(X,T,WX) :- cross(X,T), walk(X,WX).
:~ costs_cross(X,T,WX), 0 < WX. [WX:1]

%    crossTogether(X,Y) requires walk(X,WX), walk(Y,WY), X != Y,
%                                WX <= WY costs WY.
costs_crossTogether(X,Y,T,WY) :- crossTogether(X,Y,T), walk(X,WX),
                                 walk(Y,WY), X != Y, WX <= WY.
:~ costs_crossTogether(X,Y,T,WY), 0 < WY. [WY:1]
```

Appendix B

Error Messages in DLV$^{\mathcal{K}}$

In the following, we will explain the most important error messages and warnings when calling DLV$^{\mathcal{K}}$ which are partly undocumented on the DLV web-site [FP96]. Since DLV$^{\mathcal{K}}$ is a front-end of the underlying DLV system, we refer to the DLV-manual [FP96] for other error messages that are not directly related to the planning front-end.

B.1 Warnings:

In the following some warnings are described which do not cause DLV$^{\mathcal{K}}$ to terminate but should serve to issue possibly unexpected behavior of the system to the user:

```
Argument to -planlength=<n> must be an integer between ...
Argument to -plancache=<n> must be an integer between ...
Argument to -planminactions=<n> must be an integer between ...
Argument to -planmaxactions=<n> must be an integer between ...
0 and 999999999 - ignored.
```

One of these warnings is displayed if the user gives a wrong argument, e.g. a non-positive integer, to one of the command-line options `-planlength`, `-plancache`, `-planminactions`, or `-planmaxactions`. For `-planlength` either the plan length given in the input is taken or the default 0. `-planlength` defaults to 10000 in the current version of DLV$^{\mathcal{K}}$.

```
-FPsoundcheck Ignored - Implemented checks are 1 and 2.
-FPcompletecheck Ignored - Implemented checks are 1 and 2.
-FPcheck Ignored - Implemented checks are 1 and 2.
```

One of these warnings is displayed if the user gives a wrong argument for command-line options `-FPcheck`, `-FPsoundcheck` or `-FPcompletecheck`. `-FPcheck` takes default value 1 in case.

```
Warning:  Background knowledge is either unstratified or includes
disjunction.
```

Despite our claim that background knowledge Π must have a unique answer set, in DLV$^{\mathcal{K}}$ arbitrary DLV programs $bk_1, \ldots, bk_n$ are allowed. However, DLV$^{\mathcal{K}}$ throws a warning whenever Π is not stratified or contains disjunctive rules.

`Warning: No plan length given. Plan length defaults to zero.`
DLV$^{\mathcal{K}}$ can only plan for given plan length. In case the user neither sets plan length with option `-planlength` nor specifies a goal query without concluding plan length in parentheses in the input the plan length defaults to 0, but the above warning is displayed.

`Warning: #maxint is smaller than plan length.`
This warning is displayed in case the user uses command-line option `-N=`n to specify an integer limit and n is smaller than the planlength specified by either command-line or in the input. Such a situation might in some special cases cause unexpected due to the current implementation of DLV's integer arithmetic.

`Warning: The domain is probably not mux-stratified, secure check might not work correctly.`
As mentioned above, whenever secure planning is chosen, DLV$^{\mathcal{K}}$ performs a check, whether the planning domain at hand is mux-stratified wrt. all mux-pairs of rules which contain contradictory literals in the `after`-parts. This warning is displayed, whenever this check fails. Since the domain might still be `false`-committed or serial, the implemented checks might work as expected, but it can not be guaranteed (cf. Sections 4.3.4.1 and 4.3.4.2 for a detailed discussion).

`Warning: goal:` <*new goal*> `replaces:` <*old goal*>
Multiple goals are not seen as a disjunction or conjunction of different goals: In case there is more than one `goal :` section in the input, only the last goal query in the last input file is taken. So, whenever more than one goal query is specified in the input, this warning is displayed to inform the user which is the actual goal which DLV$^{\mathcal{K}}$ tries to achieve.

B.2 Errors:

The following errors in either input or command-line are unrecoverable and cause the system to terminate immediately. The following error messages might occur:

`-planminactions: Minimum number of actions must be less than or equal to maximum number of actions per time.`
The user has specified command-line options `-planminactions=`n and `-planmaxactions=`m with values $n > m$.

`Background knowledge must not contain weak constraints.`
`Background knowledge must not contain aggregates.`
In the current version we do not allow the user to use weak constraints or aggregates in the background knowledge, since, in particular weak constraints might conflict with the rules emerging from our translation.

`Predicate names used in the background knowledge can not be used as fluent/action names.`
If you use a predicate name in a fluent/action declaration, the same predicate name must not have been used in the background knowledge already, since this might cause conflict in the translation to Answer Set Programming.

`Error while translating declaration:` <*declaration*>
`Error while translating rule:` <*rule/executability condition*>
A general error while translating a planning input file, e.g. an unsafe rule. Since DLV$^{\mathcal{K}}$ translates the input rule by rule, some errors might not be detected until translation but will be detected by the core DLV parser upon rule generation of the corresponding DLP rule(s). This message usually goes with a more specific error description from the core DLV parser.

`Only variables allowed in action/fluent declaration.`
In non-propositional action/fluent declarations all of arguments of $p(X_1, \ldots, X_n)$ in the declaration of the form (3.1) from Section 3.1.1 must be variables.

`Costs not allowed for fluents.`
The user has specified a `costs`-part in a fluent declaration.

`Action costs should be 'time', a number, or a variable.`
The user has specified a non-integer constant as action costs in the `costs`-part in an action declaration.

`Action/Fluent name already in use.`
The same fluent/action can not be defined twice.

`No dynamic rules allowed in initial section.`
In the `initially`-section neither executability conditions nor rules with a non-empty `after`-part are allowed.

`Actions must not be used in the if-part of causation rules.`
There is an action literal occurring in the `if`-part of some rule.

`Empty rules not allowed.`
An empty rule like "`caused`." or "`caused false`." is not allowed.

`Only fluents can be caused.`
A causation rule with a type literal or action in the head has been detected. This maybe hints at a typo.

`No after part allowed in nonexecutable statements.`
The `nonexecutable` macro does not allow for an `after`-part.

`No after part allowed in executable statements.`
Executability conditions do not allow for an `after`-part.

`Negation as failure not allowed for A in rule:`
`nonexecutable A if B.`
Rules of the form "`nonexecutable not` ..." are not allowed.

`Only actions allowed for A in rule:`
`nonexecutable A if B.`
A statement "`nonexecutable` X ..." where X is a type literal or fluent literal has been detected. This maybe hints at a typo.

`Only actions allowed for A in rule:`
`executable A if B.`

A statement "`executable` X ..." where X is a type literal or fluent literal has been detected. This maybe hints at a typo.

`Negation as failure not allowed for A in rule:`
`inertial A if B after C.`
Rules of the form "`inertial not` ..." are not allowed.

`Only fluents allowed for A in rule:`
`total A if B.`
A `total` statement with a type literal or action in the head has been detected. This maybe hints at a typo.

`True negation not allowed for A in rule:`
`total A if B after C.`
Rules of the form "`total -f` ..." are not allowed.

`Negation as failure not allowed for A in rule:`
`default A.`
Rules of the form "`default not` ..." are not allowed.

`number outside of given integer range.`
This error occurs, if command-line option `-N=`n to specify an integer limit has been used and an integer constant greater than n occurs in the input.

`Keyword 'time' not allowed here.`
`time` is a fixed keyword in $\mathcal{K}^c$ and must not be used anywhere but in the `costs` or `where`-part of action declarations.

`No goal query has been specified.`
There has to be at least one `goal :` statement in one of the input files.

`Only fluent literals allowed in planning goal queries.` A goal query may only consist of a comma-separated list of (possibly default-negated) fluent literals. Actions or type literals are not allowed. This maybe hints at a typo.

`Goal query is not safe:` <*goal*>
This error indicates that the last goal specified is not safe, i.e. the user has used variables in a default negated goal literal which are not bound in a positive literal.

Curriculum Vitae

Axel Florian Polleres

Personal Data

Degrees:	Diplom-Ingenieur (MSc)
Date and Place of Birth:	24. March 1974, Vienna, Austria
Citizenship:	Austrian
Family Status:	unmarried, no children
Current Position:	Research assistant at Vienna University of Technology
Office and Mailing Address:	Institut für Informationssysteme Abteilung für Wissensbasierte Systeme Favoritenstraße 9–11, A–1040 Vienna, Austria
E-mail:	axel@kr.tuwien.ac.at
Homepage:	`http://www.kr.tuwien.ac.at/staff/axel`
Home Address:	Klausgasse 35/32, A–1160 Vienna, Austria

Education

09/1980 – 06/1984:	Primary School Langkampfen, Tirol
09/1984 – 06/1992:	High School Kufstein, Tirol
06/1992:	High School graduation (with honors)
10/1992 – 04/2001:	Diploma Study in Computer Science at Vienna Univ. of Technology
04/2001:	Graduation to MSc in Computer Science (with honors). Thesis: "The DLV$^{\mathcal{K}}$ System for Planning with Incomplete Knowledge" supervised by Prof. Dr. Th. Eiter
Since 06/2001:	Doktoratsstudium, Studienzweig Informatik (PhD course in Computer Science) at Vienna University of Technology.

Working Experience

Industry/Business

Summer 1993 and 1994 :	BIOCHEMIE KUNDL GesmbH, Tirol
Summer 1996 and 05/1999 – 05/2000:	FERNWÄRME WIEN (formerly ENSORGUNGS–BETRIEBE SIMMERING), Vienna
07/1997–05/1999:	TARBUK AG, Schwechat
since 2000:	Freelance Database Application Development, FTCI Financial Training GesmbH, Vienna

Tutor at University Courses

Winter term 1994:	*Einführung in die Nutzung moderner Computernetzdienste* (EDV-Zentrum, TU Vienna)
Summer term 1995:	*Folgenabschätzung von Informationstechniken* Institut für Gestaltungs- und Wirkungsforschung, TU Vienna
Winter term 1996 & 1997:	*Laborübung Datenbanksysteme*
Summer term 2002:	*Datenmodellierung*
Winter term 2001 & 2002:	*Laborübung Logikorientierte Programmiersprachen*
Summer term 2003:	*Laborübung Wissensbasierte Systeme* Institut für Informationssysteme, TU Vienna

Scientific Work

01/1999 – 04/2000: Student fellow in the research project P11580-MAT, *Design and Implementation of a Query System for Disjunctive Deductive Databases*, supported by FWF (Austrian Science Funds), Institute für Informationssysteme, TU Vienna.

06/2001 – 09/2003: Research assistant in the project P14781-INF, *A Declarative Planning System Based On Logic Programming*, supported by FWF (Austrian Science Funds), Institute für Informationssysteme, TU Vienna.

Other Activities

Conference & Workshop Participations

- *6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*
- *2nd International PLANET Summer School on AI Planning 2002*
- *8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*
- *13th International Conference on Automated Planning & Scheduling (ICAPS 2003)*
- *3nd International PLANET Summer School on AI Planning 2003*
- *APPIA-GULP-PRODE 2003 Joint Conference on Declarative Programming (AGP 2003)*

Referee Activities

- *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001)*
- *8th European Conference on Logics in Artificial Intelligence (JELIA 2002)*
- *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2002)*
- *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*
- *Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)*
- *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2003)*
- *7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*

Memberships

- Member of the *Österrreichische Computer Gesellschaft (OCG)*, Austian Computer Society
- Member of the *Österreichische Gesellschaft für Artificial Intelligence (ÖGAI)*, Austrian Society for Artificial Intelligence

Awards

- *OCG-Förderpreis 2002*, award for outstanding master theses in the field of Computer Science by the Austrian Computer Society, for the master thesis "The DLV$^{\mathcal{K}}$ System for Planning with Incomplete Knowledge"

Selected Publications

- Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. *A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System.* Artificial Intelligence, 144(1–2):157–211, March 2003.
- Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. *Answer Set Planning under Action Costs.* Journal of Artificial Intelligence Research, 19:25–71, 2003.
- Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. *A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity.* ACM Transactions on Computational Logic, 2003. To appear.