DISSERTATION

Enhancing Efficiency and Expressiveness in Answer Set Programming Systems

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

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

E184/3

Institut für Informationssysteme

eingereicht an der Technischen Universität Wien

Fakultät für Technische Naturwissenschaften und Informatik

von

Wolfgang Faber
Matrikelnummer 9225210
Harkortstraße 9/3
1020 Wien

Wien, am 15. Mai 2002

Kurzfassung

Answer Set Programming (ASP) hat sich in den letzten Jahren zu einer anerkannt effektiven Methode zur deklarativen Wissensrepräsentation und automatisierten Schlussfolgerung entwickelt. Der ASP Formalismus kann als Fusion von Logischer Programmierung und Datenbanken betrachtet werden, mithilfe dessen alle Problemstellungen einer gewissen Komplexitätsklasse der zweiten Stufe der polynomiellen Hierarchie ausdrückbar und lösbar sind.

Die vorliegende Arbeit gliedert sich in zwei Teile: Im ersten Teil wird, nach einer kurzen Einführung in Answer Set Programming, das System `DLV` und dessen Architektur vorgestellt. Zwei besonders effizienzrelevante Komponenten dieses Systems werden genau analysiert und im Zuge dessen werden neuartige Verfahren für die Implementierung dieser Komponenten präsentiert, deren Nützlichkeit experimentell belegt wird.

Im zweiten Teil wird das im ersten Teil behandelte Kernsystem auf verschiedene Weise erweitert. Einerseits werden Erweiterungen des Basissystems vorgestellt, die entweder die Ausdrückbarkeit einer größeren Menge von Problemstellungen ermöglichen, oder aber die Formulierung bestimmter Zusammenhänge erleichtern. Andererseits wird gezeigt, wie man auf einfache, aber effektive Weise das Basissystem erweitern kann, um in unterschiedlichen Formalismen repräsentierte Problemstellungen mittels eines ASP Systems lösen zu können.

"Was sich überhaupt sagen läßt,
läßt sich klar sagen;
und wovon man nicht reden kann,
darüber muß man schweigen."

Ludwig Wittgenstein

Dedicated to my parents

Acknowledgments

First of all, I would like to thank my supervisor Thomas Eiter for his invaluable advice and thorough proofreading, and my second supervisor Nicola Leone for all the enlightening discussions which formed the roots of this thesis. I would also like to thank all my colleagues at the KR and DBAI groups of the Vienna University of Technology, in particular Gerald Pfeifer and Axel Polleres, and also all the people at DEIS and the Department of Mathematics of the University of Calabria.

My work has been supported by FWF (the Austrian Science Funds) under projects P11580-MAT, Z29-INF, P14781-INF and the Vienna University of Technology.

I thank Sonja Kovacic for her kindness and for bearing with me during the writing of this thesis. I also thank all of my friends for cheering up my life.

Last but not least nothing of this work could have been accomplished without the support of my family, in particular of my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Answer Set Programming (ASP) has evolved and been recognized as a convenient and powerful method for declarative knowledge representation and reasoning during the last decade [10]. The ASP paradigm can be seen as a fusion between logic programming and database technology, which allows for a purely declarative way of specifying and solving problems up to a particular complexity class called Σ_2^P.

Seen from a database perspective, ASP enlarges the class of problems that can be dealt with by traditional database languages, while keeping the underlying relational model. Seen from a logic programming perspective, ASP is a well-defined subset of general logic programming, which abolishes the need for a procedural interpretation, and in particular the need for termination analysis. ASP has a comparatively simple semantics, which is mathematically well-defined.

To summarize, the benefits of ASP are:

- Enhanced expressiveness with respect to traditional database languages.
- Purely declarative semantics, no procedural aspects.
- Guaranteed termination.
- Well-defined and well-understood semantics.
- Clear complexity results.

When the foundations for ASP have been laid down in e.g. [67, 13, 68, 108], the focus was on finding reasonable semantics for what became known as Answer Set Programs later-on. Many semantics have been defined for these programs in the early 1990ies (see [38, 39] for an overview), but the Answer Set Semantics as defined in [68] is now widely recognized as the canonical semantics for Answer Set Programs.

After this period of defining and refining semantics, the first prototype systems for computing Answer Sets were conceived. Among these are [8, 57,

60, 28, 27, 98, 99, 114, 6] and also [92], as described in [87]. Today, DLV and Smodels are the most widely used ASP systems.

The first systems were research prototypes, the goal of which was mainly to show the feasibility of computing answer sets or to demonstrate differences among the various semantics. Soon it became clear that the performance of these prototypes was too bad to solve real-life problems, so the main focus has switched from showing feasibility to enhancing efficiency. Another issue became obvious at about the same time: While ASP provides a very simple and flexible formalism, dealing with optimization or approximation tasks is usually cumbersome or even impossible. For this reason, enhancing expressivity to deal with such problems has become another focus. Yet another observation is that often problems can be specified in a more convenient way using a formalism different from ASP, which can then be automatically translated to ASP, solved by an ASP system, and re-transformed to the original formalism. It turned out that ASP seems to be very apt for this task.

This thesis focuses on these issues, which we summarize as *enhancing efficiency and expressiveness of ASP*. The term "expressiveness" is to be understood as having two dimensions: One concerns complexity, the other one ease of knowledge representation. While the concept of weak constraints (described in Chapter 6) enhances expressiveness in both dimensions, the concepts of inheritance (see Chapter 7), diagnostic reasoning (see Chapter 8), and other front-ends enhance expressiveness in the knowledge representation dimension. A particular ASP system, DLV, has been chosen as the basis of our work. The foundations of this system have been laid out in [80, 81], and the first prototype was built in 1996/1997 [57, 58, 60, 59].

Reflecting the title, the thesis consists of two parts: The first part describes techniques for enhancing the efficiency of ASP systems, which have been applied on the aforementioned DLV prototype. The second part shows how the two dimensions of expressiveness can be enhanced, again using DLV as the underlying system. Note that these two parts are orthogonal – by combining them, a very attractive system, which is both efficient and expressive, has been achieved.

Part I

Answer Set Programming Kernel

In this part we will focus on Answer Set Programming (ASP) in its original sense [68], with a bias towards the DLV system. We will first define the syntax of the underlying language, Disjunctive Datalog, followed by the associated semantics, Answer Sets, in Chapter 2. This language and its semantics serve as the kernel of the DLV system. Its benefits are its simplicity and its clear semantics.

In Chapter 3 we briefly discuss a general methodology (suggested in [45]) for encoding concrete problems as Answer Set programs by means of several examples. This chapter shows the suitability of ASP as a knowledge representation formalism.

In the following chapters, a computational tool for actually computing answer sets is described. The general architecture of the Answer Set Programming System DLV is presented in Chapter 4.

In Chapter 5 we focus on a key component of DLV, the Model Generator, discussing its design and implementation in detail. We first present the overall method of the Model Generator, and focus on two key parameters in this general design: Consequence derivation and choice. We will present a strong method for computing deterministic consequences, exploiting some genuine properties of ASP (as opposed to traditional frameworks such as satisfiability checking). Concerning the choice parameter, we will define several heuristics, and compare them experimentally.

In total, this part concerns a comprehensive account of Answer Set Programming, and of methods on how to build a sophisticated ASP system.

Chapter 2

Language

In this chapter, we provide a formal definition of the syntax and semantics of the kernel language of DLV: Disjunctive Datalog extended with strong negation under Answer Set Semantics. This programming method is referred to as Answer Set Programming (ASP) or Disjunctive Logic Programming (DLP). For further background, see [10, 56, 88, 68].

2.1 Syntax

The most basic part of the language consists of names for entities, which are referred to as constant symbols. Names, which can represent any entity are called variable symbols. Terms are any of these two.

Definition 2.1.1
Let σ_{var}, σ_{const}, and σ_{pred} denote sets of variable symbols, constant symbols, and predicate symbols, respectively. In the DLV system, σ_{var} consists of alphanumeric strings (possibly including underscores) starting with an uppercase letter, σ_{const} consists of

- *alphanumeric strings (possibly including underscores) starting with a lowercase letter,*
- *strings of arbitrary characters (except ") delimited by "",*
- *integer numbers,*

while σ_{pred} is equal to σ_{const} without numbers. With each $p \in \sigma_{pred}$ we associate a non-negative integer, denoted $arity(p)$[1]. We define the set of terms as $\sigma_{term} = \sigma_{var} \cup \sigma_{const}$.

[1] In the DLV system arities are not defined on a global scope, but rather locally in the scope of one program (see Definition 2.1.4).

Example 2.1.1
Here are a few examples for variable, constant, and predicate symbols:
`x`, `0815`, `planB`, `plan9fromOuterSpace`, `graph_1` $\in \sigma_{const}$*;*
`Placeholder`, `X1`, `VAR`, `V_X_1` $\in \sigma_{var}$*;*
`planB`, `is_weird_film`, `predicate1`, `graph_1` $\in \sigma_{pred}$*;*

Atoms define relations between terms, while literals define possibly negated relationships. There are two variants of negation: "Classical" (or "true") negation, which can be thought of as definite negation (falsity that can be proved), and negation as failure (NAF), which can be thought of as default negation (falsity because the contrary cannot be proved).

Definition 2.1.2
An atom *is an expression* $p(t_1, \ldots, t_n)$*, where* $p \in \sigma_{pred}$*,* $arity(p) = n$*, and* $t_1, \ldots, t_n \in \sigma_{term}$*. A* classical literal l *is either an atom* p *(in this case, it is* positive*), or a negated atom* $\neg\, p$ *(in this case, it is* negative*). A* negation as failure (NAF) literal ℓ *is of the form* l *or* `not` l*, where* l *is a classical literal; in the former case, it is* positive*, and in the latter case* negative*. Unless stated otherwise, by* literal *we refer to a classical literal.*

Given a classical literal l, its *complementary literal* $\neg.l$ is defined as $\neg\, p$ if l is an atom p and as p if $l = \neg\, p$. Given a set of classical literals L, its *complementary set* $\neg.L$ is defined as $\neg.L = \{\neg.l \mid l \in L\}$. A set L of classical literals is *consistent* if $L \cap \neg.L = \emptyset$, i.e. if for each literal $l \in L$, its complementary literal is not contained in L.

In a similar way, given a NAF literal ℓ, its *complementary NAF literal* `not`$.\ell$ is defined as `not` l if ℓ is a classical literal l and as l if $\ell =$ `not` l. Given a set of NAF literals $\mathcal{L}$, its *NAF-complementary set* `not`$.\mathcal{L}$ is defined as `not`$.\mathcal{L} =$ {`not`$.\ell \mid \ell \in \mathcal{L}$}. A set $\mathcal{L}$ of NAF literals is *NAF-consistent* if $\mathcal{L} \cap$ `not`$.\mathcal{L} = \emptyset$, i.e. if for each NAF literal $\ell \in \mathcal{L}$, its NAF-complementary literal is not contained in $\mathcal{L}$.

Example 2.1.2
`naughty(Child)` *and* `is_weird_film(plan9fromOuterSpace)` *are atoms and hence also positive classical and positive NAF literals, and* $arity$(`naughty`) = $arity$(`is_weird_film`) $= 1$ *holds.*
$\neg$ `naughty(Child)` *is a negative classical literal and positive NAF literal, while* `not naughty(Child)` *and* `not` $\neg$ `naughty(Child)` *are examples for negative NAF literals.*
The set $S =$ {`naughty(john)`, $\neg$ `naughty(john)`} *is not consistent, as* $\neg.S =$ {$\neg$ `naughty(john)`, `naughty(john)`} $\cap\, S = S \neq \emptyset$*.*

Connections between literals are expressed by means of rules.

Definition 2.1.3
A disjunctive rule *(*rule*, for short)* r *is an expression of the form*

$$a_1 \ \vee \ \ldots \ \vee \ a_n \ \leftarrow \ b_1, \ldots, b_k, \ \texttt{not}\ b_{k+1}, \ldots, \ \texttt{not}\ b_m.$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ *are classical literals and* $n \geq 0$, $m \geq k \geq 0$. *The disjunction* $a_1 \vee \ldots \vee a_n$ *is the* head *of* r, *while the conjunction* $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ *is the* body *of* r. *A rule without head literals (i.e.* $n = 0$*) is usually referred to as an* integrity constraint. *If the body is empty (i.e.* $k = m = 0$*), we usually omit the "*$\leftarrow$*" sign.*

Given a rule r, let $H(r) = \{a_1, \ldots, a_n\}$ denote the set of classical literals in the head, and by $B(r) = B^+(r) \cup B^-(r)$ the set of classical literals in the body, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$ are the sets of positive and negative body atoms, respectively. Furthermore, let $\mathcal{B}(r) = B^+(r) \cup \{\texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$ denote the set of NAF literals in the body of a rule.

A rule r is *safe* (or *range-restricted*), if all the variables occurring in it occur at least once in $B^+(r)$. For non-safe rules it is not intuitive what their range is, i.e. which constants they can be substituted with. The range could consist of all (infinitely many) constants, or the set of constants occurring in some given scope. Both variants have flaws: In the former case, infinity is introduced, while in the latter case the principle of domain-independency is violated. For details we refer to [1]. In the sequel, we will consider only safe rules.

Example 2.1.3
Let r_1 *be the following rule:*

$$\texttt{a(X)} \vee \neg\, \texttt{b(Y)} \leftarrow \texttt{c(X,Y)}, \neg\, \texttt{d(Y,Z)}, \texttt{not e(X)}, \texttt{not} \neg\, \texttt{f(X,Z)}.$$

Then, $H(r_1) = \{\texttt{a(X)}, \neg\, \texttt{b(Y)}\}$, $B^+(r_1) = \{\texttt{c(X,Y)}, \neg\, \texttt{d(Y,Z)}\}$, *and* $B^-(r_1) = \{\texttt{e(X)}, \neg\, \texttt{f(X,Z)}\}$. *Moreover,* $\mathcal{B}(r_1) = \{\texttt{c(X,Y)}, \neg\, \texttt{d(Y,Z)}, \texttt{not e(X)}, \texttt{not} \neg\, \texttt{f(X,Z)}\}$ *holds.* r_1 *is safe, as all variables in the rule (*$\texttt{X}, \texttt{Y}, \texttt{Z}$*) occur in* $B^+(r_1)$.

The rule r_{1b} $\texttt{a(X,Y)} \leftarrow \neg\, \texttt{b(X)}, \texttt{not c(X,Z)}$. *is not safe, as the variables* Y *and* Z *do not occur in* $B^+(r_{1b}) = \{\neg\, \texttt{b(X)}\}$.

Collections of rules are referred to as programs. These programs can also be thought of as knowledge bases.

Definition 2.1.4
A disjunctive datalog program $\mathcal{P}$ *(for short, simply* program*) is a finite set of safe rules. A* `not` *-free program* $\mathcal{P}$ *(i.e.* $\forall r \in \mathcal{P} : B^-(r) = \emptyset$*) is called* positive, *and a* $\vee$*-free program* $\mathcal{P}$ *(i.e.* $\forall r \in \mathcal{P} : |H(r)| \leq 1$*) is called* normal.

A term, an atom, a literal, a rule, or a program is called *ground*, if no variables appear in it. A ground program is also called a *propositional* program. For reasons of presentation, we will often denote programs not in explicit set notation, but rather by omitting curly braces and commata. Still, the programs should be thought of as sets.

Example 2.1.4
The following constitutes a non-ground disjunctive datalog program $\mathcal{P}_0$*:*

$$\texttt{naughty(john)}. \quad \neg\, \texttt{naughty(jack)}.$$
$$\texttt{child(john)}. \quad \texttt{child(jack)}. \quad \texttt{child(jill)}.$$
$$\neg\, \texttt{naughty(jill)} \vee \texttt{naughty(jill)}.$$
$$\texttt{nice(Child)} \leftarrow \texttt{child(Child)}, \texttt{not naughty(Child)}.$$
$$\leftarrow \texttt{nice(Child)}, \texttt{naughty(Child)}.$$

In rigorous set notation this would look like

$$\begin{aligned}\mathcal{P}_0 \;=\; \{&\texttt{naughty(john)}.,\ \neg\, \texttt{naughty(jack)}.,\ \texttt{child(john)}.,\ \texttt{child(jack)}.,\\ &\texttt{child(jill)}.,\ \texttt{nice(Child)} \leftarrow \texttt{child(Child)}, \texttt{not naughty(Child)}.,\\ &\leftarrow \texttt{nice(Child)}, \texttt{naughty(Child)}.\}\end{aligned}$$

which is arguably harder to read, mainly because the meaning of comma is overloaded.

Often, programs or knowledge bases are used to find an answer to a particular question. These questions are referred to as queries.

Definition 2.1.5
Let us define a query q *as*

$$a_1,\ \ldots,\ a_m,\ \texttt{not}\ a_{m+1},\ \ldots,\ \texttt{not}\ a_k?$$

where $a_1, \ldots, a_k$ *are ground classical literals and* $m \geq k \geq 0$. *Let* $B^+(q) = \{a_1, \ldots, a_m\}$ *and* $B^-(q) = \{a_{m+1}, \ldots, a_k\}$.

Example 2.1.5
Here is an example for a query:

$$\texttt{nice(jill)}, \texttt{not}\ \neg\, \texttt{naughty(jack)}?$$

The DLV system allows some convenient additions to the syntax presented so far. First, an *anonymous variable*, denoted by _, can be used, which is replaced by a unique variable. Second, DLV provides a couple of *built-in predicates*. The most important one is *inequality* and is denoted by <> and usually written infix, e.g. `X <> Y`. It holds for any pair of different constants. DLV also provides numeric constraints, see [104] for details.

This concludes the syntax section, we move on to define a meaning of these constructs.

2.2 Semantics

In this section, we describe the semantics of consistent Answer Sets, which has originally been defined in [68] and extends the Stable Model Semantics as defined in [67] for normal programs and in [108] for disjunctive programs.

Many other semantics have been proposed for (disjunctive) datalog programs, and even more for normal logic programs. See [65, 88, 95, 38, 39, 16, 17] for overviews and characterizations.

The Answer Set Semantics is defined on ground programs. Therefore the first issue is transforming a program containing variables into a ground program. This task is achieved by resorting to standard techniques of mathematical logic.

Definition 2.2.1
Given a program $\mathcal{P}$, let $U_\mathcal{P}$ (the Herbrand Universe*) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant χ is added to $U_\mathcal{P}$.*

Given a program $\mathcal{P}$, let $B_\mathcal{P}$ (the Herbrand Literal Base*) be the set of all ground atoms constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$, possibly preceded by $\neg$.*

Given a rule r, $Ground(r)$ (the Ground Instantiation*) denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$. In a similar way, given a program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r\in\mathcal{P}} Ground(r)$.*

Observe that for propositional programs, $\mathcal{P} = Ground(\mathcal{P})$ holds.

For every program $\mathcal{P}$, we define its *Answer Sets* using its ground instantiation $Ground(\mathcal{P})$ in two steps, following [85]: First we define the answer sets of positive programs, then we give a reduction of programs containing negation to positive ones and use it to define answer sets of arbitrary programs, possibly containing negation as failure.

Definition 2.2.2
An interpretation $I \subseteq B_\mathcal{P}$ is a set of classical literals from $B_\mathcal{P}$. A consistent[2] *interpretation $X \subseteq B_\mathcal{P}$ is called* closed under $\mathcal{P}$*, where $\mathcal{P}$ is a positive program), if for every $r \in Ground(\mathcal{P})$ $H(r) \cap X \neq \emptyset$ whenever $B(r) = B^+(r) \subseteq X$. A set $X \subseteq B_\mathcal{P}$ is an* answer set *for $\mathcal{P}$ if it is a minimal set (w.r.t. set inclusion) that is closed under $\mathcal{P}$.*

Example 2.2.1
For the positive program $\mathcal{P}_1$

$$\mathtt{a} \vee \neg\, \mathtt{b} \vee \mathtt{c}.$$

the answer sets are $\{\mathtt{a}\}$, $\{\neg\, \mathtt{b}\}$, and $\{\mathtt{c}\}$ while for the positive program $\mathcal{P}_2$

$$\mathtt{a} \vee \neg\, \mathtt{b} \vee \mathtt{c}.$$
$$\leftarrow \mathtt{a}.$$

only $\{\neg\, \mathtt{b}\}$ and $\{\mathtt{c}\}$ are answer sets. Finally, for the positive program $\mathcal{P}_3$

[2]Note that we only consider *consistent answer sets*, while in [85] also the inconsistent set of all possible literals is a valid answer set.

$\mathtt{a} \vee \neg\, \mathtt{b} \vee \mathtt{c}.$
$\leftarrow \mathtt{a}.$
$\neg\, \mathtt{b} \leftarrow \mathtt{c}.$
$\mathtt{c} \leftarrow \neg\, \mathtt{b}.$

the set $\{\neg\, \mathtt{b}, \mathtt{c}\}$ is the only answer set.

Let us now move on and extend the definition of answer sets also for programs containing negation as failure literals.

Definition 2.2.3
The reduct *or* Gelfond-Lifschitz transform *of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by*

1. *deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;*
2. *deleting the negative body from the remaining rules.*

An answer set of a program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$. Let the set of answer sets for a program $\mathcal{P}$ be denoted by $\mathcal{AS}(\mathcal{P})$.

Example 2.2.2
Given the program $\mathcal{P}_4$

$\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}.$
$\neg\, \mathtt{b} \leftarrow \mathtt{not}\ \mathtt{a}, \mathtt{not}\ \mathtt{c}.$
$\mathtt{a} \vee \mathtt{c} \leftarrow \mathtt{not}\ \neg\, \mathtt{b}.$

and $I = \{\neg\, \mathtt{b}\}$, the reduct $\mathcal{P}_4^I$ is

$\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}.$
$\neg\, \mathtt{b}.$

It is easy to see that I is an answer set of $\mathcal{P}_4^I$, and for this reason it is also an answer set of $\mathcal{P}_4$, $I \in \mathcal{AS}(\mathcal{P}_4)$.

Now consider $J = \{\mathtt{a}\}$. The reduct $\mathcal{P}_4^J$ is

$\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}.$
$\mathtt{a} \vee \mathtt{c} \leftarrow.$

and it can be easily verified that J is an answer set of $\mathcal{P}_4^J$, so it is also an answer set of $\mathcal{P}_4$, $J \in \mathcal{AS}(\mathcal{P}_4)$.

If, on the other hand, we take $K = \{\mathtt{c}\}$, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$ (for the rule r_2: $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}.$ it holds that $B(r_2) \subseteq K$, but $H(r_2) \cap K \neq \emptyset$ is violated). Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_4$, so $\mathcal{AS}(\mathcal{P}_4) = \{\{\neg\, \mathtt{b}\}, \{\mathtt{a}\}\}$.

Remark A In some cases, it is possible to emulate disjunction by non-stratified normal rules. However, this is not possible in general. For example, consider $\mathcal{P}_{5a}$

$$\mathtt{a} \vee \mathtt{b}.$$

and $\mathcal{P}_{5b}$

$$\mathtt{a} \leftarrow \mathtt{not\ b}.$$
$$\mathtt{b} \leftarrow \mathtt{not\ a}.$$

Both programs have the same answer sets, namely $\mathcal{AS}(\mathcal{P}_{5a}) = \mathcal{AS}(\mathcal{P}_{5b}) = \{\{\mathtt{a}\}, \{\mathtt{b}\}\}$. On the other hand, $\mathcal{P}_{6a}$

$$\mathtt{a} \vee \mathtt{b}.$$
$$\mathtt{a} \leftarrow \mathtt{b}.$$
$$\mathtt{b} \leftarrow \mathtt{a}.$$

has a single answer set $\mathcal{AS}(\mathcal{P}_{6a}) = \{\{\mathtt{a}, \mathtt{b}\}\}$, while $\mathcal{P}_{6b}$

$$\mathtt{a} \leftarrow \mathtt{not\ b}.$$
$$\mathtt{b} \leftarrow \mathtt{not\ a}.$$
$$\mathtt{a} \leftarrow \mathtt{b}.$$
$$\mathtt{b} \leftarrow \mathtt{a}.$$

has no answer set at all ($\mathcal{AS}(\mathcal{P}_{6b}) = \emptyset$). A formal analysis on the relation between semantics of disjunctive and corresponding non-disjunctive programs is given in [40], in which the reduction from disjunctive to non-disjunctive programs is given by "shift-operations". Indeed, $\mathcal{P}_{5b}$ is obtained from $\mathcal{P}_{5a}$ and also $\mathcal{P}_{6b}$ is obtained from $\mathcal{P}_{6a}$ by applying a complete shift as defined in [40].

On the other hand, classical negation can be emulated by new atoms and constraints:

Proposition 2.2.1
Given a program $\mathcal{P}$, let $\mathcal{P}^{\neg}$ be $\mathcal{P}$, where each classical literal $\neg\, \mathtt{a}$ is replaced by an atom $\mathtt{a}'$, which does not occur in $\mathcal{P}$. Furthermore, for each atom $\mathtt{a}$ and its complementary literal $\neg\, \mathtt{a}$ in $\mathcal{P}$, let the constraint $\leftarrow \mathtt{a}, \mathtt{a}'.$ be contained in $\mathcal{P}^{\neg}$. Then $\mathcal{AS}(\mathcal{P}^{\neg}) = \mathcal{AS}(\mathcal{P})'$, where $\mathcal{AS}(\mathcal{P})'$ denotes $\mathcal{AS}(\mathcal{P})$ in which each classical literal $\neg\, \mathtt{z}$ is replaced by the corresponding $\mathtt{z}'$.

So far we have not dealt with the question of how to deal with a program $\mathcal{P}$ together with a query q. With queries, there are several modes of reasoning: Seen as a *decision problem*, we ask whether q follows from $\mathcal{P}$, so the semantics is answering "yes" or "no" to the question posed by the query. In this setting, usually two variants are considered: *brave* and *cautious reasoning*. The semantics can also be viewed as a *computation problem*, in which the semantics is given by those answer sets of $\mathcal{P}$ of which q is a consequence. In the literature, the focus is on the decision problem variant.

Definition 2.2.4
Given a program $\mathcal{P}$ and a query q, $\mathcal{P}$ bravely entails *q, written $\mathcal{P} \models_b q$, if q holds in at least one answer set of $\mathcal{P}$, i.e. if $\exists A \in \mathcal{AS}(\mathcal{P}) : B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset$ holds. On the other hand, $\mathcal{P}$* cautiously entails *q, written $\mathcal{P} \models_c q$, if q holds in all answer sets of $\mathcal{P}$, i.e. if $\forall A \in \mathcal{AS}(\mathcal{P}) : B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset$ holds.*

Frequently, we omit the ? of a query q when we deal with entailment.

Example 2.2.3
Let $\mathcal{P}_7$ be the following program:

$\mathtt{a} \vee \mathtt{b}.$
$\mathtt{c} \leftarrow \mathtt{a}.$
$\mathtt{c} \leftarrow \mathtt{b}.$

The answer sets of the problem are $\mathcal{AS}(\mathcal{P}_7) = \{\{\mathtt{a}, \mathtt{c}\}, \{\mathtt{b}, \mathtt{c}\}\}$. Now consider q_{7a}:

$\mathtt{a}, \mathtt{not\ b}?$

and q_{7b}:

$\mathtt{c}?$

The following holds: $\mathcal{P}_7 \models_b q_{7a}$, $\mathcal{P}_7 \not\models_c q_{7a}$, $\mathcal{P}_7 \models_b q_{7c}$, $\mathcal{P}_7 \models_c q_{7c}$.

Note that $\mathcal{P}_{incons} \models_c q$ holds for any query q if $\mathcal{AS}(\mathcal{P}_{incons}) = \emptyset$ holds.

Definition 2.2.5
For the computation problem semantics of a program $\mathcal{P}$ and a query q, let $\mathcal{AS}(\langle \mathcal{P}, q \rangle) = \{A \mid A \in \mathcal{AS}(\mathcal{P}) \wedge B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset\}$.

Example 2.2.4
Reconsider $\mathcal{P}_7$ and q_{7a} — it is easy to see that $\mathcal{AS}(\langle \mathcal{P}_7, q_{7a} \rangle) = \{\{a, c\}\}$.

Let us note a few properties, which are easy to see, without proof.

Lemma 2.2.1
Given a program $\mathcal{P}$ and query q,

- *$\mathcal{P} \models_b q$ iff $|\mathcal{AS}(\langle \mathcal{P}, q \rangle)| \geq 1$,*
- *$\mathcal{P} \models_c q$, iff $\mathcal{AS}(\mathcal{P}) = \mathcal{AS}(\langle \mathcal{P}, q \rangle)$.*

We will now reduce the problem of reasoning about queries to the computation of answer sets of programs without a query:

Proposition 2.2.2
Given a program $\mathcal{P}$ and a query q of the form $\mathtt{a}_1, \ldots, \mathtt{a}_m, \mathtt{not\ a}_{m+1}, \ldots, \mathtt{not\ a}_k?$, let $\mathcal{P}_{q_1}$ be the following program:

$$\leftarrow \texttt{not a}_1.\ \ldots\ \leftarrow \texttt{not a}_m.\ \leftarrow \texttt{a}_{m+1}.\ \ldots\ \leftarrow \texttt{a}_k.$$

and let $\mathcal{P}_{q_2}$ *be*

$$\leftarrow \texttt{a}_1, \ldots, \texttt{a}_m, \texttt{not a}_{m+1}, \ldots, \texttt{not a}_k.$$

Now, if $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}_{q_1}$ *and* $\mathcal{P}'' = \mathcal{P} \cup \mathcal{P}_{q_2}$ *the following statements hold:*

1. $\mathcal{AS}(\langle \mathcal{P}, q \rangle) = \mathcal{AS}(\mathcal{P}')$
2. $\mathcal{P} \models_b q$ *iff* $|\mathcal{AS}(\mathcal{P}')| \geq 1$
3. $\mathcal{P} \models_c q$ *iff* $\mathcal{AS}(\mathcal{P}'') = \emptyset$

Proof

1. First we note that $\mathcal{AS}(\mathcal{P}') \subseteq \mathcal{AS}(\mathcal{P})$ and $\mathcal{AS}(\langle \mathcal{P}, q \rangle) \subseteq \mathcal{AS}(\mathcal{P})$. Assume that $\exists A : A \in \mathcal{AS}(\langle \mathcal{P}, q \rangle)$ holds. By Definition 2.2.5 $A \in \mathcal{AS}(\mathcal{P})$ and $B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset$ holds. This means that $\mathcal{P}'^A$ does not contain $\leftarrow \texttt{not a}_1.\ \ldots\ \leftarrow \texttt{not a}_m.$, and therefore $\mathcal{P}'^A = \mathcal{P}^A$ and $A \in \mathcal{AS}(\mathcal{P}')$. Now assume that $\exists A : A \in \mathcal{AS}(\mathcal{P}')$ holds. $\texttt{a}_1 \in A, \ldots, \texttt{a}_m \in A$ must hold, otherwise $\mathcal{P}'^A$ would contain a rule $\leftarrow .$ which is not closed under any interpretation. Also $\texttt{a}_{m+1} \in A, \ldots, \texttt{a}_k \notin A$, otherwise $\mathcal{P}'^A$ would not be closed in A. Together, $B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset$ holds, and so A is an answer set of $\langle \mathcal{P}, q \rangle$. In total, we have proved $\mathcal{AS}(\langle \mathcal{P}, q \rangle) = \mathcal{AS}(\mathcal{P}')$.
2. Follows directly from 1 and Lemma 2.2.1.
3. First let us note that $\mathcal{AS}(\mathcal{P}'') \subseteq \mathcal{AS}(\mathcal{P})$. Suppose now $\mathcal{P} \not\models_c q$, then $B^+(q) \not\subseteq A \vee B^-(q) \cap A \neq \emptyset$ must hold for some $A \in \mathcal{AS}(\mathcal{P})$. If $B^-(q) \cap A \neq \emptyset$ then $\mathcal{P}''^A = \mathcal{P}^A$ and thus $A \in \mathcal{AS}(\mathcal{P}'')$; if, on the other hand $B^-(q) \cap A = \emptyset$ and $B^+(q) \not\subseteq A$ then $\mathcal{P}''^A$ is closed under A, and $A \in \mathcal{AS}(\mathcal{P}'')$. In total $A \in \mathcal{AS}(\mathcal{P}'')$ holds in any case. Suppose $\mathcal{P} \models_c q$ and $\mathcal{AS}(\mathcal{P}'') \neq \emptyset$ hold. Let $A \in \mathcal{AS}(\mathcal{P}'')$ be an arbitrary answer set of $\mathcal{P}''$. Either $\{\texttt{a}_{m+1}, \ldots, \texttt{a}_k\} \cap A \neq \emptyset$ holds, or otherwise $\{\texttt{a}_1, \ldots, \texttt{a}_m\} \not\subseteq A$ must hold for $\mathcal{P}''^A$ to be closed under A. But then $B^+(q) \subseteq A \wedge B^-(q) \cap A = \emptyset$ does not hold for $A \in \mathcal{AS}(\mathcal{P}'') \subseteq \mathcal{AS}(\mathcal{P})$, and therefore $\mathcal{P} \not\models_c q$, which is a contradiction to the assumption. This concludes the proof to 3.

□

Chapter 3

Applications – Guess and Check Paradigm

The core language of DLV can be used to encode problems in a highly declarative fashion, following a "Guess & Check" paradigm. In this section, we will describe this technique and illustrate how to apply it on a number of examples. We will see that many problems, also problems of comparatively high computational complexity (that is, even Σ_2^P-complete problems), can be solved naturally with DLV by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are even more complex than NP uniformly over varying instances of the problem using a fixed program (i.e., a fixed program containing variables that works on any input).

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a Guess & Check program $\mathcal{P}$ for $\mathbf{P}$ consists of the following two parts:

Guessing Part: The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, in a way such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part: The checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program tests whether a solution candidate is in fact a solution, such that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the solutions for the problem instance I.

In general, we may allow both $\mathcal{G}$ and $\mathcal{C}$ to be arbitrary collections of rules in the program, and it may depend on the complexity of the problem which kind of rules are needed to realize these parts (in particular, the checking part); we defer this discussion to a later point in this chapter.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., all checking is integrated into the guessing part such that solution candidates are always solutions. However, in general the generation of the search space

may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For many problems, however, a natural Guess & Check program can be designed, in which the two parts are clearly identifiable and have a simple structure:

- The guessing part $\mathcal{G}$ consists of a disjunctive rule which "guesses" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints which check the admissibility of S, possibly using auxiliary predicates which are defined by normal stratified[1] rules.

Thus, the disjunctive rule defines the search space[2], in which rule applications are branching points, while the integrity constraints prune illegal branches.

3.1 3-Colorability

As an example which matches this scheme, let us consider the well-known *3-Colorability* problem.

Definition 3.1.1 (3COL)
Given a (finite) graph $G = (V, E)$ in the input, assign each node one of three colors (say, red, green, or blue) such that adjacent nodes v_1, v_2 ($(v_1, v_2) \in E$) always have different colors.

3-Colorability is a classical NP-complete problem. Assuming that the set of nodes V and the set of edges E are specified by means of predicates `node` (which is unary, i.e. $arity(\texttt{node}) = 1$) and `edge` (binary, i.e. $arity(\texttt{node}) = 2$), respectively, it can be encoded by the following Guess & Check program $\mathcal{P}_{3col}$:

$$\texttt{r}_{\texttt{3col}} : \texttt{col}(\texttt{X}, \texttt{r}) \vee \texttt{col}(\texttt{X}, \texttt{g}) \vee \texttt{col}(\texttt{X}, \texttt{b}) \leftarrow \texttt{node}(\texttt{X}). \quad \} \quad \textbf{Guess}$$

$$\texttt{c}_{\texttt{3col}} :\leftarrow \texttt{edge}(\texttt{X}, \texttt{Y}), \texttt{col}(\texttt{X}, \texttt{C}), \texttt{col}(\texttt{Y}, \texttt{C}). \quad \} \quad \textbf{Check}$$

The first rule, r_{3col}, nondeterministically guesses color assignments for the nodes in the graph, and the constraint, c_{3col}, ensures that these choices

[1] For a definition of stratification, see [7].

[2] As described in Remark A at the end of Section 2.2, in some cases it would be possible to replace the disjunctive guessing rule by rules with unstratified negation. However, this is not possible in general. Disjunctive rules also have the advantage of being more compact and usually also more natural.

are legal, i.e., that no two nodes which are connected by an edge have the same color.[3]

More precisely, let us suppose that the nodes and edges of the graph G are represented by a set $\mathcal{F}$ of facts with predicates `node` and `edge`. Then the "guessing" rule r_{3col} above states that every node is colored either red or green or blue, while the "checking" constraint c_{3col} forbids the assignment of the same color to two adjacent nodes. The answer sets of $\mathcal{F} \cup \{r_{3col}\}$ are all possible ways of coloring the graph. Note that minimality of answer sets guarantees that every node has only one color.

If an answer set of $\mathcal{F} \cup \{r_{3col}\}$ satisfies the constraint c_{3col}, then it represents an admissible 3-coloring of the graph. There is in fact a one-to-one correspondence between the solutions of the 3-coloring problem and the answer sets of $\mathcal{F} \cup \{r_{3col}, c_{3col}\}$. The graph is thus 3-colorable if and only if $\mathcal{F} \cup \{r_{3col}, c_{3col}\}$ has some answer set, and each of the answer sets of $\mathcal{F} \cup \{r_{3col}, c_{3col}\}$ represents a (distinct) legal 3-coloring of G.

3.2 Hamiltonian Path

Let us consider next another classical NP-complete problem in graph theory, namely *Hamiltonian Path*.

Definition 3.2.1 (HAMPATH)
Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by using predicates `node` (unary) and `arc` (binary), and the starting node is specified by the predicate `start` (unary). Then, the following Guess & Check program $\mathcal{P}_{hp}$ solves the problem HPATH.

$$\left.\begin{array}{l} \texttt{inPath}(\texttt{X},\texttt{Y}) \vee \texttt{outPath}(\texttt{X},\texttt{Y}) \leftarrow \texttt{arc}(\texttt{X},\texttt{Y}). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \left.\begin{array}{l} \leftarrow \texttt{inPath}(\texttt{X},\texttt{Y}), \texttt{inPath}(\texttt{X},\texttt{Y1}), \texttt{Y} <> \texttt{Y1}. \\ \leftarrow \texttt{inPath}(\texttt{X},\texttt{Y}), \texttt{inPath}(\texttt{X1},\texttt{Y}), \texttt{X} <> \texttt{X1}. \\ \leftarrow \texttt{node}(\texttt{X}), \texttt{not reached}(\texttt{X}). \end{array}\right\} \textbf{Constraints} \\ \left.\begin{array}{l} \texttt{reached}(\texttt{X}) \leftarrow \texttt{start}(\texttt{X}). \\ \texttt{reached}(\texttt{X}) \leftarrow \texttt{reached}(\texttt{Y}), \texttt{inPath}(\texttt{Y},\texttt{X}). \end{array}\right\} \begin{array}{l}\textbf{Auxiliary} \\ \textbf{Predicate}\end{array} \end{array}\right\} \textbf{Check}$$

The first rule guesses a subset S of all given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian Path. Here, the checking part $\mathcal{C}$ uses an auxiliary predicate `reached`, which is defined using positive recursion.

[3] In this example, we assume that G contains no loops, i.e., edges from a node to itself. Such loops can be easily handled by adding `X <> Y` to c_{3col}.

In particular, the first two constraints in $\mathcal{C}$ check whether the set of arcs S selected by `inPath` meets the following requirements, which any Hamiltonian Path must satisfy: There must not be two arcs starting at the same node, and there must not be two arcs ending in the same node.

The two rules after the constraints define the reachability relation from the starting node with respect to the set of arcs S. This relation is used in the third constraint, which enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. This constraint also ensures that this subgraph is connected.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $p = v_0, v_1, \ldots, v_k$ in G that starts at node a, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is a Hamiltonian Cycle, and by dropping the last arc, we have a Hamiltonian Path.

Thus, given a set of facts $\mathcal{F}$ for *node*, *arc*, and *start* which specify the problem input, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the input graph has a Hamiltonian Path.

We remark that if we want to compute a Hamiltonian Path rather than only answering whether such a path exists, we can strip off the last arc from a Hamiltonian Cycle by adding a further constraint $\leftarrow$ `start(Y)`, `inPath(_,Y)`. to the program. Then, the set S of selected arcs in an answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a.

3.3 3-Satisfiability

Satisfiability, and in particular 3-Satisfiability, is one of the best researched problems in AI and generally used for solving many other problems by translating them to 3SAT, solving the 3SAT problem and transforming the solution back to the original domain:

> Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee d_{i,2} \vee d_{i,3})$ where the $d_{i,j}$ are literals over the propositional variables $x_1, \ldots, x_m$.
>
> Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$ (see e.g. [101] for a complete definition).

3SAT is a classical NP-complete problem and can be easily represented in our formalism as follows:

For every propositional variable x_i $(1 \leq i \leq m)$, we add the following rule which ensures that we either assume that variable x_i or its complement nx_i true.

$$x_i \vee nx_i.$$

The set of all such rules constitutes the guessing part of 3SAT.
For each clause $d_1 \vee \ldots \vee d_3$ in Φ we add the constraint

$$\leftarrow \texttt{not}\ \bar{d}_1, \ldots, \texttt{not}\ \bar{d}_3.$$

where $\bar{d}_i$ $(1 \leq i \leq 3)$ is x_j if d_i is a positive literal x_j, and nx_j if d_i is a negative literal $\texttt{not}\ x_j$. The checking part consists exactly of these constraints.

3.4 Strategic Companies

The problems considered so far are popular examples of NP-complete problems. We next show that also harder problems, located at the second level of the polynomial hierarchy, can be encoded by the Guess & Check technique. To this end, we consider the following problem *Strategic Companies*.

Definition 3.4.1 (STRATCOMP [24])
Given the collection $C = \{c_1, \ldots, c_m\}$ of companies c_i owned by a holding, the set $G = \{g_1, \ldots, g_n\}$ of goods, and for each c_i a set $G_i \subseteq G$ of goods produced by c_i, and a set $O_i \subseteq C$ of companies controlling (owning) c_i. This control can be thought of as a majority in shares; companies not in C, which we do not model here, might have shares in companies as well. O_i is referred to as controlling set *of c_i. Note that, in general, a company might have more than one controlling set, and only irredundant controlling sets (i.e., no proper subset is a controlling set) are recorded then. Let the holding produce all goods in G, i.e. $G = \bigcup_{c_i \in C} G_i$.*

A subset of the companies $C' \subset C$ is a production-preserving *set if the following conditions hold: (1) The companies in C' produce all goods in G, i.e., $\bigcup_{c_i \in C'} G_i = G$. (2) The companies in C' are closed under the controlling relation, i.e. if $O_i \subseteq C'$ for some $i = 1, \ldots, m$ then $c_i \in C'$ must hold.*

A subset-minimal set C', which is production-preserving, *is called a* strategic set. *A company $c_i \in C$ is called* strategic, *if it belongs to some strategic set of C.*

Computing the set of all strategic companies is relevant when companies should be sold, as selling any company which is strategic would for sure lead to a violation of any of conditions (1) and (2). This problem is Σ_2^P-hard in general [24]; reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is one of the rare KR problems from the business domain of this complexity that have been considered so far.

In the following, we adopt the setting from [24] where each product is produced by at most two companies (for each $g \in G$ $|\{c_i \mid g \in G_i\}| \leq 2$) and each company is jointly controlled by at most three other companies,

i.e. $|O_i| \leq 3$ for $i = 1, \ldots, m$ (in this case, the problem is still Σ_2^P-hard). Assume that $\mathcal{F}$ for a given instance of STRATCOMP contains the following facts:

- $\texttt{company}(\texttt{c})$ for each $c \in C$
- $\texttt{prod\_by}(g, c_j, c_k)$ if $\{c_i \mid g \in G_i\} = \{c_j, c_k\}$ and $c_j \neq c_k$
  $\texttt{prod\_by}(g, c_j, c_j)$ if $\{c_i \mid g \in G_i\} = \{c_j\}$
- $\texttt{contr\_by}(c_i, c_k, c_m, c_n)$ if $c_i \in C$ and $O_i = \{c_k, c_m, c_n\}$ and $c_k \neq c_m \neq c_n$
  $\texttt{contr\_by}(c_i, c_k, c_m, c_m)$ if $c_i \in C$ and $O_i = \{c_k, c_m\}$ and $c_k \neq c_m$
  $\texttt{contr\_by}(c_i, c_k, c_k, c_k)$ if $c_i \in C$ and $O_i = \{c_k\}$

We next present a program, $\mathcal{P}_{strat1}$, which solves the complex problem STRATCOMP in a surprisingly elegant way by only two rules:

$$r_{s1}: \quad \texttt{strat}(\texttt{Y}) \vee \texttt{strat}(\texttt{Z}) \leftarrow \texttt{prod\_by}(\texttt{X}, \texttt{Y}, \texttt{Z}). \quad \Big\} \textbf{Guess}$$

$$r_{s2}: \quad \texttt{strat}(\texttt{W}) \leftarrow \texttt{contr\_by}(\texttt{W}, \texttt{X}, \texttt{Y}, \texttt{Z}), \texttt{strat}(\texttt{X}), \texttt{strat}(\texttt{Y}), \texttt{strat}(\texttt{Z}). \quad \Big\} \textbf{“Check”}$$

Here $\texttt{strat}(\texttt{X})$ means that company $\texttt{X}$ is a strategic company. The guessing part $\mathcal{G}$ of the program consists of the disjunctive rule r_{s1}, and the checking part $\mathcal{C}$ consists of the normal rule r_{s2}. The program $\mathcal{P}_{strat1}$ exploits the minimization which is inherent to the semantics of answer sets for the check whether a candidate set C' of companies that produces all goods and obeys company control is also minimal with respect to this property.

The guessing rule r_{s1} intuitively selects one of the companies $\texttt{c}_1$ and $\texttt{c}_2$ that produce some item $\texttt{g}$, which is described by $\texttt{prod\_by}(\texttt{g}, \texttt{c}_1, \texttt{c}_2)$. If there were no company control information, minimality of answer sets would then naturally ensure that the answer sets of $\mathcal{F} \cup \{r_{s1}\}$ correspond to the strategic sets; no further checking would be needed. However, in case such control information (given by facts $\texttt{contr\_by}(\texttt{c}, \texttt{c}_1, \texttt{c}_2, \texttt{c}_3)$) is available, the rule r_{s2} in the program checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets.

Proposition 3.4.1
The answer sets of $\mathcal{P}_{strat1} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$. Thus, a company $\texttt{c}$ is strategic if $\texttt{strat}(\texttt{c})$ is a brave consequence, i.e. $\mathcal{P}_{strat1} \cup \mathcal{F} \models_b \texttt{strat}(\texttt{c})$ holds (see Definition 2.2.4).

An important note here is that the checking “constraint” r_{s2} interferes with the guessing rule r_{s1}: applying r_{s2} may “spoil” the minimal answer set generated by rule r_{s1}. For example, suppose the guessing part gives rise to a ground rule r_{sg1}

$$\texttt{strat(c1)} \ \vee \ \texttt{strat(c2)} \leftarrow \texttt{prod\_by(g, c1, c2)}.$$

and the fact $\texttt{prod\_by(g, c1, c2)}$ is given in $\mathcal{F}$. Now suppose the rule is satisfied in the guessing part by making $\texttt{strat(c1)}$ true. If, however, in the checking part an instance of rule r_{s2} is applied which derives $\texttt{strat(c2)}$, then the application of the rule r_{sg1} to derive $\texttt{strat(c1)}$ is invalidated, as the minimality of answer sets implies that not both $\texttt{strat(c1)}$ and $\texttt{strat(c2)}$ can be derived from r_{sg1}.

"Feedback" or "influence" of the checking part $\mathcal{C}$ on the guessing part $\mathcal{G}$, in terms of literals which are derived in $\mathcal{C}$ and invalidate the application of rules in $\mathcal{G}$ or make further rules in $\mathcal{G}$ applicable (and thus change the guess), can be made precise in terms of a "potentially uses" relation [44, 56] and a "splitting set" [83]. Such interference is in fact needed to solve STRATCOMP in the way we have outlined, if $\mathcal{C}$ does not contain disjunctive rules. This follows from complexity considerations: If $\mathcal{C}$ had no influence on $\mathcal{G}$ (augmented by $\mathcal{F}$), then the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}$ would just be the answers sets of $\mathcal{C} \cup A_0$, where A_0 is an answer set of $\mathcal{G} \cup \mathcal{F}$. Furthermore, if $\mathcal{C}$ contains no disjunctive rule and $\mathcal{G}$ has complexity in NP (e.g., if $\mathcal{G}$ is, as in this case, head-cycle-free [11]), along with A_0 an answer set A of $\mathcal{C} \cup A_0$ can be guessed and both A_0 and A can be verified in polynomial time. The complexity of deciding whether an atom belongs to some answer set of such a feedback-free program is thus in NP. Now we recall that a particular company is strategic iff it belongs to some strategic set, and deciding this is a Σ_2^P-complete problem. Unless $\Sigma_2^P = \text{NP}$, a hypothesis that is widely believed to be false, we can thus conclude that it is impossible to solve the STRATCOMP problem by a program whose answer sets correspond to the strategic sets, but in which $\mathcal{C}$ has no influence on $\mathcal{G}$.

In general, if a program encodes a problem that is Σ_2^P-complete, then the checking part $\mathcal{C}$ must contain disjunctive rules unless $\mathcal{C}$ has influence on the guessing part $\mathcal{G}$ (or $\mathcal{G}$ has Σ_2^P-complexity). In particular, if the above program $\mathcal{P}_{strat1}$ is rewritten to eliminate such influence, then further disjunctive rules must be added. This is exemplified by the following program $\mathcal{P}_{strat2}$, which expresses the strategic sets in the generic Guess & Check-paradigm, where the guessing and the checking part are clearly separated:

$$\texttt{strat(X)} \vee \neg\, \texttt{strat(X)} \leftarrow \texttt{company(X)}. \qquad \} \textbf{ Guess}$$

$$\left.\begin{array}{l}
\leftarrow \texttt{prod_by(X, Y, Z), not strat(Y), not strat(Z).} \\
\leftarrow \texttt{contr_by(W, X, Y, Z), strat(X), strat(Y), strat(Z), not strat(W).} \\
\texttt{min(X)} \vee \texttt{strat1(X, Y)} \vee \texttt{strat1(X, Z)} \leftarrow \texttt{prod_by(G, Y, Z), strat(X).} \\
\texttt{min(X)} \vee \texttt{strat1(X, C)} \leftarrow \texttt{contr_by(C, W, Y, Z), strat(X),} \\
\qquad\qquad\qquad\qquad \texttt{strat1(X, W), strat1(X, Y), strat1(X, Z).} \\
\texttt{strat1(X, Y)} \leftarrow \texttt{min(X), strat(X), strat(Y), X <> Y.} \\
\leftarrow \texttt{strat(X), not min(X).} \\
\leftarrow \texttt{strat1(X, Y)}, \neg\, \texttt{strat(Y).} \\
\leftarrow \texttt{strat1(X, X).}
\end{array}\right\} \textbf{ Check}$$

Recall that company(X), which does not occur in $\mathcal{P}_{strat1}$, means that X is a company. The guessing rule selects a subset $C' \subseteq C$ of the companies that serves as a candidate strategic set. The checking part of $\mathcal{P}_{strat2}$ is far more complicated than in the previous examples. The first two constraints ensure that C' is "production-preserving" as defined in Definition 3.4.1, i.e. that all goods can be produced and that C' is closed under the company control relation.

The remaining part of the program then checks for the minimality of C'. The principal idea is trying to find a subset $C''_{\mathtt{c}} \subseteq C' \setminus \{\mathtt{c}\}$ such that $C''_{\mathtt{c}}$ is production-preserving. If such a $C''_{\mathtt{c}}$ exists, then C' is not a strategic set. In this case, min(c) cannot be derived. On the other hand, if $C''_{\mathtt{c}}$ does not exist for some c, then min(c) will be derived. Only if min(d) is derived for each $d \in C'$, no $C''_{\mathtt{c}}$ exists and therefore C' is minimal, and only in this case the check succeeds.

Note that the checking part of $\mathcal{P}_{strat2}$ contains disjunctive rules. As we have discussed above, this is actually necessary given the one-way relationship of the guessing and the checking parts in $\mathcal{P}_{strat2}$.

Proposition 3.4.2
The answer sets of $\mathcal{P}_{strat2} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$. So, as in Proposition 3.4.1, a company c *is strategic if* strat(c) *is a brave consequence, i.e. $\mathcal{P}_{strat1} \cup \mathcal{F} \models_b$ strat(c) holds.*

Proof It is easy to see that the answer sets of the guessing part $\mathcal{G}$ of $\mathcal{P}_{strat2}$ correspond (one-to-one) to all $C' \subseteq C$, more precisely, $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$ for exactly one $A \in \mathcal{AS}(\mathcal{G} \cup \mathcal{F})$, and conversely, each $A \in \mathcal{AS}(\mathcal{G} \cup \mathcal{F})$ corresponds to exactly one C'.

Let $\mathcal{C}_1$ consist of the first two constraints in $\mathcal{P}_{strat2}$:

$\leftarrow$ prod_by(X, Y, Z), not strat(Y), not strat(Z).
$\leftarrow$ contr_by(W, X, Y, Z), strat(X), strat(Y), strat(Z), not strat(W).

It is straightforward to see that $\mathcal{C}_1$ admits only production-preserving sets $C' \subseteq C$, so there is a ono-to-one correspondence between production-preserving sets and answer sets of $\mathcal{C}_1 \cup \mathcal{G} \cup \mathcal{F}$.

Among the production-preserving sets, only subset-minimal sets are strategic sets. More precisely, if for a production-preserving set C' a set $C''_{\mathtt{c}} \subseteq C' \setminus \{\mathtt{c}\}$ exists for some $\mathtt{c} \in C'$ such that $C''_{\mathtt{c}}$ is production-preserving, then C' violates this minimality criterion.

We will proceed as follows: We will show that for any production-preserving $C' \subseteq C$, if such a $C''_{\mathtt{c}}$ exists, then no answer set $A \in \mathcal{AS}(\mathcal{P}_{strat2} \cup \mathcal{F})$ exists such that $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$, while if such a $C''_{\mathtt{c}}$ does not exist, then a unique answer set $A \in \mathcal{AS}(\mathcal{P}_{strat2} \cup \mathcal{F})$ exists such that $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$.

First, let us consider the case in which a $C''_{\mathtt{c}}$ exists (and thus C' is not a strategic set). We will show that in this case no A with $\mathtt{min}(\mathtt{c}) \in A$ and $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$ can be an answer set of $\mathcal{P}_{strat2} \cup \mathcal{F}$, as A would not be a minimal answer set of $(\mathcal{P}_{strat2} \cup \mathcal{F})^A$. On the other hand, we will show that no A with $\mathtt{min}(\mathtt{c}) \notin A$ and $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$ can be an answer set either.

Let us assume first that $\mathtt{min}(\mathtt{c}) \in A$ holds. Then also $\mathtt{strat1}(\mathtt{c}, x) \in A$ for each $x \in C' \setminus \{\mathtt{c}\}$ because of rule instances of the form

$$\mathtt{strat1}(\mathtt{c}, x) \leftarrow \mathtt{min}(\mathtt{c}), \mathtt{strat}(\mathtt{c}), \mathtt{strat}(x), \mathtt{c} <> x.$$

We will next show that $A' = A \setminus (\{\mathtt{min}(\mathtt{c})\} \cup \{\mathtt{strat1}(\mathtt{c}, x) \mid x \in C \setminus C''_{\mathtt{c}}\})$ satisfies all rules in $(\mathcal{P}_{strat2} \cup \mathcal{F})^A$ provided that A satisfies them. Only rules in which atoms from $A \setminus A'$ occur in the head need to be considered, as $\leftarrow \mathtt{strat}(\mathtt{c}), \mathtt{not}\ \mathtt{min}(\mathtt{c}). \notin (\mathcal{P}_{strat2} \cup \mathcal{F})^A$ holds.

For rule instances of the form

$$\mathtt{min}(\mathtt{c}) \vee \mathtt{strat1}(\mathtt{c}, x) \vee \mathtt{strat1}(\mathtt{c}, y) \leftarrow \mathtt{prod\_by}(g, x, y), \mathtt{strat}(\mathtt{c}).$$

for which the body is true in A, at least one of the companies x, y is in $C''_{\mathtt{c}}$ (otherwise $C''_{\mathtt{c}}$ would not be production-preserving), and therefore the rule is satisfied in A'.

For rule instances of the form

$$\begin{aligned}\mathtt{min}(\mathtt{c}) \vee \mathtt{strat1}(\mathtt{c}, v) \leftarrow\ & \mathtt{contr\_by}(v, w, x, y), \mathtt{strat}(\mathtt{c}),\\ & \mathtt{strat1}(\mathtt{c}, w), \mathtt{strat1}(\mathtt{c}, x), \mathtt{strat1}(\mathtt{c}, y).\end{aligned}$$

for which the body is true in A, either all w, x, y are in $C''_{\mathtt{c}}$, then also $v \in C''_{\mathtt{c}}$ (otherwise $C''_{\mathtt{c}}$ would not be production-preserving), hence the rule is satisfied also in A', or if not all w, x, y are in $C''_{\mathtt{c}}$, the body is not true in A', also satisfying the rule.

Finally, rule instances of the form

$$\mathtt{strat1}(\mathtt{c}, x) \leftarrow \mathtt{min}(\mathtt{c}), \mathtt{strat}(\mathtt{c}), \mathtt{strat}(x), \mathtt{c} <> x.$$

for which the body is true in A, have a false body in A', satisfying these rules.

We have shown that whenever $\mathtt{min}(\mathtt{c}) \in A$ holds such that $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$, a smaller A' exists which satisfies all rules in $(\mathcal{P}_{strat2} \cup \mathcal{F})^A$, so such an A cannot be an answer set of $\mathcal{P}_{strat2} \cup \mathcal{F}$. On the other hand, if $\mathtt{min}(\mathtt{c}) \notin A$ holds and $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$, then A does not satisfy the constraint

$$\leftarrow \mathtt{strat}(\mathtt{c}), \mathtt{not}\ \mathtt{min}(\mathtt{c}).$$

and hence such an A cannot be an answer set of $\mathcal{P}_{strat2} \cup \mathcal{F}$ either. In total we have that no answer set A exists such that $C' = \{\mathtt{c} \mid \mathtt{strat}(\mathtt{c}) \in A\}$, if some $C''_{\mathtt{c}}$ is a production-preserving set.

Now assume that a production-preserving $C''_{\mathtt{c}}$ does not exist for any $\mathtt{c} \in C$, which means that C' is a strategic set. We will show that $A = \{f \mid f. \in \mathcal{F}\} \cup \{\mathtt{strat}(x) \mid x \in C'\} \cup \{\neg\ \mathtt{strat}(x) \mid x \in C \setminus C'\} \cup \{\mathtt{min}(x) \mid x \in C'\} \cup \{\mathtt{strat1}(\mathtt{c}, x) \mid x \in C' \setminus \{\mathtt{c}\}\}$ is an answer set of $\mathcal{P}_{strat2} \cup \mathcal{F}$, i.e., (i) A satisfies all rules of $\mathcal{P}_{strat2} \cup \mathcal{F}$ (and thus of $(\mathcal{P}_{strat2} \cup \mathcal{F})^A$) and (ii) no subset of A satisfies all rules of $(\mathcal{P}_{strat2} \cup \mathcal{F})^A$.

Concerning (i), the instances of the guessing rules are clearly satisfied and instances of the first two constraints in the check are satisfied because C' is production-preserving. Instances of the two rules with $\mathtt{min}(\mathtt{X})$ in their heads and $\mathtt{strat}(\mathtt{X})$ in their bodies are satisfied because $\mathtt{min}(x) \in A$ holds for $x \in C'$ and $\mathtt{strat}(x) \notin A$ holds for $x \in C \setminus C'$. Finally, the instances of the rule with $\mathtt{strat1}(\mathtt{X}, \mathtt{Y})$ and the instances of the last three constraints are satisfied by the construction of A.

To see (ii), observe that $\{f \mid f. \in \mathcal{F}\}$, $\{\mathtt{strat}(x) \mid x \in C'\}$, and $\{\neg\ \mathtt{strat}(x) \mid x \in C \setminus C'\}$ cannot be omitted from A, as either the facts in $\mathcal{F}$ or instances of the guessing rule could not be satisfied any more. Furthermore, $\{\mathtt{min}(x) \mid x \in C'\}$ must be in A, otherwise instances of $\leftarrow \mathtt{strat}(\mathtt{c}), \mathtt{not}\ \mathtt{min}(\mathtt{c})$. could not be satisfied. Finally, $\{\mathtt{strat1}(\mathtt{c}, x) \mid x \in C' \setminus \{\mathtt{c}\}\}$ must be included in A in order to satisfy instances of the last rule.

In total, we have shown that an answer set A exists if no $C''_{\mathtt{c}}$ exists. It straightforward to see that this A is the unique answer set, since starting from $\{\mathtt{strat}(x) \mid x \in C'\}$, all atoms in A are necessarily contained in any answer set, as shown for (ii) above. Since A is an answer set, no superset of A can be an answer set, and so A is unique, which concludes our proof. □

Note that STRATCOMP can not be expressed by a fixed normal logic program uniformly on all collections of facts $\mathtt{contr\_by}(\mathtt{c}, \mathtt{c1}, \mathtt{c2}, \mathtt{c3})$ and $\mathtt{prod\_by}(\mathtt{p}, \mathtt{c1}, \mathtt{c2})$ (unless NP $= \Sigma_2^P$, an unlikely scenario).

Further examples of programs in which the checking part necessarily contains disjunctive rules can be found in [56].

An interesting issue for future work is finding a general methodology of creating Guess & Check-programs for problems in Σ_2^P. This methodology could start from a formulation in second order logic. In general, such Guess & Check-programs must contain a means to derive some atoms ($\mathtt{min}(\mathtt{c})$ in $\mathcal{P}_{strat2}$) in order to represent some property of a candidate. If some candidate does not have the required property, then it must be a proper subset of all candidates having the property. So in general a "saturation rule" is needed, which ensures this superset relationship (in $\mathcal{P}_{strat2}$ this saturation rule is $\mathtt{strat1}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{min}(\mathtt{X}), \mathtt{strat}(\mathtt{X}), \mathtt{strat}(\mathtt{Y}), \mathtt{X} <> \mathtt{Y}.$).

Chapter 4

System Architecture

In this chapter, we describe the general architecture of the ASP system DLV. While DLV's principal language is the one defined in Chapter 2, DLV has been enhanced by several high-level extensions and front-ends, which will be described in Part II. Even parts of the core language are implemented by means of front-ends, exploiting Proposition 2.2.1 and Proposition 2.2.2. Because of these results, it is sufficient to deal with programs without classical negation in the DLV kernel, and also queries are not dealt with there.

Figure 4.1 on the following page illustrates DLV's general architecture. The general flow in this illustration is top-down. The principal "User Interface" is command-line oriented, but also a Graphical User Interface (GUI) for Answer Sets and most front-ends is available as well. Subsequently, front-end transformations (also combinations are feasible) might be performed. Input data can be supplied by regular files, and also by Oracle or Objectivity databases. Each time an answer set is found by the DLV kernel (the shaded part in the figure), "Filtering" is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The DLV kernel consists of three major components: The "Intelligent Grounding," "Model Generator," and "Model Checker" modules share a principal datastructure, the "Ground Program". It is created by the "Intelligent Grounding", and used by the "Model Generator" and the "Model Checker." In general, the "Intelligent Grounding" does not create the full ground program $Ground(\mathcal{P})$ as defined in Definition 2.2.1, but $\mathcal{P}_{ground} \subseteq Ground(\mathcal{P})$ such that $\mathcal{AS}(\mathcal{P}_{ground}) = \mathcal{AS}(Ground(\mathcal{P}))$. Details on this module can be found in [1, 64]. The "Model Generator" creates candidate answer sets, and will be described in detail in Chapter 5. Finally, the "Model Checker" tests the stability, and most importantly the minimality of the generated candidates. Details on this component can be found in [76]. The whole system is described also in [104].

Ideally, the performance of a system reflects the complexity of the prob-

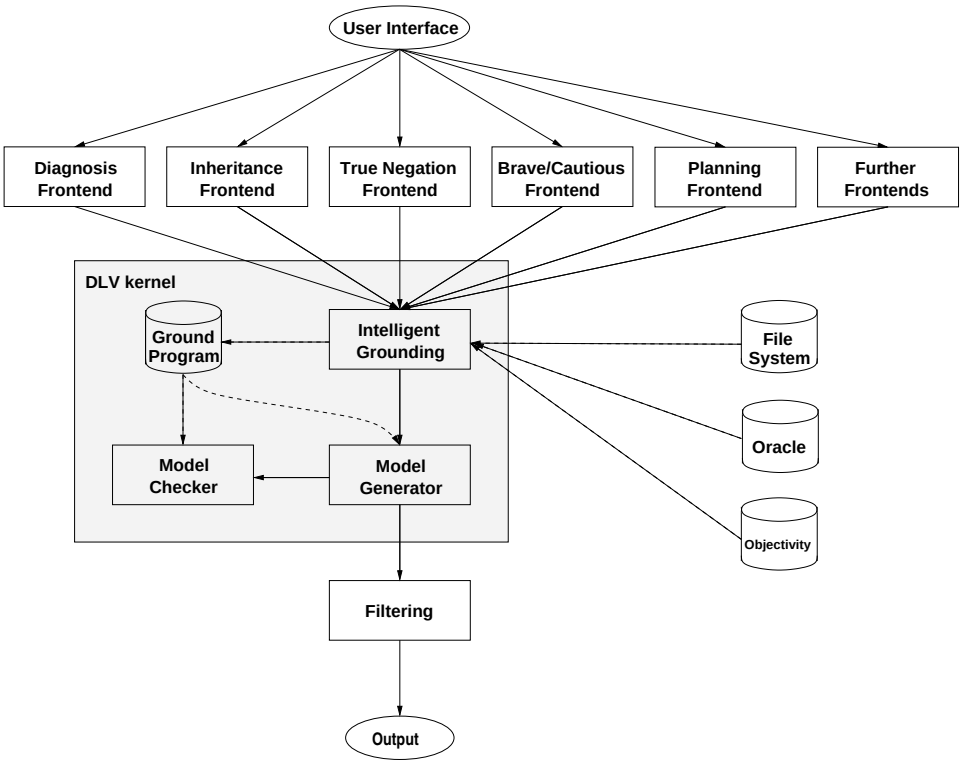

Figure 4.1: The System Architecture of DLV

lem at hand, such that "easy" problems (say, some of polynomial complexity, which are recognizable with reasonable efforts) are solved fast, while more time is spent for more difficult problems (like NP-hard problems), and indeed the DLV system is designed in this direction. For example, stratified normal programs (which have polynomial complexity) are evaluated fast using techniques from the fields of deductive databases and logic programming. Note that in general, that is without considering syntactical restrictions, it is impossible to detect whether a program uniformly encodes an "easy" (even trivial) problem, since this task is clearly undecidable.

The architecture of the DLV kernel closely reflects complexity results for various subsets of our language: As mentioned before, the Intelligent Grounding (IG) module is able to completely solve many problems of polynomial complexity, and in particular all normal stratified programs; the Model Generator (together with the Grounding) is capable of solving NP-hard problems; finally, adding the Model Checker is needed to solve Σ_2^P-hard problems.

Upon startup, the DLV kernel or one of the front-ends parses the input specified by the user and transforms it into the internal data structures of

DLV. In both cases, this takes only linear time and memory in the input size.

Using differential (and other advanced) database techniques together with suitable data structures, the *Intelligent Grounding* module then efficiently generates a subset of the ground instantiation of that input which has the same answer sets, but is much smaller in general. For example, in case of a stratified program, the IG module already computes the single answer set.

For harder problems, most of the computation is performed by the Model Generator and the Model Checker. Roughly, the *Model Generator* (MG) produces some "candidate" answer sets (models). The stability of each of them is subsequently verified by the Model Checker.

The *Model Checker* (MC) verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is well-known to be co-NP-complete (cf. [56]). However, recent studies [11, 12, 81] showed that minimal model checking — the hardest part of answer set (or stable model) checking — can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs [11].

The MC satisfies the above complexity bounds. Indeed (a) it terminates in polynomial time on every HCF program; and (b) it always runs in polynomial space and single exponential time. Moreover, even on general (non-HCF) programs, the MC limits the inefficient part of the computation to the modules that are not HCF. Note that it may well happen that only a very small part of the program is not HCF.

We do not give a detailed description of the various components here, except for the "Model Generator", which will be dealt with in detail in the following chapter. For the other modules, we refer to [104].

Chapter 5

Model Generation

In this chapter, we will describe the Model Generator (MG) module of the DLV system in full detail. After an overview of the general design of MG, we focus on the two key parts, computing deterministic consequences and choice, concluding with an experimental analysis of various choice heuristics and optimizations.

5.1 General Design

In this section, we briefly review the model generation algorithm of the DLV system. The Model Generator (MG) produces a set of interpretations that are "candidates" for answer sets, which are then submitted to the Model Checker for verification. The Model Generator essentially relies on a backtracking technique which spans the search space for computing all answer sets, based on the Davis-Putnam-Logemann-Loveland procedure [33, 32] for computing models which satisfy a propositional formula. It employs a number of techniques for pruning the search space, and makes use of sophisticated data structures in order to improve the overall efficiency.

During the computation, we deal with four-valued interpretations, which we will refer to as *partial*[1] interpretations, as opposed to total interpretations as defined in Chapter 2. We consider the following truth values: *true (T), must-be-true (M* or *mbt), undefined (U),* and *false (F).* The effect of negation as failure on these truth values is given by the following table:

x	not x
T	F
M	F
U	U
F	T

[1]Note that in the literature partial interpretations are usually three-valued, without the must-be-true value.

We associate the total ordering $T > M > U > F$ to the truth values, this ordering is called the *truth-ordering*. The partial order $\preceq$ on the four truth values is defined through the following relationships: $U \preceq F$, $U \preceq M$, $U \preceq T$, and $M \preceq T$; moreover, $X \preceq X$ for any $X \in \{F, U, M, T\}$. This partial order is called the *knowledge ordering*. The two orderings are illustrated in Figure 5.1.

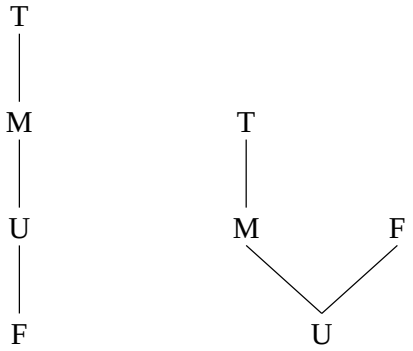

Figure 5.1: Truth Ordering (left) and Knowledge Ordering (right)

Definition 5.1.1
A partial interpretation $\mathcal{I}$ for $\mathcal{P}$ is a total mapping from $B_\mathcal{P}$ to $\{T, M, U, F\}$. We will use a functional notation $\mathcal{I}(\mathtt{p}) = V$ to denote that $\mathtt{p} \in base\mathcal{P}$ is mapped to the truth-value V by $\mathcal{I}$. We will also employ a set-based notation

$$\mathcal{I} = \{\mathtt{p} \rightarrow V \mid \mathcal{I}(\mathtt{p}) = V \neq U\}$$

for $\mathcal{I}$, in which undefined classical literals are not explicitly represented.

Such a partial interpretation can also be seen as inducing four partitions of $B_\mathcal{P}$. We will denote these partitions as

$$\begin{array}{rcl} \mathcal{I}^T & = & \{a \mid a \rightarrow T \in \mathcal{I}\} \\ \mathcal{I}^M & = & \{a \mid a \rightarrow M \in \mathcal{I}\} \\ \mathcal{I}^U & = & \{a \mid a \rightarrow U \in \mathcal{I}\} \\ \mathcal{I}^F & = & \{a \mid a \rightarrow F \in \mathcal{I}\} \end{array}$$

Note that $\mathcal{I}^T$, $\mathcal{I}^M$, $\mathcal{I}^U$, $\mathcal{I}^F$ are pairwise disjoint sets.

We next define several valuation functions.

Definition 5.1.2
Given a partial interpretation $\mathcal{I}$ for $\mathcal{P}$, the valuation function $val_\mathcal{I}()$ is defined as follows: For any atom $p \in B_\mathcal{P}$, $val_\mathcal{I}(p) = \mathcal{I}(p)$, and $val_\mathcal{I}(\mathtt{not}\ p) = \mathtt{not}\ val_\mathcal{I}(p)$. For a set S of literals, we define its valuation function $val^H_\mathcal{I}(S)$

(if S is to be interpreted as a disjunction as in the head of a rule) and $val^B_{\mathcal{I}}(S)$ (if S is to be interpreted as a conjunction as in the body of a rule):

$$
\begin{aligned}
val^H_{\mathcal{I}}(S) &= \begin{cases} F & \text{if } S = \emptyset \\ \max_{p \in S} val_{\mathcal{I}}(p) & \text{if } S \neq \emptyset \end{cases} \\
val^B_{\mathcal{I}}(S) &= \begin{cases} T & \text{if } S = \emptyset \\ \min_{p \in S} val_{\mathcal{I}}(p) & \text{if } S \neq \emptyset \end{cases}
\end{aligned}
$$

So $val^H_{\mathcal{I}}(S)$ is F if S is empty or if all elements in S are F. $val^H_{\mathcal{I}}(S)$ is U if at least one element in S is U and none is T or M. $val^H_{\mathcal{I}}(S)$ is M if at least one element in S is M and none is T. Finally $val^H_{\mathcal{I}}(S)$ is T if at least one element in S is T.

Symmetrically, $val^B_{\mathcal{I}}(S)$ is T if S is empty or if all elements in S are T, it is M if at least one element in S is M and none is U or F, it is U if at least one element in S is U and none is F, and finally it is F if at least one element is F.

We will next define the notion of satisfaction for ground rules with respect to partial interpretations.

Definition 5.1.3
A ground rule r is satisfied *w.r.t. $\mathcal{I}$ if the truth value of its head is not less than the truth value of its body, i.e. $val^H_{\mathcal{I}}(H(r)) \geq val^B_{\mathcal{I}}(B(r))$.*

Basically, the MG works as follows: (1) Derive what is deterministically derivable from the program, (2) make an "educated" guess for one of those literals which have not been decided yet, and (3) propagate the consequences of this choice. To formalize what we have called "educated guess", we introduce the concept of a possibly-true (PT) literal:

Definition 5.1.4
Let $\mathcal{I}$ be a partial interpretation for $\mathcal{P}$.

A *positive PT literal* of $\mathcal{P}$ w.r.t. $\mathcal{I}$ is a positive literal p such that $U \leq \mathcal{I}(p) \leq M$ and there exists a rule $r \in \mathit{Ground}(\mathcal{P})$ for which all of the following conditions hold:

1. $p \in H(r)$;
2. $val^H_{\mathcal{I}}(H(r)) < T$ (i.e., the head is not true w.r.t. $\mathcal{I}$);
3. $val^B_{\mathcal{I}}(\mathcal{B}(r)) = T$ (i.e., the body is true w.r.t. $\mathcal{I}$).

A *negative PT literal* of $\mathcal{P}$ w.r.t. $\mathcal{I}$ is an undefined negative literal `not q` such that there exists a rule $r \in \mathit{Ground}(\mathcal{P})$ for which all of the following conditions hold:

1. `not q` $\in \mathcal{B}(r)$;

2. $val^H_{\mathcal{I}}(H(r)) < T$ (i.e., the head is not true w.r.t. $\mathcal{I}$);

3. $val^B_{\mathcal{I}}(B^+(r)) = T$ (i.e., the body is true w.r.t. $\mathcal{I}$).

4. $val^B_{\mathcal{I}}(B^-(r)) \leq U$ (i.e., no negative body literal is false w.r.t. $\mathcal{I}$).

The set of all (positive and negative) PT literals of $\mathcal{P}$ w.r.t. $\mathcal{I}$ is denoted by $PT_{\mathcal{P}}(\mathcal{I})$.

Example 5.1.1
Consider the program $\mathcal{P}_8 = \{\mathtt{a} \vee \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}., \mathtt{e} \leftarrow \mathtt{c}, \mathtt{not\ f}.\}$ *and let* $\mathcal{I}_8 = \{\mathtt{c} \rightarrow T, \mathtt{d} \rightarrow F\}$ *be an interpretation for* $\mathcal{P}_8$*. Then, three PT literals of* $\mathcal{P}_8$ *exist w.r.t.* $\mathcal{I}_8$*:* `a`*,* `b` *and* `not f`*.*

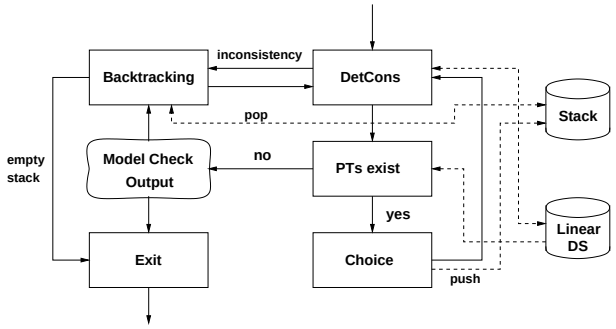

Figure 5.2: Model Generator — Basic Structure

The architecture of the Model Generator is illustrated in Figure 5.2. The solid arcs represent flow control, while the dashed arcs represent data access and manipulation. The principal datastructures are "Linear Datastructures" which represent the ground program and a current partial interpretation using similar techniques as in the Dowling-Gallier algorithm [41, 96]. The stack can store PT literals and states of the "Linear Datastructures" (using differential methods).

Initially, the "Linear Datastructures" are built and initialized (the initial partial interpretation maps all atoms to U). Then *DetCons* is invoked and derives atoms which are contained in each answer set of the input program. Note that *DetCons* is also used to detect inconsistencies, i.e. the situations in which no answer set is an extension of the current partial interpretation. If such an inconsistency is determined, "Backtracking" is invoked, which immediately leads to "Exit" if no choices had been taken so far, as the "Stack" is empty in this case.

If initially no inconsistency has been detected by *DetCons*, then some answer set might be among the extensions of the current partial interpretation. We then compute the set of PT literals. If this set is not empty,

one of them is chosen by the module "Choice", it is pushed onto the stack together with the current status of the relevant datastructures, and the current partial interpretation is modified to valuate the chosen literal to T by mapping it to T if it is a positive PT literal or to F if it is a negative PT literal. Subsequently, the process is started from the beginning by invoking *DetCons*.

If, however, the set of PT literals is empty, then at most one answer set (the totalization in which all undefined atoms are turned into false) is an extension of the current partial interpretation. In this case, this answer set candidate is submitted to the Model Checker, which delivers the answer set to "Filtering" and "Output" upon success (see Figure 4.1 on page 25). Subsequently, if the Model Check did not succeed or if more answer sets are to be computed, "Backtracking" is invoked. If the stack is not empty, then the topmost element is popped, and all datastructures are restored to the status before the choice of the popped element and the partial interpretation is modified to associate an inverse truth value of the popped element, and *DetCons* is invoked again.

The actual algorithm for computing answer sets is presented in Figure 5.3. There, *isAnswerSet* is a function which implements the Model Checker module. For details on the basic Model Generator architecture (in particular on the notion of PT literals and for correctness analyses) we refer to [81, 57, 59, 104].

In the sequel, we will focus on the *DetCons* and "Choice" modules, as they are most critical to performance. We will define a powerful *DetCons* procedure and complement it with suitable heuristics. Finally, experimental analyzes show the effectiveness of our approach.

5.2 Deterministic Consequences

In this section, we describe the function *DetCons* which, given a ground program $\mathcal{P}$ and a four-valued interpretation $\mathcal{I}$, derives certain knowledge concerning the literals which are still undefined in $\mathcal{I}$. In other words, *DetCons* extends the interpretation $\mathcal{I}$ in a deterministic way, such that *every* answer set containing $\mathcal{I}$ also contains its extension computed by *DetCons*. This function is used in the `DLV` system in two ways: First, it is called at the very beginning of the computation (with $\emptyset$ as the interpretation), in order to derive literals which belong to all answer sets. Later, each time an assumption has been made, *DetCons* is invoked in order to derive what follows from it. Thus, *DetCons* is crucial for the efficiency of the system: the larger the interpretation it derives, the smaller the remaining search space. We will proceed by first giving a formal definition of the concepts and operators employed in *DetCons*, and subsequently we will present the actual algorithm which is used in `DLV`.

Algorithm ComputeAnswerSets
Input: A ground DLP program $\mathcal{P}$.
Output: The answer sets of $\mathcal{P}$ (if some exist).

Procedure *ComputeAnswerSets*($\mathcal{I}$: Interpretation)
(* The procedure outputs all answer sets of $\mathcal{P}$ *)
var Q: SetOfLiterals; L: Literal;
(1) *DetCons*($\mathcal{P}, \mathcal{I}$,contradiction);
(2) **if** contradiction **then** **exit procedure**;
(3) **if** ($PT_{\mathcal{P}}(\mathcal{I}) = \emptyset$) **then** (* $\mathcal{I}^T \cup \mathcal{I}^M$ is a model of $\mathcal{P}$ *)
(4) **if** ($\mathcal{I}^M = \emptyset$) **and** *isAnswerSet*($\mathcal{P}$, $\mathcal{I}^T$) **then**
(5) **output** $\mathcal{I}^T$; (* $\mathcal{I}^T$ is an answer set *)
else
(6) Take a literal L from $PT_{\mathcal{P}}(\mathcal{I})$;
(* Assume the truth of a PT literal *)
(7) **if** L is a negative literal `not p` **then**
(8) $\mathcal{I}$(`p`) := F;
else (* L is a positive literal *)
(9) $\mathcal{I}(L) := T$;
(10) *ComputeAnswerSets*($\mathcal{I}$);
(* At this point all answer sets containing $\mathcal{I} \cup \{L\}$ have been generated *)
(* L must be false in following computations *)
(11) **if** L is a negative literal `not p` **then**
(12) $\mathcal{I}(p) := M$;
else
(13) $\mathcal{I}(L) = F$;
(14) *ComputeAnswerSets*($\mathcal{I}$);
end_procedure

var $\mathcal{I}$: Interpretation;
begin (* Main *)
$\mathcal{I}^T := \emptyset$; $\mathcal{I}^F := \emptyset$; $\mathcal{I}^M := \emptyset$; $\mathcal{I}^U := B_{\mathcal{P}}$;
ComputeAnswerSets($\mathcal{I}$);
end.

Figure 5.3: Algorithm for the Computation of Answer Sets

5.2.1 Basic Concepts

Let us first formalize the notion of a partial interpretation extending another one. Informally, true atoms which are true in some interpretations, should also be true in extending interpretations. Likewise, false atoms should remain false, and must-be-true atoms should either be still must-be-true or should have become true.

Definition 5.2.1
Given a partial interpretation $\mathcal{I}$, we define the set of partial interpretations $\mathcal{J}$ extending $\mathcal{I}$ as

$$ext(\mathcal{I}) = \{\mathcal{J} \mid \text{is a partial interpretation}, \mathcal{J}^T \supseteq \mathcal{I}^T, \mathcal{J}^M \cup \mathcal{J}^T \supseteq \mathcal{I}^M, \mathcal{J}^F \supseteq \mathcal{I}^F\}$$

Intuitively, all partial interpretations in $ext(\mathcal{I})$ represent more or equally concrete knowledge than $\mathcal{I}$ does.

Next, we define a connection from partial interpretations to total interpretations as defined in Chapter 2, called totalizations. True and must-be-true atoms must be contained in all totalizations, while false atoms must not be contained in any totalization. The connection in the other direction is achieved by simply mapping atoms contained in the total interpretation to T and those which do not occur in it to F:

Definition 5.2.2
Let $\mathcal{P}$ be a ground program, $\mathcal{I}$ a partial interpretation, and I a total interpretation. We define:

$$total_{\mathcal{P}}(\mathcal{I}) = \{I \mid I \subseteq B_{\mathcal{P}} \wedge \mathcal{I}^T \cup \mathcal{I}^M \subseteq I \wedge \mathcal{I}^F \cap I = \emptyset\}$$

$$partial_{\mathcal{P}}(I) = \{a \rightarrow T \mid a \in I\} \cup \{a \rightarrow F \mid a \in B_{\mathcal{P}} \setminus I\}$$

Intuitively, the truth value M is assigned to atoms that cannot be derived from any program rule at the current computation step, but must eventually be true in the answer set to be computed. These atoms are not immediately taken as true in order to guarantee the "supportedness" of the interpretation at hand. This is a main peculiarity of answer sets w.r.t. ordinary models: Any atom a belonging to an answer set of $\mathcal{P}$ has a rule which *supports* a. Formally, a is contained in an answer set S if and only if there exists a rule $r \in \mathcal{P}$ such that $a \in H(r)$, $H(r) \setminus \{a\}$ is $false$ w.r.t. S, and $B(r)$ is $true$ w.r.t. $S \setminus \{a\}$. Thus, enforcing supportedness is a principal difference between DLP systems and satisfiability solvers (like, e.g., the Davis-Putnam procedure).

Let us formalize the notion of supportedness.

Definition 5.2.3
Given a ground program $\mathcal{P}$, a ground atom a, a total interpretation I and a partial interpretation $\mathcal{I}$, let

$$\begin{aligned} supp_{\mathcal{P}}(a, I) &= \{r \in \mathcal{P} \mid a \in H(r) \wedge (H(r) \setminus \{a\}) \cap I = \emptyset \wedge B^+(r) \subseteq I \\ &\qquad \wedge B^-(r) \cap I = \emptyset\} \\ supp_{\mathcal{P}}(a, \mathcal{I}) &= \{r \in \mathcal{P} \mid a \in H(r) \wedge (\forall p \in H(r) \setminus \{a\} : \mathcal{I}(p) = F) \\ &\qquad \wedge (\forall p \in B^+(r) : \mathcal{I}(p) = T) \wedge (\forall p \in B^-(r) : \mathcal{I}(p) = F)\} \end{aligned}$$

denote the sets of supporting rules for a w.r.t. I and $\mathcal{I}$, respectively.

We now state two properties connecting total and partial interpretations.

Proposition 5.2.1
Given a ground program $\mathcal{P}$, a total interpretation I, a partial interpretation $\mathcal{I}$, and a ground atom $a \in B_\mathcal{P}$, the following statements hold:

$$supp_\mathcal{P}(a, I) = supp_\mathcal{P}(a, partial_\mathcal{P}(I)) \quad (5.1)$$

$$\forall I \in total_\mathcal{P}(\mathcal{I}) : \; supp_\mathcal{P}(a, \mathcal{I}) \subseteq supp_\mathcal{P}(a, I) \quad (5.2)$$

Proof

5.1 For each $r \in supp_\mathcal{P}(a, I)$ by definition of $partial_\mathcal{P}(I)$:

1. $(H(r) \setminus \{a\}) \cap I = \emptyset$ iff $\forall p \in H(r) \setminus \{a\} : p \rightarrow F \in partial_\mathcal{P}(I)$
2. $B^+(r) \subseteq I$ iff $\forall p \in H(r) : p \rightarrow T \in partial_\mathcal{P}(I)$
3. $B^-(r) \cap I = \emptyset$ iff $\forall p \in B^-(r) : p \rightarrow F \in partial_\mathcal{P}(I)$

5.2 We show that each rule in $supp_\mathcal{P}(a, \mathcal{I})$ also occurs in all $supp_\mathcal{P}(a, I)$, where $I \in total_\mathcal{P}(\mathcal{I})$.

1. If $\forall p \in H(r) \setminus \{a\} : \mathcal{I}(p) = F$ then $\forall p \in H(r) \setminus \{a\} : p \notin I$ for each $I \in total_\mathcal{P}(\mathcal{I})$ and consequently $(H(r) \setminus \{a\}) \cap I = \emptyset$.
2. If $\forall p \in B^+(r) : \mathcal{I}(p) = T$ then $\forall p \in B^+(r) : p \in I$ for each $I \in total_\mathcal{P}(\mathcal{I})$ and consequently $B^+(r) \subseteq I$.
3. If $\forall p \in B^-(r) : \mathcal{I}(p) = F$ then $\forall p \in B^-(r) : p \notin I$ for each $I \in total_\mathcal{P}(\mathcal{I})$ and consequently $B^-(r) \cap I = \emptyset$.

□

Example 5.2.1
Let $\mathcal{P}_3$ consist of the single rule r_3:

$$\mathtt{a} \vee \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}.$$

Consider the total interpretation $I_{3.1} = \{\mathtt{a}, \mathtt{c}\}$:

$$supp_{\mathcal{P}_3}(\mathtt{a}, I_{3.1}) = \{r_3\}$$
$$partial_{\mathcal{P}_3}(I_{3.1}) = \{\mathtt{a} \rightarrow T, \mathtt{b} \rightarrow F, \mathtt{c} \rightarrow T, \mathtt{d} \rightarrow F\}$$
$$supp_{\mathcal{P}_3}(\mathtt{a}, partial_{\mathcal{P}_3}(I_{3.1})) = \{r_3\}$$

Now consider the partial interpretation $\mathcal{I}_{3.2} = \{\mathtt{a} \rightarrow T, \mathtt{b} \rightarrow F, \mathtt{c} \rightarrow T, \mathtt{d} \rightarrow U\}$:

$$supp_{\mathcal{P}_3}(\mathtt{a}, \mathcal{I}_{3.2}) = \emptyset$$
$$total_{\mathcal{P}_3}(\mathcal{I}_{3.2}) = \{\{\mathtt{a}, \mathtt{c}\} = I_{3.1}, \{\mathtt{a}, \mathtt{c}, \mathtt{d}\} = I_{3.3}\}$$
$$supp_{\mathcal{P}_3}(\mathtt{a}, I_{3.1}) = \{r_3\} \supset supp_{\mathcal{P}_3}(\mathtt{a}, \mathcal{I}_{3.2})$$
$$supp_{\mathcal{P}_3}(\mathtt{a}, I_{3.3}) = \emptyset = supp_{\mathcal{P}_3}(\mathtt{a}, \mathcal{I}_{3.2})$$

We now show that each atom in an answer set must have a non-empty set of support w.r.t. the answer set, which is a well-known property of answer sets.

Proposition 5.2.2
Given a ground program $\mathcal{P}$, the following statement holds:

$$\forall A \in \mathcal{AS}(\mathcal{P}) : \forall a \in A : supp_{\mathcal{P}}(a, A) \neq \emptyset$$

Proof Assume that $A \in \mathcal{AS}(\mathcal{P})$ exists such that $\exists a \in A : supp_{\mathcal{P}}(a, A) = \emptyset$. If A is not closed under $\mathcal{P}^A$, $A \notin \mathcal{AS}(\mathcal{P})$, directly contradicting the assumption. If A is closed under $\mathcal{P}^A$, we show that $A \setminus \{a\}$ is closed under $\mathcal{P}^A$, contradicting the minimality requirement of Definition 2.2.2, and thus contradicting the assumption $A \in \mathcal{AS}(\mathcal{P})$.

Recall that if A is closed under $\mathcal{P}^A$, then for each $r \in \mathcal{P}^A$ either $B^+(r) \not\subseteq A$ or $H(r) \cap A \neq \emptyset$. If $B^+(r) \not\subseteq A$ holds, then also $B^+(r) \not\subseteq A \setminus \{a\}$ holds. If $H(r) \cap A \neq \emptyset$ and $B^+(r) \subseteq A$, let us distinguish two cases:

$a \notin H(r)$**:** Then $H(r) \cap (A \setminus \{a\}) \neq \emptyset$ holds.

$a \in H(r)$**:** Since $supp_{\mathcal{P}}(a, A) = \emptyset$ holds by assumption, $\nexists r_1 \in \mathcal{P} : a \in H(r_1) \wedge (H(r_1) \setminus \{a\}) \cap A = \emptyset \wedge B^+(r_1) \subseteq A \wedge B^-(r_1) \cap A = \emptyset$, so $\nexists r_1 \in \mathcal{P}^A : a \in H(r_1) \wedge (H(r_1) \setminus \{a\}) \cap A = \emptyset \wedge B^+(r_1) \subseteq A$ and consequently $(H(r) \setminus \{a\}) \cap A \neq \emptyset$ holds also in this case.

Thus, $A \setminus \{a\}$ is closed under $\mathcal{P}^A$. □

While $supp_{\mathcal{P}}(a, \mathcal{I})$ denotes the definitely supporting rules (i.e. those rules which are supporting rules for a in each totalization of $\mathcal{I}$), we define $psupp_{\mathcal{P}}(a, \mathcal{I})$ as those rules which could possibly support a in some totalization, so that $psupp_{\mathcal{P}}(a, \mathcal{I})$ can serve as an upper bound for $supp_{\mathcal{P}}(a, I)$ of any totalization of $\mathcal{I}$.

Definition 5.2.4
Given a ground program $\mathcal{P}$, a ground atom a and a partial interpretation $\mathcal{I}$, let the set of potentially supporting rules of a in $\mathcal{P}$ with respect to $\mathcal{I}$ be given by

$$psupp_{\mathcal{P}}(a, \mathcal{I}) = \{r \in \mathcal{P} \mid a \in H(r) \wedge \nexists p \in H(r) \setminus \{a\} : \mathcal{I}(p) \geq M \wedge \nexists p \in B^+(r) : \mathcal{I}(p) = F \wedge \nexists p \in B^-(r) : \mathcal{I}(p) = T\}$$

Proposition 5.2.3
Given a ground program $\mathcal{P}$, a ground atom $a \in B_{\mathcal{P}}$, and a partial interpretation $\mathcal{I}$, $psupp_{\mathcal{P}}(a, \mathcal{I})$ is an upper bound for $supp_{\mathcal{P}}(a, I)$ of any totalization I of $\mathcal{I}$; that is,

$$\forall I \in total_{\mathcal{P}}(\mathcal{I}) : supp_{\mathcal{P}}(a, I) \subseteq psupp_{\mathcal{P}}(a, \mathcal{I})$$

Proof Let r be a rule and I a total interpretation such that $a \in H(r)$, $(H(r) \setminus \{a\}) \cap I = \emptyset \wedge B^+(r) \subseteq I \wedge B^-(r) \cap I = \emptyset$ (i.e., $r \in supp_{\mathcal{P}}(a, I)$). Then, for any $\mathcal{I}$ such that $I \in total_{\mathcal{P}}(\mathcal{I})$, it holds that $\nexists p \in H(r) \setminus \{a\} : \mathcal{I}(p) \geq T$, $\nexists p \in B^+(r) : \mathcal{I}(p) = F$, $\nexists p \in B^-(r) : \mathcal{I}(p) = T$ (i.e., $r \in psupp_{\mathcal{P}}(a, \mathcal{I})$). As a consequence, $\forall I \in total_{\mathcal{P}}(\mathcal{I}) : supp_{\mathcal{P}}(a, I) \subseteq psupp_{\mathcal{P}}(a, \mathcal{I})$ holds. □

Remark Given a ground program $\mathcal{P}$, a ground atom a and a partial interpretation $\mathcal{I}$,

$$psupp_{\mathcal{P}}(a, \mathcal{I}) = \bigcup_{I \in total_{\mathcal{P}}(\mathcal{I})} supp_{\mathcal{P}}(a, I)$$

holds if $\forall r \in \mathcal{P} : H(r) \cap B^+(r) = \emptyset \wedge B^+(r) \cap B^-(r) = \emptyset$. We note that rules which do not satisfy this property are eliminated in a pre-processing step in `DLV`.

5.2.2 Operators

In the sequel, we will consider a special kind of partial interpretation, the inconsistent interpretation $\Box$. It can be thought of as representing inconsistency or the fact that no answer set is an extension of it. Even more, we set $ext(\Box) = \emptyset$ and $total_{\mathcal{P}}(\Box) = \emptyset$.

For the definition of operators, we define a merging function $\uplus$ on partial interpretations. This function merges the two truth values of each atom in the two interpretations according to the knowledge ordering $\preceq$. If for some atom these truth values are comparable with respect to $\preceq$, then it is mapped to the greater one in the result, unless some other atom causes inconsistency, which happens if the two truth values are incomparable.

Definition 5.2.5
Given two partial interpretations $\mathcal{I}$ and $\mathcal{J}$ defined over a common $B_{\mathcal{P}}$, let

$$\mathcal{I} \uplus \mathcal{J} = \begin{cases} \Box, & \text{if } \exists a \in B_{\mathcal{P}} : \mathcal{I}(a) \not\preceq \mathcal{J}(a) \wedge \mathcal{J}(a) \not\preceq \mathcal{I}(a) \\ \{a \rightarrow \max_{\preceq}\{\mathcal{I}(a), \mathcal{J}(a)\} \mid a \in B_{\mathcal{P}}, \mathcal{I}(a) \neq U \vee \mathcal{J}(a) \neq U\}, & \text{else.} \end{cases}$$

Note that this merging operator preserves atoms which are true or false in any of the two interpretations, it preserves must-be-true atoms unless they are true in the other interpretation, and it yields the inconsistent interpretation if some atom is true or must-be-true in one interpretation, but false in the other.

Let us now review the generalization of two well-known operators to our framework: The immediate consequence operator $T_{\mathcal{P}}$ (a variant of the operator defined in [120]) and a backward-chaining operator $F_{\mathcal{P}}$ (similar to an operator given in [66], and to concepts from [26]). The former derives the

truth of an atom occurring in the head of a rule the body of which is true and in which all other head atoms are false with respect to the given partial interpretation. The latter operator establishes the falsity of a NAF literal occurring in the body of a rule the head of which is false and in which all other body literals are true with respect to the given partial interpretation, by deriving the atom in the NAF literal as false if it is a positive literal or by deriving the atom in the NAF literal as must-be-true if it is a negative literal. Note that $T_{\mathcal{P}}$ can derive truth of an atom as this atom is guaranteed to have a supporting rule, while $F_{\mathcal{P}}$ can derive an atom only to be must-be-true, as the existence of a supporting rule can not be anticipated.

Definition 5.2.6
Given a ground program $\mathcal{P}$, and a partial interpretation $\mathcal{I}$, we define

$$T_{\mathcal{P}}(\mathcal{I}) = \{a \rightarrow T \mid \exists r \in \mathcal{P} : a \in H(r), val_{\mathcal{I}}^{H}(H(r) \setminus \{a\}) = F, val_{\mathcal{I}}^{B}(\mathcal{B}(r)) = T\}$$

$$\begin{aligned} F_{\mathcal{P}}(\mathcal{I}) = \{a \rightarrow F \mid \exists r \in \mathcal{P} : a \in B^{+}(r), & val_{\mathcal{I}}^{H}(H(r)) = F, \\ & val_{\mathcal{I}}^{B}(\mathcal{B}(r) \setminus \{a\}) = T\} \\ \uplus \{a \rightarrow M \mid \exists r \in \mathcal{P} : a \in B^{-}(r), & val_{\mathcal{I}}^{H}(H(r)) = F, \\ & val_{\mathcal{I}}^{B}(\mathcal{B}(r) \setminus \{\texttt{not } a\}) = T\} \end{aligned}$$

Furthermore, let $T_{\mathcal{P}}(\square) = F_{\mathcal{P}}(\square) = \square$.

Let us now complement these basic operators by derivations which are motivated by must-be-true atoms and supporting rules. We introduce three auxiliary operators $S_{1,\mathcal{P}}$, $S_{2,\mathcal{P}}$, and $S_{3,\mathcal{P}}$. $S_{1,\mathcal{P}}$ forces the last potentially supporting rule for a must-be-true atom b to be the definitely supporting rule for b. To this end, derivations are performed which guarantee that b is the only true head atom, and that the body evaluates to true. $S_{2,\mathcal{P}}$ derives atoms without potentially supporting rules as false. Finally, $S_{3,\mathcal{P}}$ is an adaptation of $T_{\mathcal{P}}$ to consider must-be-true atoms.

Definition 5.2.7
Given a ground program $\mathcal{P}$ and a partial interpretation $\mathcal{I}$, we define

$$\begin{aligned} S_{1,\mathcal{P}}(\mathcal{I}) = \{a \rightarrow F \mid \exists b \in \mathcal{I}^{M} \cup \mathcal{I}^{T} : & psupp_{\mathcal{P}}(b, \mathcal{I}) = \{r\}, \\ & a \in H(r) \setminus \{b\} \vee a \in B^{-}(r)\} \\ \uplus \{a \rightarrow M \mid \exists b \in \mathcal{I}^{M} \cup \mathcal{I}^{T} : & psupp_{\mathcal{P}}(b, \mathcal{I}) = \{r\}, a \in B^{+}(r)\} \end{aligned}$$

$$S_{2,\mathcal{P}}(\mathcal{I}) = \{a \rightarrow F \mid psupp_{\mathcal{P}}(a, \mathcal{I}) = \emptyset\}$$

$$S_{3,\mathcal{P}}(\mathcal{I}) = \{a \rightarrow M \mid \exists r \in \mathcal{P} : a \in H(r), val_{\mathcal{I}}^{H}(H(r) \setminus \{a\}) = F, val_{\mathcal{I}}^{B}(\mathcal{B}(r)) = M\}$$

We now define a combined operator $S_{\mathcal{P}}$ by merging the traditional and new operators.

Definition 5.2.8
Given a ground program $\mathcal{P}$ and a partial interpretation I, we define

$$S_{\mathcal{P}}(\mathcal{I}) = T_{\mathcal{P}}(\mathcal{I}) \uplus F_{\mathcal{P}}(\mathcal{I}) \uplus S_{1,\mathcal{P}}(\mathcal{I}) \uplus S_{2,\mathcal{P}}(\mathcal{I}) \uplus S_{3,\mathcal{P}}(\mathcal{I})$$

Given an operator Op, let $\overline{Op}$ denote its inflationary variant, i.e. $\overline{Op}(\mathcal{I}) = \mathcal{I} \uplus Op(\mathcal{I})$. It is relatively straightforward to see that the operator $\overline{S_{\mathcal{P}}}$ is monotonic with respect to the partial order induced by $\preceq$ on partial interpretations (where $\square$ is defined as an artificial supremum connected to all partial interpretations), since only additional knowledge or $\square$ can be produced by $\overline{S_{\mathcal{P}}}$, and no knowledge in $\mathcal{I}$ is ever retracted by $\overline{S_{\mathcal{P}}}$.

Since $\overline{S_{\mathcal{P}}}$ is monotonic in the partial order described above, and since this partial order forms a complete lattice over the partial interpretations, by virtue of the famous Knaster-Tarski theorem a least fixpoint for $\overline{S_{\mathcal{P}}}$ exists, which we will refer to as $\Delta_{\mathcal{P}}$.

Definition 5.2.9
Given a ground program $\mathcal{P}$ and a partial interpretation $\mathcal{I}$, define the n-fold application of $\overline{S_{\mathcal{P}}}$ (i.e., its powers) as follows:

$$\begin{aligned} &\overline{S_{\mathcal{P}}}^{(0)}(\mathcal{I}) = \mathcal{I}, \\ &\overline{S_{\mathcal{P}}}^{(n)}(\mathcal{I}) = \overline{S_{\mathcal{P}}}(\overline{S_{\mathcal{P}}}^{(n-1)}(\mathcal{I})) \quad \text{for } n > 0. \end{aligned}$$

Since the number of partial interpretations is finite, for any partial interpretation $\mathcal{I}$ a nonnegative integer j exists such that $\overline{S_{\mathcal{P}}}^{(j)}(\mathcal{I}) = \overline{S_{\mathcal{P}}}^{(j+1)}(\mathcal{I})$. Let $k_{\mathcal{P}}^{\mathcal{I}}$ denote the least such integer j.

Definition 5.2.10
We define (for a partial interpretation $\mathcal{I}$ and a ground program $\mathcal{P}$) the fixpoint iteration operator of $\overline{S_{\mathcal{P}}}$ as

$$\Delta_{\mathcal{P}}(\mathcal{I}) = \overline{S_{\mathcal{P}}}^{(k_{\mathcal{P}}^{\mathcal{I}})}(\mathcal{I}).$$

We next state a requirement to be fulfilled by operators which are suitable for deterministic consequence computation, and subsequently we will show that our operators indeed satisfy this requirement.

Operators for deterministic consequence computation should be answer set preserving. That is, all answer sets which are extensions of the initial partial interpretation should also be among the extensions of the interpretation yielded by the operator.

Definition 5.2.11
Let, for a ground program $\mathcal{P}$ and a partial interpretation $\mathcal{I}$, $\mathcal{AS}(\mathcal{I}, \mathcal{P}) = total_{\mathcal{P}}(\mathcal{I}) \cap \mathcal{AS}(\mathcal{P})$ denote the answer sets which are extensions of $\mathcal{I}$. An operator Op is answer set preserving *if $\mathcal{AS}(\mathcal{I}, \mathcal{P}) = \mathcal{AS}(Op(\mathcal{I}), \mathcal{P})$ is guaranteed for arbitrary $\mathcal{I}$ and $\mathcal{P}$.*

We will next state a property on totalizations (and thus extending answer sets) in connection with the merging function $\uplus$, which we will use in a subsequent proof.

Lemma 5.2.1
For any partial interpretations $\mathcal{I}$ and $\mathcal{J}$ of a ground program $\mathcal{P}$, $total_{\mathcal{P}}(\mathcal{I} \uplus \mathcal{J}) = total_{\mathcal{P}}(\mathcal{I}) \cap total_{\mathcal{P}}(\mathcal{J})$ holds.

Proof If either $\mathcal{I}$ or $\mathcal{J}$ is $\square$, the result trivially holds, as in this case $\mathcal{I} \uplus \mathcal{J} = \square$. Otherwise, if $\mathcal{I} \uplus \mathcal{J} = \square$ but neither $\mathcal{I} = \square$ nor $\mathcal{J} = \square$, then an atom a exists, such that, without loss of generality, $\mathcal{I}(a) \in \{T, M\}$ and $\mathcal{J}(a) = F$. Then, $a \in A$ holds for each $A \in total_{\mathcal{P}}(\mathcal{I})$ and $a \notin A$ holds for each $A \in total_{\mathcal{P}}(\mathcal{J})$, and therefore $total_{\mathcal{P}}(\mathcal{I}) \cap total_{\mathcal{P}}(\mathcal{J}) = \emptyset = total_{\mathcal{P}}(\mathcal{I} \uplus \mathcal{J})\mathcal{P} = total_{\mathcal{P}}(\square)$ holds.

If $\mathcal{I} \uplus \mathcal{J} \neq \square$, then we will show (i) if $I \in total_{\mathcal{P}}(\mathcal{I}) \wedge I \in total_{\mathcal{P}}(\mathcal{J})$ then $I \in total_{\mathcal{P}}(\mathcal{I} \uplus \mathcal{J})$, and (ii) if $I \in total_{\mathcal{P}}(\mathcal{I} \uplus \mathcal{J})$ then $I \in total_{\mathcal{P}}(\mathcal{I}) \wedge I \in total_{\mathcal{P}}(\mathcal{J})$.

To see (i), note that for each $a \in I$, $\mathcal{I}(a) \in \{T, M, U\} \wedge \mathcal{J}(a) \in \{T, M, U\}$ holds, which implies $(\mathcal{I} \uplus \mathcal{J})(a) \in \{T, M, U\}$. In a similar way, for each $a \in B_{\mathcal{P}} \setminus I$, $\mathcal{I}(a) \in \{F, U\} \wedge \mathcal{J}(a) \in \{F, U\}$ holds, which implies $(\mathcal{I} \uplus \mathcal{J})(a) \in \{F, U\}$. Because in $\uplus$, each atom is treated independently, it follows that $I \in total_{\mathcal{P}}(\mathcal{I} \uplus \mathcal{J})$.

To see (ii), we note that, symmetrically to (i), for each $a \in I$, $(\mathcal{I} \uplus \mathcal{J})(a) \in \{T, M, U\}$ holds, which implies $\mathcal{I}(a) \in \{T, M, U\} \wedge \mathcal{J}(a) \in \{T, M, U\}$. Also, for each $a \in B_{\mathcal{P}} \setminus I$, $(\mathcal{I} \uplus \mathcal{J})(a) \in \{F, U\}$ holds, which implies $\mathcal{I}(a) \in \{F, U\} \wedge \mathcal{J}(a) \in \{F, U\}$. Again, by independency of atoms in $\uplus$, we conclude that $I \in total_{\mathcal{P}}(\mathcal{I})$ and $I \in total_{\mathcal{P}}(\mathcal{J})$ hold, which concludes the proof. $\square$

As the set of answer sets extending a partial interpretation is a subset of its totalizations, we obtain the following result as a direct consequence of Lemma 5.2.1.

Corollary 5.2.1
Given partial interpretations $\mathcal{I}$ and $\mathcal{J}$ and a ground program $\mathcal{P}$, $\mathcal{AS}(\mathcal{I} \uplus \mathcal{J}, \mathcal{P}) = \mathcal{AS}(\mathcal{I}, \mathcal{P}) \cap \mathcal{AS}(\mathcal{J}, \mathcal{P})$ holds.

We are now ready to state our main result concerning the suitability of the operator $\overline{S_{\mathcal{P}}}$ for computing deterministic consequences.

Theorem 5.2.1
The operator $\overline{S_{\mathcal{P}}}$ is answer set preserving.

Proof (Sketch) We first note that any inflationary operator $\overline{Op}$ is answer set preserving iff Op is sound with respect to answer sets, i.e. if it derives atoms as true or must-be-true only if the atom is in all answer sets extending the given partial interpretation, if it derives atoms as false only if the atom

is not in any answer set extending the given partial interpretation, and if it yields $\square$ only if no answer set extends the given partial interpretation. This follows from Corollary 5.2.1.

We will argue for each of the operators constituting $\overline{S_{\mathcal{P}}}$ that it is sound with respect to answer sets. Then, by Corollary 5.2.1, $\overline{S_{\mathcal{P}}}$ is answer set preserving. Let $\mathcal{I}$ denote an arbitrary consistent partial interpretation and $\mathcal{P}$ an arbitrary ground program.

$T_{\mathcal{P}}$: If an atom a occurs in the head of a rule r such that all other head atoms are false and all body literals are true w.r.t. $\mathcal{I}$, then $a \in A$ for any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, otherwise r would not be satisfied in A.

$F_{\mathcal{P}}$: Similar to $T_{\mathcal{P}}$, if an atom a occurs in the positive body of a rule r such that all other body literals are true and all head atoms are false w.r.t. $\mathcal{I}$, then $a \notin A$ for any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, otherwise r would not be satisfied in A. Symmetrically, if an atom a occurs in the negative body of a rule r such that all other body literals are true and all head atoms are false w.r.t. $\mathcal{I}$, then $a \in A$ for any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, otherwise r would not be satisfied in A.

$S_{1,\mathcal{P}}$: By Proposition 5.2.2, each atom in an answer set must have at least one supporting rule, and by Proposition 5.2.3 the number of potentially supporting rules is always greater or equal than the number of supporting rules. Together, if a must-be-true or true atom b has only one potentially supporting rule with respect to $\mathcal{I}$, then exactly this rule must be the supporting rule for b in any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$. So all head atoms different from b will not be contained in any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, all positive body atoms must occur in all $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, and all atoms in the negative body will not be in any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$.

$S_{2,\mathcal{P}}$: Again by Proposition 5.2.2 and Proposition 5.2.3, if any atom a has no potentially supporting rule with respect to $\mathcal{I}$, then it cannot have any supporting rule in any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, and consequently $a \notin A$ for all $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$.

$S_{3,\mathcal{P}}$: Similar to $T_{\mathcal{P}}$, if an atom a occurs in the head of a rule r such that all other head atoms are false and all body literals are must-be-true or true w.r.t. $\mathcal{I}$, then $a \in A$ for any $A \in \mathcal{AS}(\mathcal{I}, \mathcal{P})$, otherwise r would not be satisfied in A.

$\square$

Since $\Delta_{\mathcal{P}}$ is just an iterated application of $\overline{S_{\mathcal{P}}}$, we can immediately conclude the following result from Theorem 5.2.1.

Corollary 5.2.2
The operator $\Delta_{\mathcal{P}}$ is answer set preserving.

We have defined the operator $\Delta_{\mathcal{P}}$ and shown its suitability for computing deterministic consequences in DLV.

5.2.3 Implementation

We will now give an algorithm that computes $\Delta_{\mathcal{P}}$. The resulting procedure *DetCons* is shown in Figure 5.4. Given a partial interpretation $\mathcal{I}$ for $\mathcal{P}$, it extends $\mathcal{I}$ by what we call the *deterministic consequences* of $\mathcal{I}$ w.r.t. $\mathcal{P}$. It can assign F, M or T to any undefined atom, but can only assign T to *mbt* atoms. *DetCons* also detects inconsistencies, e.g. if some *mbt* atom should get the value F.

As long as *DetCons* has modified the interpretation $\mathcal{I}$ for the ground program $\mathcal{P}$, the Boolean variable $modified$ is true at the end of the **repeat** loop of Step 2. If any inconsistency is detected, the procedure immediately aborts by means of an **exit** instruction.

Steps 4–12 focus on rules which are not satisfied (as defined in Definition 5.1.3). If the head of the rule is *false* and its body is either *true* or *mbt*, then the procedure exits returning $contradiction = true$, because there is no way to satisfy r (recall that *true* and $false$ atoms cannot be changed and *mbt* can evolve only into *true*). Steps 7–12 enforce the satisfaction of a rule $r \in \mathcal{P}$ if this can be done deterministically, i.e., by changing the value of exactly one literal occurring in r. Consider Steps 7–8: If the truth value of $B(r)$ according to $\mathcal{I}$ is X, where X is at least M, i.e., *mbt* or *true*, and every atom , except for one atom p, in the head of r is $false$, we can draw a deterministic consequence. We enforce the satisfaction of r by incrementing the truth value of p up to the value of $B(r)$. For instance, if p is either undefined or *mbt* w.r.t. $\mathcal{I}$ and $val_{\mathcal{I}}^{B}(B(r)) = T$, then $\mathcal{I}$ is modified by assigning the value T to p, denoted by $\mathcal{I}(p) := T$ in the algorithm. Note that this is the only step which can assign the value *true* to an atom, that is, *DetCons* assigns the value *true* to an atom p only if p is supported.

Now, consider Steps 9–12: If the head of r is $false$, but its body is at least *mbt*, except for one undefined literal L, then L should get the truth value $false$ in order to satisfy r. Note that, if L is a negative literal `not` p, this is accomplished by setting $\mathcal{I}(p) := M$. Indeed, declaring p *true* would not guarantee the supportedness of this atom.

Steps 13–23 draw deterministic conclusions following the supportedness principle. In particular, if a *true* or *mbt* atom has no potentially supporting rule according to the interpretation $\mathcal{I}$, then we get a contradiction (Step 14), while an undefined atom without any potentially supporting rule can be declared $false$ (Steps 15–16).

If a *true* or *mbt* atom p has only one potentially supporting rule r, i.e., $psupp_{\mathcal{P}}(p, \mathcal{I}) = \{r\}$, then r must be able to derive the truth of p. Thus, we enforce that p is derivable from r assigning suitable truth values to every undefined literal occurring in r (see Steps 18–23). This is a sort of backward

```
Procedure DetCons(P: Program; var I: Interpretation; var contradiction: Boolean)
(* Computes the deterministic consequences for P w.r.t. I *)
var modified: Boolean;
begin
(1)     contradiction := false;
(2)     repeat
(3)         modified := false;
            (* Enforce satisfaction of all rules *)
(4)         for each rule r ∈ Ground(P) not satisfied w.r.t. I do
(5)             if val_I^B(B(r)) ≥ M and val_I^H(H(r)) = F
(6)                 contradiction := true; exit procedure;
(7)             else if val_I^B(B(r)) ≥ M
                        and val_I^H(H(r) \ {p}) = F for some p ∈ H(r) then
(8)                 I(p) := val_I^B(B(r)); modified := true;
(9)             if val_I^H(H(r)) = F and val_I^B(B(r) \ {L}) ≥ M
                for some undefined literal L ∈ B(r) then
(10)                modified := true;
(11)                if L ∈ B^+(r) then I(p) := F;
(12)                else (* L is a negative literal not p *) I(p) := M;
        end for;
        (* Ensure supportedness *)
(13)    if |psupp_P(p, I)| = 0 and I(p) ≥ M for some atom p then
(14)        contradiction := true; exit procedure;
(15)    for each atom p s.t. I(p) = U and |psupp_P(p, I)| = 0 do
(16)        I(p) := F;
(17)    for each atom p s.t. I(p) ≥ M and |psupp_P(p, I)| = 1 do
          Let r be the (unique) rule in psupp_P(p, I);
(18)        for each undefined atom q ∈ (H(r) \ {p}) do
(19)            I(q) := F; modified := true;
(20)        for each undefined positive literal q ∈ B(r) do
(21)            I(q) := M; modified := true;
(22)        for each undefined negative literal not q ∈ B(r) do
(23)            I(q) := F; modified := true;
          end for;
(24)    until not modified
end_procedure
```

Figure 5.4: Function for computing the deterministic consequences

propagation step: From the truth of the head we derive that all body literals must be true.

In total, the derivations in the main repeat loop correspond to the derivations by $S_{\mathcal{P}}$. The difference is that in *DetCons*, derivations are performed sequentially, while they are done in parallel in $S_{\mathcal{P}}$. But the iterations ensure that the fixpoints will eventually be equal, and that *DetCons* effectively computes $\Delta_{\mathcal{P}}$. We state this result without proof.

Theorem 5.2.2
The procedure DetCons$(\mathcal{P}; \mathcal{I}; bool)$ *computes* $\Delta_{\mathcal{P}}(\mathcal{I})$. *More precisely, given*

(i) $\texttt{reached(X)} \leftarrow \texttt{start(X)}.$
(ii) $\texttt{reached(X)} \leftarrow \texttt{reached(Y)}, \texttt{inPath(Y, X)}.$
(iii) $\texttt{inPath(X, Y)} \vee \texttt{outPath(X, Y)} \leftarrow \texttt{arc(X, Y)}.$
(iv) $\leftarrow \texttt{inPath(X, Y)}, \texttt{inPath(X, Y1)}, \texttt{Y} <> \texttt{Y1}.$
$\leftarrow \texttt{inPath(X, Y)}, \texttt{inPath(X1, Y)}, \texttt{X} <> \texttt{X1}.$
(v) $\leftarrow \texttt{node(X)}, \texttt{not reached(X)}.$

Figure 5.5: The Hamiltonian path program $\mathcal{P}_{hp}$ from Chapter 3

a program $\mathcal{P}$ and a partial interpretation $\mathcal{I}$, if $\Delta_{\mathcal{P}}(\mathcal{I}) \neq \square$, then $\mathcal{I}$ is equal to $\Delta_{\mathcal{P}}(\mathcal{I})$ after the procedure execution and bool is false; if $\Delta_{\mathcal{P}}(\mathcal{I}) = \square$, then bool is true.

Example 5.2.2
We have reproduced the program for HAMPATH given in Chapter 3 in Figure 5.5. Now consider this program $\mathcal{P}_{hp}$ together with the graph of Figure 5.6 on the following page, encoded as shown in Figure 5.7 on the next page, starting with the interpretation $\mathcal{I} = \emptyset$, where $\mathcal{I}^T = \mathcal{I}^M = \mathcal{I}^F = \emptyset$ and $\mathcal{I}^U = B_{\mathcal{P}}$.

First, all atoms in facts are derived as true (by lines 4–8), and all atoms not occurring in any head are derived as false (by lines 15–16). Subsequently, by rule (i), $\texttt{reached(a)}$ *is immediately derived (by lines 4–8). The constraint (v) essentially serves as a query that assures that all nodes are indeed reached. As* $\texttt{node(X)}$ *is true for all nodes,* $\texttt{reached(X)}$ *is derived as* mbt *by means of lines 9–12, for each $X \in \{a, b, c, d, e\}$.*

The mbt *atom* $\texttt{reached(b)}$ *is only derivable by a single ground instance*[2] *of rule (ii), namely* $\texttt{reached(b)} \leftarrow \texttt{reached(a)}, \texttt{inPath(a, b)}$. *At this point, the backward propagation step described above comes into play and sets* $\texttt{inPath(a, b)}$ *to* mbt *(lines 17–21). Lines 17–23 implement back-propagation, where from the truth values of the head we can infer some knowledge on the truth values of the body in the case where only a single potentially supporting rule remains for some* mbt *literal. Then, $psupp_{\mathcal{P}}(\texttt{outPath(a, b)}, \mathcal{I})$ becomes empty, since the only rule with* $\texttt{outPath(a, b)}$ *in the head contains also the* mbt $\texttt{inPath(a, b)}$. *Thus,* $\texttt{outPath(a, b)}$ *is derived as* false *(lines 15–16). In turn, this causes lines 4–8 to derive* $\texttt{inPath(a, b)}$ *as* true. *Now we easily derive* $\texttt{reached(b)}$ *as true from (ii).*

Also $\texttt{inPath(c, d)}$ *and* $\texttt{outPath(c, d)}$ *are derived as* true *and* false, *respectively, in analogy to* $\texttt{inPath(a, b)}$ *and* $\texttt{outPath(a, b)}$.

Each node in a Hamiltonian path has exactly one outgoing arc, and indeed lines 9–12 derive $\texttt{inPath(a, c)}$ *and* $\texttt{inPath(a, e)}$ *as* false, *which then leads to* $\texttt{outPath(a, c)}$ *and* $\texttt{outPath(a, e)}$ *to be set to* true.

[2]Note that the instantiation procedure of DLV generates only ground rules that are constructible from the facts in the input ([59]).

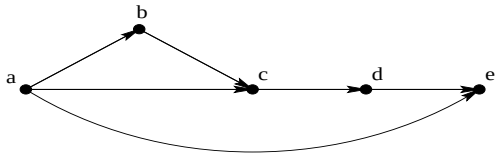

Figure 5.6: Example graph 1 for Hamiltonian path.

```
start(a).
node(a).  node(b).  node(c).  node(d).  node(e).
arc(a,b).  arc(a,c).  arc(a,e).  arc(b,c).  arc(c,d).  arc(d,e).
```

Figure 5.7: The encoding of the graph depicted in Figure 5.6.

Now we derive `inPath(b,c)` *and* `inPath(d,e)` *as* true *in the same way we derived* `inPath(a,b)` *above, and eventually are able to obtain* `reached(c)`, `reached(d)`, *and* `reached(e)`. *That is, starting from an "empty" interpretation, a single invocation of* DetCons *has deterministically and efficiently found the unique Hamiltonian path for this graph.*

It is worthwhile noting that *DetCons* has been implemented very carefully in the `DLV` system. By using sophisticated data structures and techniques similar to those described in [41, 96] for representing rules and interpretations, it runs in linear time, i.e., in time $O(\| \mathcal{P} \| + \| I \|)$, where $\| \cdot \|$ denotes the size of an object. Basically, this is possible by keeping counters of the number of undefined atoms and must-be-true atoms in the head, the positive, and the negative body instead of computing the truth value of head and body by iteration.

5.3 Choice

In this section we focus on the question how to select *PT literals* in line (6) of *ComputeAnswerSets* in Figure 5.3, such that the likelihood of finding an answer set is maximized.

5.3.1 Basic Method

To this end, we employ so-called "look-ahead", that is, we temporarily assume the truth of one PT literal[3] at a time and perform the deterministic derivations, i.e. we apply the *DetCons* function (and hence the operator $\Delta_{\mathcal{P}}$). On the basis of the changes which have been derived during this look-ahead, we then make the decision which PT literal should be taken. Note that the *Smodels* system [99, 117] and many SAT checkers also employ

[3]For a negative literal this means assigning false to its atom.

look-ahead, but they employ different heuristic criteria, as discussed below in Section 5.3.2.

Definition 5.3.1
Let $\mathcal{I}$ be a partial interpretation and $\mathcal{P}$ a ground program, furthermore let us define for an atom p, for which $\mathcal{I}(p) = M$ holds, its *level* n, if $|psupp_{\mathcal{P}}(p, \mathcal{I})| = n$.

During the computation, we maintain the following counters:

$mbt^-(p)$ The number of eliminated must-be-true atoms, i.e. the number of atoms the truth value of which changed from M to T during the computation of *DetCons*.

$mbt^+(p)$ The number of inserted must-be-true atoms, i.e. the number of atoms the truth value of which changed from U to M during the computation of *DetCons*.

$mbt_2^-(p)$ The number of eliminated must-be-true atoms of level 2, i.e. the number of atoms the truth value of which changed from M (with two potentially supporting rules) to T during the computation of *DetCons*.

$mbt_2^+(p)$ The number of inserted must-be-true atoms of level 2, i.e. the number of atoms the truth value of which changed from U or M (with more than two potentially supporting rules) to M (with two potentially supporting rules) during the computation of *DetCons*.

$mbt_3^-(p)$ The number of eliminated must-be-true atoms of level 3, i.e. the number of atoms the truth value of which changed from M (with three potentially supporting rules) to T or M (with fewer than three potentially supporting rules) during the computation of *DetCons*.

$mbt_3^+(p)$ The number of inserted *mbt* atoms of level 3, i.e. the number of atoms the truth value of which changed from U or M (with more than three potentially supporting rules) to M (with two potentially supporting rules) during the computation of *DetCons*.

$Sat(p)$ The number of rules which become satisfied during the computation of *DetCons*.

In addition, we define the following differential counters:

$$\Delta_{mbt}(p) = mbt^-(p) - mbt^+(p)$$

$$\Delta_{mbt2}(p) = mbt_2^-(p) - mbt_2^+(p)$$

$$\Delta_{mbt3}(p) = mbt_3^-(p) - mbt_3^+(p)$$

Using these counter values, we define a heuristic relation over the set of PT literals with respect to $\mathcal{I}$ as follows:

Definition 5.3.2
Given two PT literals a and b, we define a lexicographic ordering relation $<$ as follows:

If $(mbt^-(a) = 0 \wedge mbt^-(b) > 0) \vee \ (mbt^-(a) > 0 \wedge mbt^-(b) = 0)$ then
$a < b \Leftrightarrow mbt^-(a) < mbt^-(b)$
otherwise
$a < b$ holds if one of the following conditions applies:

1. $\Delta_{mbt}(a) < \Delta_{mbt}(b)$
2. $\Delta_{mbt2}(a) < \Delta_{mbt2}(b) \wedge \Delta_{mbt}(a) = \Delta_{mbt}(b)$
3. $\Delta_{mbt3}(a) < \Delta_{mbt3}(b) \wedge \Delta_{mbt}(a) = \Delta_{mbt}(b) \wedge \Delta_{mbt2}(a) = \Delta_{mbt2}(b)$
4. $Sat(a) < Sat(b) \wedge \Delta_{mbt}(a) = \Delta_{mbt}(b) \wedge \Delta_{mbt2}(a) = \Delta_{mbt2}(b)$
   $\wedge \Delta_{mbt3}(a) = \Delta_{mbt3}(b)$

In other words, if exactly one of $mbt^-(a)$ and $mbt^-(b)$ is zero, we prefer the PT literal for which mbt^- is non-zero. Otherwise (i.e., both $mbt^-(a)$ and $mbt^-(b)$ are zero or both are non-zero), we prefer the one for which the overall number of *mbt* atoms becomes smaller. If this number is equal, we prefer the one for which the overall number of *mbt* atoms of level 2 becomes smaller. If also this number is equal, we use the number of *mbt* atoms of level 3.

The reasoning behind this relation is that the total numbers of *mbt* atoms can be viewed as constraints which are not yet satisfied but eventually have to be for any answer set. So the fewer *mbt* atoms there are, the smaller is the distance to an answer set. Additionally, *mbt* atoms of level 2 and 3 are the ones which are the "hardest" to become satisfied (observe that *mbt* atoms of level 1 are always derived by *DetCons*).

The purpose of the test whether exactly one of $mbt^-(a)$ or $mbt^-(b)$ is zero is that in this case we want to avoid preferring a PT literal, which only introduces new *mbt* atoms but does not eliminate any, over one which eliminates some but introduces more (the former is like a "null action").

The guessing step in the Model Generator (line (6) in *ComputeAnswer-Sets* in Figure 5.3 on page 32) takes a PT literal which is a maximum w.r.t. $<$.

Example 5.3.1
Consider again the program for computing Hamiltonian paths shown in Figure 5.5 on page 43, now together with the encoding of the graph depicted in Figure 5.8 on the next page, given in Figure 5.9 on the following page.

By the first call to DetCons, *only* `reached(a)` *is set to true, while* `reached(b)`, `reached(c)`, `reached(d)`, *and* `reached(e)` *are assigned* mbt *because of the single literal constraints obtained by* (v) *(see Appendix A).*

The choice rule (iii) is instantiated with the arcs (see Appendix A); these rules supply the PT literals (all of which are positive).

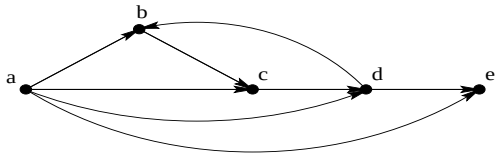

Figure 5.8: Example graph 2 for Hamiltonian path

```
start(a).
node(a).  node(b).  node(c).  node(d).  node(e).
arc(a,b).  arc(a,c).  arc(a,d).  arc(a,e).
arc(b,c).  arc(c,d).  arc(d,b).  arc(d,e).
```

Figure 5.9: The encoding of the graph depicted in Figure 5.8.

Note that the rules which define the predicate `reached` *are instantiated in a way such that* `reached(n)` *occurs in the head of exactly two rules for each node* `n`*, apart from* `a`*. This is because each of these nodes has exactly two incoming arcs.*

Each of the `reached(n)` (`n` $= \{$`b`$, \ldots,$ `e`$\}$) *needs support, but it is not yet known which of the two rules will supply it eventually.*

To evaluate the heuristic relation, we perform a look-ahead: for each PT L*, we assume* L *true, compute its deterministic consequences (by a call to* DetCons*), and store the values of the respective* mbt *counters.*

Let us first consider the PT literal `inPath(a,b)`*: Upon assuming it true, we immediately derive* `reached(b)` *as true, and thus eliminate a* mbt *atom of level 2 (since it occurs in the head of two unsatisfied rules). By statements (9) – (11) in* DetCons *we derive false for* `inPath(a,c)`, `inPath(a,d)`, `inPath(a,e)`, *and* `inPath(d,b)`*, reflecting the fact that no two arcs in the Hamiltonian path may begin in the same node or end in the same node (constraints (iv) in Figure 5.5).*

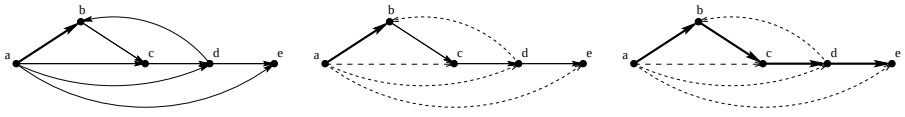

Figure 5.10: Steps during look-ahead for `inPath(a,b)`

After that, for each of `reached(c)`, `reached(d)`, `reached(e)` *only one supporting rule is left, so we can infer that the yet undefined positive body literals of these rules (*`inPath(b,c)`, `inPath(c,d)`, `inPath(d,e)`*) are* mbt*. Moreover, since each of them occurs in the head of exactly one rule and the body of this rule is true, we infer them as true immediately afterwards and even-*

PT literal	mbt^-	mbt^+	Δ_{mbt}	mbt_2^-	mbt_2^+	Δ_{mbt2}	mbt_3^-	mbt_3^+	Δ_{mbt3}
inPath(a, b)	7	3	4	1	0	1	0	0	0
outPath(a, e)	8	4	4	0	0	0	0	0	0
outPath(d, b)	8	4	4	0	0	0	0	0	0
inPath(d, e)	7	3	4	0	0	0	0	0	0
inPath(a, e)	4	3	1	1	0	1	0	0	0
outPath(a, b)	5	4	1	0	0	0	0	0	0
inPath(d, b)	4	3	1	0	0	0	0	0	0
outPath(a, b)	5	4	1	0	0	0	0	0	0
outPath(a, c)	1	1	0	0	0	0	0	0	0
outPath(a, d)	1	1	0	0	0	0	0	0	0
inPath(b, c)	0	0	0	0	0	0	0	0	0
inPath(c, d)	0	0	0	0	0	0	0	0	0

Table 5.1: PT literals and their values, ordered by $<$

tually we also infer reached(c), reached(d), *and* reached(e) *as true. These steps are visualized in Figure 5.10, where bold arcs are in the Hamiltonian path, while dashed arcs are not.*

In total, the deterministic derivation has generated 3 new mbt *atoms (all of which have subsequently been derived as true) and eliminated 7* mbt *atoms, one of which was of level 2.*

All PT literals and their corresponding counter values (we have omitted Sat, as it never used), ordered by $<$, are shown in Table 5.1. Those which are not listed (inPath(a, c), inPath(a, d), outPath(b, c), outPath(c, d)) *generate an inconsistency during propagation. Note that the atoms* reached(n) (n $= \{$b, . . . , e$\}$) *are not PT literals, as they occur only in rules, the body of which is not true yet.*

Thus, following the heuristics, the PT literal inPath(a, b) *is chosen by our computation. Then, the propagation of it, done by* DetCons, *immediately leads to the computation of the Hamiltonian path. Thanks to the heuristics only one choice was sufficient!*

Note that performing look-ahead has an additional merit: If an inconsistency is detected during the propagation of the PT literal, we can then set it to false and apply *DetCons*, thus pruning the search tree quite a bit.

Example 5.3.2
Consider again Example 5.3.1, and let us trace the look-ahead for the PT literal inPath(a, c)*: First,* reached(c) *is set to true, then* inPath(a, b), inPath(a, d), inPath(a, e), *and* inPath(b, c) *are assigned false (via constraints). Because of this,* inPath(c, d), inPath(d, e), *and* inPath(d, b) *become* mbt*. But then the constraint ensuring that exactly one of the outgoing edges of* d *may be in the Hamiltonian path is violated, thus creating an inconsistency (see the visualization in Figure 5.11).*

Therefore we may assume inPath(a, c) *as false and derive its consequences, which include* inPath(b, c) *becoming true, as depicted in Figure*

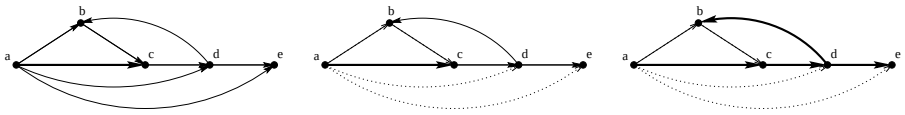

Figure 5.11: Steps during look-ahead for `inPath`(`a`, `c`)

5.12.

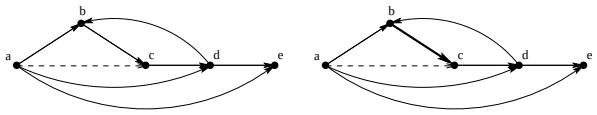

Figure 5.12: Consequences of `inPath`(`a`, `c`) being false

5.3.2 Advanced Methods

Throughout this section, we assume that a ground ASP program $\mathcal{P}$ and a partial interpretation $\mathcal{I}$ have been fixed. Here, we describe the heuristic criteria that will be compared in Section 5.4.1. We consider "dynamic heuristics" (the ASP equivalent of UP heuristics for SAT), that is, branching rules where the heuristic value of a literal L depends on the result of taking L true and computing its consequences. Let $\mathcal{I}_L$ contain the truth assignment for the PT literal L (i.e., $val_{\mathcal{I}}(p) = T$ if $L = p$, or $val_{\mathcal{I}}(p) = F$ if $L = \texttt{not}\ p$), and be equal to $\mathcal{I}$ otherwise. The result of the look-ahead on L is then given by $\mathcal{J}_L = \Delta_{\mathcal{P}}(\mathcal{I}_L)$. We assume that $\mathcal{J}_L \neq \Box$, otherwise L is automatically set to be false (i.e., $val_{\mathcal{I}}(p) = F$ if $L = p$, or $val_{\mathcal{I}}(p) = M$ if $L = \texttt{not}\ p$) and the heuristics is not evaluated on L at all.

Heuristics $\mathbf{h_1}$. This is an extension of the branching rule adopted in the system SATZ [82] – one of the most efficient SAT solvers – to the framework of ASP.

Let the *length* of a rule r (w.r.t. an interpretation $\mathcal{I}$), be the number of undefined literals occurring in r, denoted by $l(r, \mathcal{I}) = |\{a \in H(r) \cup B(r) \mid \mathcal{I}(a) = U\}|$. Let $Unsat_k(Q)$ denote the number of rules r, which are unsatisfied[4] in $\mathcal{J}_Q$, with $l(r, \mathcal{J}_Q) = k$, where $l(r, \mathcal{I}) > k$. In other words, $Unsat_k(Q)$ is the number of unsatisfied rules whose length shrinks to k if the truth of Q is assumed and propagated in the interpretation $\mathcal{I}$. The weight $w_1(Q)$ is

$$w_1(Q) = \Sigma_{k>1}\ Unsat_k(Q) * 5^{-k}$$

[4]Recall that a rule r is satisfied w.r.t. $\mathcal{I}$ if $val^H_{\mathcal{I}}(H(r)) \geq val^B_{\mathcal{I}}(B(r))$.

The factor 5^{-k} has been determined by an empirical analysis (see [82] for details). Thus, the weight function w_1 prefers literals introducing a higher number of short unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more and stronger constraints on the interpretation so that a contradiction can be found earlier [82]. We combine the weight of an atom Q with the weight of its complement `not` Q to favor Q such that $w_1(Q)$ and $w_1(\texttt{not } Q)$ are roughly equal, to avoid that a possible failure leads to a very bad state. To this end, as in SATZ, we define the combined weight $\mathit{comb}\text{-}w_1(Q)$ of an atom Q as follows:

$$\mathit{comb}\text{-}w_1(Q) = w_1(Q) * w_1(\texttt{not } Q) * 1024 + w_1(Q) + w_1(\texttt{not } Q).$$

Again, the factor 1024 has been determined empirically. Given two atoms A and B, the heuristics h_1 prefers B over A ($A <_{h1} B$) iff $\mathit{comb}\text{-}w_1(A) < \mathit{comb}\text{-}w_1(B)$. Once a $<_{h1}$-maximum atom Q is selected, heuristics h_1 takes Q if $w_1(Q) > w_1(\texttt{not } Q)$, `not`.$Q$ else.

Heuristics $\mathbf{h_2}$. The second heuristics we consider is inspired to the branching rule of Smodels, which is a well-known ASP system [118]. Let $\mathit{def}(\mathcal{I}) = |\mathcal{I}^T \cup \mathcal{I}^M \cup \mathcal{I}^F|$ denote the number of atoms which are either true, must-be-true or false in a partial interpretation $\mathcal{I}$. Then, define

$$w_2(Q) = \mathit{def}(\mathcal{J}_Q).$$

Since w_2 maximizes the size of the resulting interpretation, it minimizes the atoms which are left undefined. Intuitively, this minimizes the size of the remaining search space (which is 2^u, where u is the number of undefined atoms in $\mathcal{J}_Q$) [118]. Similar to Smodels, the heuristics h_2 cautiously maximizes the minimum of $w_2(Q)$ and $w_2(\texttt{not } Q)$. More precisely, the preference relationship $<_{h2}$ of h_2 is defined as follows. Given two atoms A and B, $A <_{h2} B$ if $\mathit{min}(w_2(A), w_2(\texttt{not } A)) < \mathit{min}(w_2(B), w_2(\texttt{not } B))$; otherwise, $A <_{h2} B$ if $\mathit{min}(w_2(A), w_2(\texttt{not } A)) = \mathit{min}(w_2(B), w_2(\texttt{not } B))$ and $\mathit{max}(w_2(A), w_2(\texttt{not } A)) < \mathit{max}(w_2(B), w_2(\texttt{not } B))$. Once a $<_{h2}$-maximum PT literal Q is selected, heuristics h_2 takes Q if $w_1(Q) > w_1(\texttt{not } Q)$, `not`.$Q$ else.

Remark. It is worthwhile noting that the heuristics of Smodels, while following the above intuition, is more advanced and sophisticated than h_2. Unfortunately, it is defined for non-disjunctive programs, and centered around properties of unstratified negation. which is not so important in our framework. We do not see any immediate extension of Smodels' heuristics to the framework of disjunctive ASP programs.

Heuristics $\mathbf{h_3}$. Let us consider now the traditional heuristics used in the `DLV` system, which has been defined in Section 5.3.1.

Recall that this heuristics (h_3) considers $\Delta_{mbt}(Q)$, $\Delta_{mbt2}(Q)$, and $\Delta_{mbt3}(p)$ in a prioritized way, to favor atoms for which it is more likely to lead to a supported model. If all *mbt* counters are equal, then the heuristics considers the total number $Sat(Q)$ of rules which are satisfied w.r.t. $\mathcal{J}_Q$.

A $<_{h3}$-maximum atom is selected by the heuristics h_3 of DLV. Unlike the previous heuristics, h_3 considers only atoms (instead of literals), and it does not take into account what happens when the selected atom Q leads to a failure (i.e., $\mathcal{J}_{\mathtt{not}\ Q}$ is not considered in the heuristics).

Heuristics h_4. Finally, we consider a simple "balanced version" h_4 of the heuristics h_3 of DLV, where also the complement of an atom is evaluated for the heuristics. Given a PT literal A, let $\Delta'_\alpha(A) = \Delta_\alpha(A) + \Delta_\alpha(\mathtt{not}.A)$ for $\alpha = \{mbt, mbt, mbt3\}$, and $Sat'(A) = Sat(A) + Sat(\mathtt{not}\ A)$. The heuristics h_4 works precisely as h_3, but considers the primed counters. Once the best atom has been selected, it is taken positive or negative, depending on h_3.

5.3.3 Reducing Look-Ahead Overhead

As described in the previous sections, most heuristics of DLV are "dynamic" heuristics. These heuristics are based on "look-ahead" techniques: to evaluate the heuristic value of a literal L w.r.t. the partial interpretation $\mathcal{I}$ at hand, truth and falsity of L are assumed in the current interpretation, and its consequences are derived by computing its deterministic extensions $\mathcal{J}_L = DetCons(\mathcal{I}_L)$ and $\mathcal{J}'_L = DetCons(\mathcal{I}_{\mathtt{not}.L})$. Note that either of $\mathcal{J}_L$ and $\mathcal{J}'_L$ can be inconsistent, in which case the search space can be pruned early.

The heuristic value of L is a measure of the "quality" of the resulting interpretations $\mathcal{J}_L$ and $\mathcal{J}'_L$. Some of these heuristics proved to be very useful, as they drastically reduce the number of choice-points arising in an ASP computation. However, the computation of these heuristics is very expensive, since the number of literals to be "looked-ahead" is very large in some cases, and the cost of a look-ahead is linear in the size of the Herbrand Base in the worst case. The computation of the heuristics thus often consumes most of the total time taken by an ASP system, and may slow down the ASP system significantly.

Thus, part of the computational gain through the better choices of the branching literals is consumed by the heuristics itself to perform the many look-aheads.

In the sequel, we describe methods which try to reduce the time needed to evaluate the heuristics, by reducing the number of look-aheads that need to be performed. The main contributions are:

A. A new condition which is sufficient to guarantee that, at a given stage of the computation, two literals $\langle A, \mathtt{not}.B\rangle$ have precisely the same set of deterministic consequences w.r.t. the interpretation $\mathcal{I}$ at hand, that

is, $\mathcal{J}_A = \mathcal{J}_{\texttt{not}.B}$). Consequently, A and *not* B are guaranteed to have precisely the same heuristic values, and we avoid the look-ahead for one of them.

This technique allows us to save 50% of the look-aheads in several cases including, e.g., Hamiltonian Path and 3SAT programs.

B. The design of a 2-layered heuristics. A computationally cheap heuristic criterion reduces the set of literals to be considered, and the look-ahead to select the branching literal is applied only to the literals in this set. This method significantly reduces the number of look-aheads, but, unlike the previous technique, it is not an "exact" or "faithful" method, that is, it might exclude literals which would otherwise have had high heuristic values. Also some literals for which the look-ahead detects inconsistency can be missed in this way, so there will be less pruning in general.

C. An implementation of the above techniques in the ASP system DLV, and an evaluation of their efficiency on a number of benchmark problems taken from various domains. The results of the experiments are very positive and both techniques prove to be useful. Moreover, they are orthogonal and their integration performs at least as well as the best individual technique, resulting in a relevant improvement of the performance of the DLV system.

It is worthwhile noting that techniques for reducing the number of look-aheads have been employed in SAT solvers and in other ASP systems. In particular, the ASP system Smodels makes a drastic pruning of the look-aheads by eliminating each literal which has been derived during a previous look-ahead at the same branch-point: For each literal $B \in \mathcal{J}_A$, the look-ahead for B is not performed, because B is guaranteed to be worse than A w.r.t. the heuristic function of Smodels. This technique eliminates a higher number of look-aheads than our technique described in Item A. However, our technique is more general and it is applicable to a wider class of heuristic functions. Indeed, the technique of Smodels relies on a monotonicity property of the heuristics: $\mathcal{J}_B \subset \mathcal{J}_A$ implies that B is worse than A w.r.t. the heuristic function of Smodels. Our technique, instead, is applicable to every criterion determining the heuristic value from the result of the look-ahead (i.e., the heuristic value of A depends only on $\mathcal{J}_A$. In fact, our technique can also be applied in Smodels, while the optimization employed by Smodels cannot be used in DLV, since the heuristics employed in DLV is not monotonic in the sense described above. A 2-layered heuristics similar to the technique of Item B above has been successfully employed in the SAT solver SATZ [82].

Look-Ahead Equivalences

Dynamic heuristics vary only in the interpretation $\mathcal{J}_A$ (resp. $\mathcal{J}_{\texttt{not}\ A}$). It is therefore interesting to identify cases where two literals L and L' are *look-ahead equivalent*, i.e., $\mathcal{J}_L = \mathcal{J}_{L'}$, since one of the two look-ahead computations could be saved. This notion of equivalence is formalized next.

Definition 5.3.3
Let p and q be two undefined literals w.r.t. an interpretation $\mathcal{I}$. p and q are look-ahead equivalent *if $\mathcal{J}_p = \mathcal{J}_q$.*

We can now formulate the following:

Proposition 5.3.1
If two undefined classical literals a and b occur in the head of a rule r in a program $\mathcal{P}$, and a and b are the only undefined literals w.r.t. an interpretation I in r (where we assume that there is no multiple occurrence of classical literals in rules), then it holds that:

1. *If $psupp_{\mathcal{P}}(b, I) = 1$, then a and* `not` *b are look-ahead equivalent.*
2. *If $psupp_{\mathcal{P}}(a, I) = 1$, then* `not` *$a$ and b are look-ahead equivalent.*

Proof (Sketch) Suppose $psupp_{\mathcal{P}}(b, I) = 1$. Then r is the only rule in $\mathcal{P}$ which might derive b. Since the body of r is already true in I, such a derivation is performed iff a becomes false. Therefore, $DetCons(I)$ either contains both a and `not` b or it contains none of them. A symmetric argument shows item 2. □

Example 5.3.3
Consider the program $\mathcal{P}_9 = \{\texttt{a} \vee \texttt{b}.\}$ and $\mathcal{I} = \emptyset$. Both `a` *and* `b` *are PT literals, so look-ahead for* `a`, `not a`, `b`, *and* `not b` *is performed, i.e. we compute $\Delta_{\mathcal{P}}(\{\texttt{a} \rightarrow T\}) = \{\texttt{a} \rightarrow T, \texttt{b} \rightarrow F\}$, $\Delta_{\mathcal{P}}(\{\texttt{a} \rightarrow F\}) = \{\texttt{a} \rightarrow F, \texttt{b} \rightarrow T\}$, $\Delta_{\mathcal{P}}(\{\texttt{b} \rightarrow T\}) = \{\texttt{a} \rightarrow F, \texttt{b} \rightarrow T\}$, and $\Delta_{\mathcal{P}}(\{\texttt{b} \rightarrow F\}) = \{\texttt{a} \rightarrow T, \texttt{b} \rightarrow F\}$. In this example we can save the look-aheads for* `not b` *and* `b` *because of proposition 5.3.1, and thus save half of the look-aheads.*

In `DLV` computations, we can recognize the applicability of Proposition 5.3.1 very efficiently and avoid extraneous look-aheads. Experimental results reported in Section 5.4.1 will show that we avoid up to 50% of look-aheads in some cases (e.g. on 3SAT) by exploiting this simple condition.

2-Layered Heuristics

In [82] a different idea on reducing look-aheads is presented: An easy-to-compute heuristics is defined as a first layer, and look-ahead is only computed on those possible choices which look promising w.r.t. to this easier heuristics. This gives a kind of 2-layered heuristics.

The simple heuristic criteria defined in [82] involve the number of *binary clauses* a classical literal occurs in. The rationale is that this is the number of immediate propagations that can be performed during the look-ahead. This idea can be directly transferred to our ASP framework:

Definition 5.3.4
A binary clause *is a rule which contains exactly two undefined classical literals w.r.t. an interpretation I. The* number of binary occurrences *of an undefined literal a is the number of binary clauses a occurs in.*

Note that this notion directly corresponds to the number of immediate propagations which can be performed by assuming a and `not` a, so it matches the intuition of [82]. To reduce the number of literals to be looked-ahead, we adopt the following criterion:

First-Layer Heuristics $S^{bin}_{\mathcal{P}}(\mathcal{I})$. Let $PT_{\mathcal{P}}(\mathcal{I})$ be the set of PT literals of a ground program $\mathcal{P}$ w.r.t. a partial interpretation $\mathcal{I}$, and let $S^{bin}_{\mathcal{P}}(\mathcal{I}) \subseteq PT_{\mathcal{P}}(\mathcal{I})$ be the set of PT literals having more than the average number of binary occurrences w.r.t. all literals in $PT_{\mathcal{P}}(\mathcal{I})$. Then, consider only the literals in $S^{bin}_{\mathcal{P}}(\mathcal{I})$ for the selection of the branching literals (i.e., make look-ahead only on these literals).

Note that our first-layer heuristics is inspired by the same intuition as the first-layer heuristics in [82], even though it is not precisely the same.

5.4 Experiments

5.4.1 Experimenting with Heuristics

To evaluate the different heuristics presented in Section 5.3, we have chosen a suite of benchmark problems. Among them are three problems which we have already presented in Chapter 3, namely 3SAT (cf. Section 3.3), Hamiltonian Path (cf. Section 3.2), and Strategic Companies (cf. Section 3.4).

In addition, we have performed experiments on Blocksworld Planning, which we briefly describe in the sequel:

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI:

> Given a table and a number of blocks in a (known) initial state and a desired goal state, try to reach that goal state by moving one block at a time such that each block is either on top of another block or the table at any given time step.

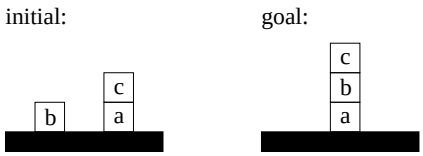

Figure 5.13: Simple Blocksworld Example

Figure 5.13 shows a simple example that can be solved in three time steps: First we move block c to the table, then block b on top of a, and finally c on top of b.

For a complete encoding we refer to [61, 64].

Note that the benchmark problems have been chosen to cover a variety of features in answer set programs: E.g., the HAMPATH includes transitive closure, which is a frequent pattern in logic programs. 3SAT is close to a constraint satisfaction problem, and is an example of a program without recursion. STRATCOMP is a Σ_2^P-complete problem involving non-HCF components. Finally, BW has been included because it employs a kind of temporal reasoning.

Benchmark Data

For 3SAT, we have randomly generated 3CNF formulas over n variables using a tool by Selman and Kautz [115]. For each size we generated 8 such instances, where we kept the ratio between the number of clauses and the number of variables at 4.3, which is near the cross-over point for random 3SAT [31].

The instances for HAMPATH were generated using a tool by Patrik Simons which has been used to compare Smodels against SAT solvers (cf. [118]), and is available at `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`. For each problem size n we generated 8 instances, always assuming node 1 as the starting node.

The blocksworld problems P3 and P4 have been employed in [61] to compare ASP systems, and can be solved in 8 and 9 steps, respectively. We augmented these by problems P5 and P6 which require 11 and 12 steps, respectively. For each of these problems, we have generated 8 random permutations of the input program, which can give indications on the sensitivity of a heuristics to the ordering of rules. Such a sensitivity occurs when the heuristics is not very fine-grained and chooses one among many "best" PT literals by its position in the program.

For STRATCOMP, finally, we randomly generated 8 instances for each problem size n, with n companies and n products. Each company O is controlled by one to three companies, where the actual number of companies is uniform randomly chosen. On average there are 1.5 *cont* relations per

company.

The benchmark data are available at `http://www.dbai.tuwien.ac.at/proj/dlv/`. All experiments were performed on an Athlon/750 FreeBSD 4.2 machine with 256MB of main memory. The binaries were produced with GCC 2.95.2.

Experimental Results

The results of our experiments are displayed in the graphs of Figures 5.14–5.17. In each graph, the horizontal axis reports a parameter representing the size of the instance. On the vertical axis, we report the average running time (measured in seconds) over the 8 instances of the same size we have run (see previous section).

Remark. All heuristics have been implemented in a straightforward way, without optimizations, so the running times reported in the graph are meaningful only for comparing the relative efficiencies of the heuristics.

We have allowed a running time of 600 seconds for each problem instance. In the graphs, the line of an heuristics ends whenever some problem instance was not solved in the maximum time allowed. Table 5.2 displays, for each heuristics, the maximum instance-size where the heuristics could solve *all* problem instances in the maximum allowed time. An entry $> i$ means that even for the largest instance size that was considered in the experiments, all instances were solvable within the time limit.

	h_1	h_2	h_3	h_4
3SAT	270	270	250	260
HAMPATH	80	40	70	> 100
BW	$>$ P6	$>$ P6	P5	$>$ P6
STRATCOMP	> 1200	> 1200	700	> 1200

Table 5.2: Maximal totally solvable instance sizes.

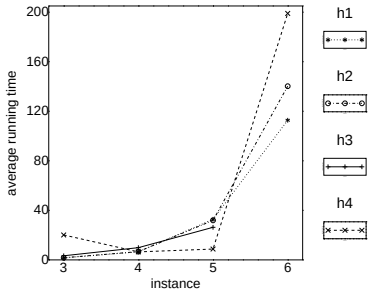

Figure 5.14: Comparison of heuristics: Blocksworld problems, average running times

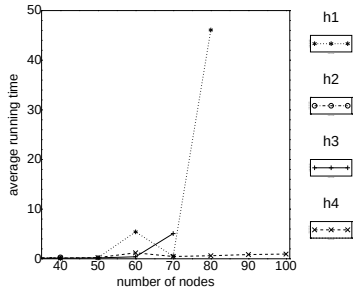

Figure 5.15: Comparison of heuristics: Hamiltonian Path problems, average running times

As expected, heuristics h_3, the "native" heuristics of the DLV system, which does not combine the heuristic values of complementary atoms, is the worst in most cases. It does not terminate on the instance P6 of BW, it could not solve any of the benchmark instances of STRATCOMP (h_3 does not appear at all in Figure Figure 5.16), and could solve fewer problem instances than the others on 3SAT.

Heuristics h_1, the extension of SATZ heuristics to ASP, behaves very well on average. On 3SAT, BW, and STRATCOMP, h_1 could solve all benchmark instances we have run. It is the fastest on BW and one of the two fastest on 3SAT. It shows a negative behavior only on HAMPATH. In this problem, considering must-be-true atoms seems to be crucial for the efficiency.

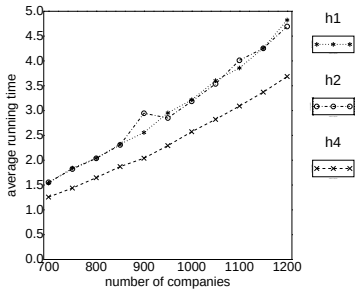

Figure 5.16: Comparison of heuristics: Strategic Companies, average running times

Heuristics h_4 is surprisingly good compared to h_3. It is a simple "balanced version" of heuristics h_3 (the heuristic values of the positive and of the negative literal are combined by sum). This simple extension to h_3 dramatically improves the performance. Indeed, heuristics h_4 solves nearly all instances we ran (only on 3SAT it stopped a bit earlier than other heuristics). It is the best heuristics on STRATCOMP and, importantly, on HAMPATH, where it beats all other heuristics by a relevant factor.

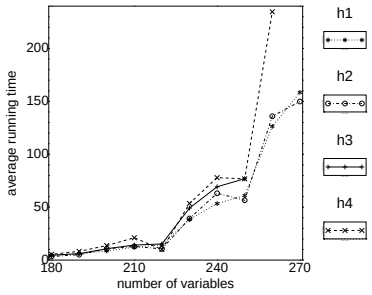

Figure 5.17: Comparison of heuristics: 3SAT problems, average running times

The behavior of heuristics h_2, based on the minimization of the undefined atoms, is rather controversial. It behaves very well on 3SAT and BW, but it is extremely bad on HAMPATH, where it stops at 40 nodes already and is beaten even by the "naive" heuristics h_3. This confirms that further studies are needed to find a proper extension of the heuristics of Smodels to the framework of disjunctive ASP.

Concluding, we observe that both heuristics h_1 and heuristics h_4, significantly improve the efficiency of the native heuristics h_3 of the `DLV` system. The dramatic improvement obtained by the simple change from h_3 to h_4, confirms even more the importance of a careful study of branching rules in ASP systems. This work is only a first step in this field, our future work will include proposing new heuristics for ASP and careful analyses of existing ones, in order to improve the efficiency of ASP systems.

5.4.2 Experimenting with Look-Ahead Reductions

The results of our experiments are displayed in the graphs of Figures 5.18-5.21. For each problem domain we report two graphs: In both graphs the horizontal axis reports a parameter representing the size of the instance, while on the vertical axis we report the running time (measured in seconds) and the number of look-aheads, respectively, averaged over the 8 instances of the same size we have run (see previous section). The curves labeled by "no opt.", "opt. 1", "opt. 2", and "opt. 1+2", denote, respectively, the initial (unoptimized) version, the look-ahead equivalence optimization, the 2-layered optimization, and the combination of both look-ahead equivalence and 2-layered optimization.

Observe first that both optimizations always bring some gain over the original version, as the "no optimization" curve is always on top of the other three curves in all graphs.

The two optimizations have different impact, depending on the problem domain: For Blocksworld, the equivalence optimization performs better than the 2-layered approach, while for Strategic Companies and 3SAT the

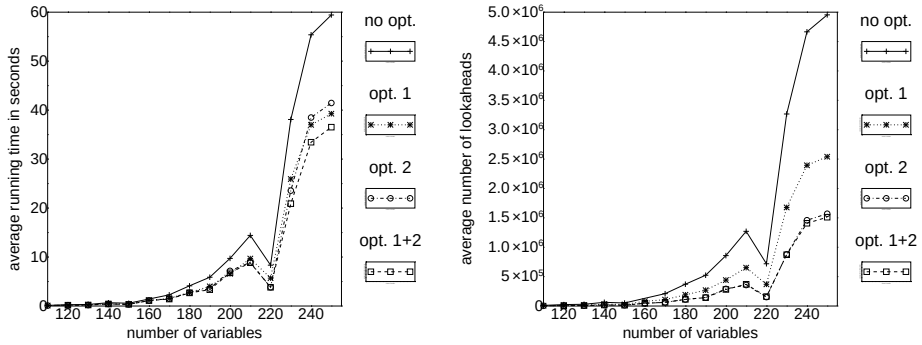

Figure 5.18: Comparison of look-ahead reductions: 3SAT problems, average running times and look-aheads

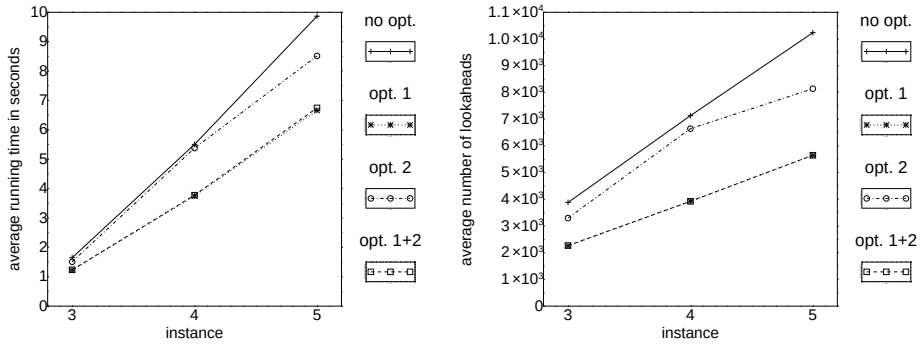

Figure 5.19: Comparison of look-ahead reductions: Blocksworld problems, average running times and look-aheads

opposite holds. For Hamiltonian Path both optimizations behave roughly equal.

The combination of the two optimizations combines the benefits in the sense that performance is always as good as for the better of the two strategies. Indeed, the curve combining the two strategies (opt.1+2) often nearly coincides with the curve of the best of opt.1 and opt.2, e.g. for Blocksworld opt.1+2 and opt.1 are almost equal, while for Strategic Companies opt.1+2 and opt.2 coincide. On Hamiltonian Path opt.1, opt.2, and opt.1+2 all give the same speed-up. Finally, in the case of 3SAT there are even better results for opt.1+2 than for any of the two methods alone.

Note that for opt.2 (and opt.1+2), the runtime and the number of look-aheads need not correlate, as fewer look-aheads are performed but the quality of the PTs may be worse, which may lead to larger trees. For opt.1 the choices remain the same, but only the number of look-aheads can be reduced, so avoided look-aheads directly reduce the runtime in this case.

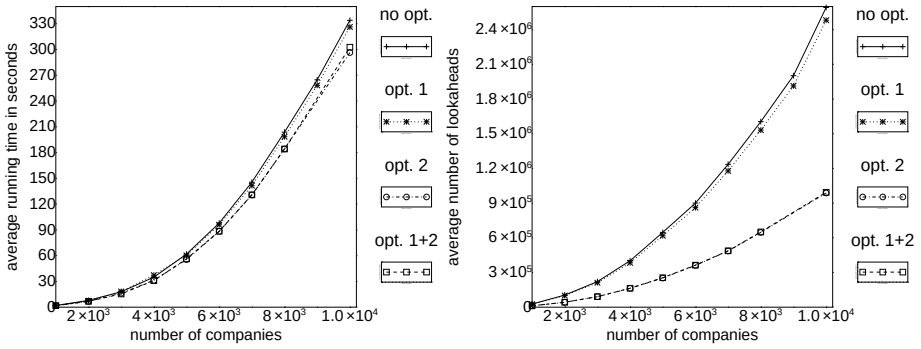

Figure 5.20: Comparison of look-ahead reductions: Strategic Companies, average running times and look-aheads

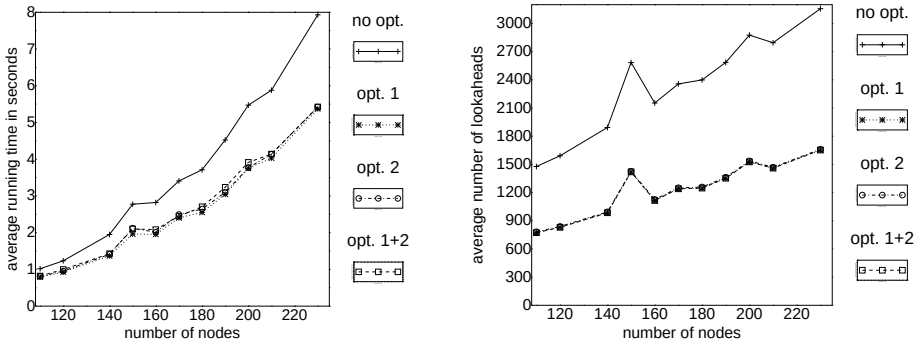

Figure 5.21: Comparison of look-ahead reductions: Hamiltonian Path problems, average running times and look-aheads

Thus, both optimizations turned out to be useful, and we have incorporated their combination in the version of DLV released in June 2001. We believe that this is a promising way towards the improvement of ASP systems that should be subject of further investigation. Indeed, besides optimizing the implementation of the techniques proposed in this work, we have already planned future work to explore other promising ways to reduce the number of look-aheads.

Part II

Extensions and Front-Ends

After having described in the previous part how to build an efficient system for pure Answer Set Programming, we will in this part consider ways of extending this basic formalism, and utilizing the implementation for these extensions. A simple, but fruitful way of re-using the core system is that of a front-end. This concept, which has already been briefly introduced in Chapter 4, consists only of

- a transformation from some formalism to the core language,
- an invocation of the computational core, and
- some post-processing each time an answer has been found (and possibly once after the computation has been completed).

In chapters 6–9 we present several language extensions and front-ends to the core `DLV` system, which have been developed in the recent years.

Chapter 6 introduces a means for qualitative reasoning: Weak Constraints. The basic formalism is generalized by providing a weaker form of constraint, which should be satisfied, but if it is not, it just adds some disadvantage instead of making an answer set impossible. The ASP semantics is extended to those answer sets which minimize such disadvantages. We will sketch how the system can be enhanced to handle this extension. This extension cannot be dealt with by a front-end, as it is computationally more expensive than the core system, so an efficient (polynomial time) reduction (as a front-end would be) is not feasible. Instead, we extend the Model Generator module, described in Chapter 5, and possibly apply it repeatedly.

In Chapter 7, an extension of ASP is defined which allows for declaring hierarchical structures among program rules. Basically, this hierarchy permits an elegant way of modeling exceptions, as knowledge supported by more important rules (with respect to the hierarchy) can yield exceptions to that supported only by less important rules. It turns out that this formalism is of the same computational complexity as basic ASP, which allows for a front-end approach in `DLV`.

While in Chapter 7 a front-end implementing an extension of ASP is described, in Chapter 8 front-ends for tasks which are quite different from ASP are introduced. In particular, we will define front-ends for variations of diagnostic reasoning.

Finally, in Chapter 9 we will give an overview over other existing front-ends for `DLV`.

Chapter 6

Weak Constraints

Pure ASP programs as defined in Definition 2.1.4 require that answer sets satisfy all rules. In this setting, it is impossible to state *desiderata* — properties which are not strictly required to hold, but which *should* hold if possible. We refer to such "soft requirements" as weak constraints. Allowing more than one weak constraint poses the difficulty that some (sets of) weak constraints may be conflicting, and in such cases it is often desirable to define suitable preferential criteria.

In principle, one could define a preference relation over the elements in the power set of all ground weak constraints (each set represents the violated weak constraints w.r.t. some interpretation). However, this is not practical in general. A more reasonable way of defining these preferences is to associate a weight to each weak constraint. The preference among sets of weak constraints is then given by the ordinary less-than relation over the sums of the weights associated to the constraints in a set. Alternatively, one could define a partial order on the set of weak constraints, and define a lexicographic order based on this partial order. In the language we propose, we combine these two approaches in a reasonable way.

6.1 Syntax and Semantics

We define weak constraints as variants of integrity constraints. In order to differentiate clearly between these two, we use the symbol $:\sim$ instead of $\leftarrow$. Additionally, a weight and a priority level or layer (inducing a partial order) of the weak constraint are specified explicitly.

Definition 6.1.1 (Weak Constraints)
A weak constraint *w is an expression of the form*

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

where $m \geq k \geq 0$, $b_1, \ldots, b_m$ are classical literals, w (the weight*), and l (the* level *or* layer*) are both positive integers. For convenience, w and/or l might*

be omitted and are set to 1 by default. A program $\mathcal{P}$ can now also contain weak constraints.

The sets $B(w)$, $B^+(w)$, and $B^-(w)$ of a weak constraint w are defined by viewing w as a regular integrity constraint, and also the grounding of a program is defined by employing this view. For a program $\mathcal{P}$, let $WC(\mathcal{P})$denotes the set of weak constraints in $\mathcal{P}$.

Given a ground program $\mathcal{P}$ with weak constraints, we are interested in the answer sets of the part without weak constraints which minimize the sum of weights of the violated constraints in the highest priority level, and among them those which minimize the sum of weights of the violated constraints in the next lower level, etc. This is expressed by an objective function $H_A^{\mathcal{P}}$ for $\mathcal{P}$ and an answer set A ($f_{\mathcal{P}}$ is an auxiliary function mapping leveled weights to weights without levels):

$$\begin{aligned} f_{\mathcal{P}}(1) &= 1 \\ f_{\mathcal{P}}(n) &= f_{\mathcal{P}}(n-1) \cdot |WC(\mathcal{P})| \cdot w_{max}^{\mathcal{P}} + 1, \quad n > 1 \\ H_A^{\mathcal{P}} &= \sum_{i=1}^{l_{max}^{\mathcal{P}}} (f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{A,\mathcal{P}}} w_N) \end{aligned}$$

where $w_{max}^{\mathcal{P}}$ and $l_{max}^{\mathcal{P}}$ denote the maximum weight and maximum level of a weak constraint in $\mathcal{P}$, respectively; $N_i^{A,\mathcal{P}}$ denotes the set of weak constraints in level i which are violated by A, and w_N denotes the weight of the weak constraint N. Note that $|WC(\mathcal{P})| \cdot w_{max}^{\mathcal{P}} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than any sum of weights of a single level. If weights in level i are multiplied by $f_{\mathcal{P}}(i)$, it is sufficient to calculate the sum of these updated weights, such that the updated weight of a violated constraint of a greater level is always greater than any sum of updated weights of violated constraints of lower levels.

Definition 6.1.2
For a ground program with weak constraints $\mathcal{P}$, A is an (optimal) answer set *of $\mathcal{P}$, if (1) $A \in \mathcal{AS}(\mathcal{P} \backslash WC(\mathcal{P}))$ and (2) $H_A^{\mathcal{P}}$ is minimal over $\mathcal{AS}(\mathcal{P} \backslash WC(\mathcal{P}))$. Let $\mathcal{OAS}(\mathcal{P})$ denote the set of all optimal answer sets.*

Example 6.1.1
Consider the following program $\mathcal{P}_{wc}$, which contains three weak constraints.

$\mathtt{a} \vee \mathtt{b}.$
$\mathtt{b} \vee \mathtt{c}.$
$\mathtt{d} \vee \neg\, \mathtt{d} \leftarrow \mathtt{a}, \mathtt{c}.$
$:\sim \mathtt{a}, \mathtt{c}.\ [2:1]$
$:\sim \neg\, \mathtt{d}.\ [1:1]$
$:\sim \mathtt{b}.\ [3:1]$

$\mathcal{P}_{wc} \setminus WC(\mathcal{P}_{wc})$ admits three answer sets: $\{\mathtt{a}, \mathtt{c}, \mathtt{d}\}$, $\{\mathtt{a}, \mathtt{c}, \neg\, \mathtt{d}\}$, and $\{\mathtt{b}\}$. $\mathcal{OAS}(\mathcal{P}_{wc})$ consists of the single answer set $\{\mathtt{a}, \mathtt{c}, \mathtt{d}\}$ with weight 2 in level 1.

6.2 Implementation

Computing optimal answer sets is more complex than computing answer sets, as those answer sets which are minimal with respect to weak constraint violation have to be selected among all answer sets. An implementation as a simple front-end is therefore not feasible.

Instead, we have integrated weak constraint handling tightly into DLV. Obviously, the program parsing and grounding have to be adapted to accommodate weak constraints. These modifications are quite straightforward and will not be discussed here. The critical change has to be applied inside the Model Generator.

First of all, the Model Generator will in general be applied twice: Once to determine the least cost among all answer sets, and a second time to actually compute those answer sets with that least cost. This two-pass approach avoids a potentially exponential space consumption by doubling the runtime in the worst case.

To see why a one-pass approach would lead to a possibly exponential space consumption, note that the optimal cost of an answer set cannot be known before having computed all answer sets. So all currently optimal answer sets would have to be stored during a one-pass computation — since there might be exponentially many answer sets, exponential space could be consumed by this caching. Nevertheless, we cache the currently best answer set. Frequently, one wants to compute at most one optimal answer set, and in this case pass two can be saved.

So in the presence of weak constraints, the model generator acts in two modes:

1. Optimal cost unknown — compute the optimal cost.

2. Optimal cost fixed — compute answer sets with optimal cost.

The partial interpretations in the Model Generator are extended to include a cost value. So at each point of the computation, the minimum cost of any answer set extending the current partial interpretation is known. DetCons is updated to adjust this value whenever a weak constraint becomes definitely violated. It can also make use of this value, if for example the truth of some atom is necessary to satisfy a weak constraint, which would raise the current cost to a value greater than the currently optimal cost in the first phase. In the second phase, backtracking is possible as soon as a current cost higher than the optimal cost is reached. For heuristics, one can formulate a criterion which tries to keep the cost as low as possible.

These modifications and also the semantics of optimal answer sets are discussed in-depth in [62]. We have included this extension in this thesis to document the whole spectrum of our work on language extensions, which have been conducted so far.

Chapter 7

Inheritance

Answer Set Programs (as defined in Chapter 2) are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [9, 88, 68]. One of the attractions of disjunctive logic programming is its ability to naturally model incomplete knowledge [9, 88]. The need to differentiate between atoms which are false because of the failure to prove them true (NAF) and atoms the falsity of which is explicitly provable led to extend disjunctive logic programs by strong negation [68]. Strong negation, permitted also in the heads of rules, further enhances the knowledge modeling features of the language, and its usefulness is widely acknowledged in the literature [3, 9, 78, 4, 112, 5]. However, it does not always allow to represent default reasoning with exceptions in a direct and natural way. Indeed, to render a default rule r *defeasible*, r must at least be equipped with an extra negative literal, which "blocks" inferences from r for abnormal instances [70]. For instance, to encode the famous nonmonotonic reasoning (NMR) example stating that birds *normally* fly while penguins do not fly, one should write the rule

$$\texttt{fly(X)} \leftarrow \texttt{bird(X)}, \texttt{not} \neg \texttt{fly(X)}.$$

along with the fact

$$\neg \texttt{fly(penguin)}.$$

In [20, 21] we have proposed an extension of disjunctive logic programming by inheritance, called $DLP^{<}$. The addition of inheritance enhances the knowledge modeling features of the language. Possible conflicts are solved in favor of the rules which are "more specific" according to the inheritance hierarchy. This way, a direct and natural representation of default reasoning with exceptions is achieved (e.g., defeasible rules do not need to be equipped with extra literals as above – see Section 7.3).

In this chapter, we will review some aspects of $DLP^{<}$:

- We formally define the $DLP^<$ language, providing a declarative model theoretic semantics of $DLP^<$, which is shown to generalize the Answer Set Semantics of [68].

- We illustrate the knowledge modeling features of the language by encoding classical nonmonotonic problems in $DLP^<$. Interestingly, $DLP^<$ also supplies a very natural representation of frame axioms.

- We analyze the computational complexity of reasoning over $DLP^<$ programs. Importantly, while inheritance enhances the knowledge modeling ability of disjunctive logic programming, it does not cause any computational overhead, as reasoning in $DLP^<$ has exactly the same complexity as reasoning in disjunctive logic programming.

- We compare $DLP^<$ to related work proposed in the literature. In particular, we stress the differences between $DLP^<$ and Disjunctive Ordered Logic ($\mathcal{DOL}$) [22, 23]; we point out the relation to the Answer Set Semantics of [68]; we compare $DLP^<$ with prioritized disjunctive logic programs [112]; we analyze its relationships to inheritance networks [119] and we discuss the possible application of $DLP^<$ to give a formal semantics to updates of logic programs. [2, 90, 79, 53, 54].

- We implement a $DLP^<$ system. To this end, we first design an efficient translation from $DLP^<$ to plain disjunctive logic programming. Then, using this translation, we implement a $DLP^<$ evaluator on top of the `DLV` system [60]. It is part of `DLV` and can be freely retrieved from [63].

The sequel of the chapter is organized as follows. The next two sections provide a formal definition of $DLP^<$; in particular, its syntax is given in Section 7.1 and its semantics is defined in Section 7.2. Section 7.3 shows the use of $DLP^<$ for knowledge representation and reasoning, providing a number of sample $DLP^<$ encodings. The main issues underlying the implementation of our $DLP^<$ system are tackled in Section 7.6.

7.1 Syntax of $DLP^<$

This section provides a formal description of syntactic constructs of the language.

Variables, constants, predicates, atoms, classical literals and NAF literals are defined as in Section 2.1. Additionally, we consider a finite partially ordered set of symbols $(\mathcal{O}, <)$, where $\mathcal{O}$ is a set of strings, called *object identifiers*, and $<$ is a strict partial order (i.e., the relation $<$ is: (1) irreflexive – $c \not< c \ \ \forall c \in \mathcal{O}$, and (2) transitive – $a < b \wedge b < c \Rightarrow a < c \ \ \forall a, b, c \in \mathcal{O}$).

A *rule* r is an expression of the form

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m \ \bigotimes \qquad n \geq 1,\ m \geq 0$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are literals, and $\bigotimes$ is either (1) the symbol '.' or (2) the symbol '!'. In case (1) r is a *defeasible rule*, in case (2) it is a *strict rule*.

The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction b_1, $\ldots$, b_k, $\texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of r. $b_1, \ldots, b_k$ is called the *positive part* of the body of r and $\texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is called the *NAF (negation as failure) part* of the body of r. As in Section 2.1, we denote the sets of literals appearing in the head, in the positive, and in the NAF part of the body of a rule r by $H(r)$, $B^+(r)$, and $B^-(r)$, respectively.

If the body of a rule r is empty, then r is called *fact*. The symbol '$\leftarrow$' is usually omitted from facts.

An *object* o is a pair $\langle oid(o), \Sigma(o) \rangle$, where $oid(o)$ is an object identifier in $\mathcal{O}$ and $\Sigma(o)$ is a (possibly empty) set of rules.

A *knowledge base* on $\mathcal{O}$ is a set of objects, one for each element of $\mathcal{O}$.

Given a knowledge base $\mathcal{K}$ and an object identifier $o \in \mathcal{O}$, the $DLP^<$ *program for* o (*on* $\mathcal{K}$) is the set of objects $\mathcal{P} = \{(o', \Sigma(o')) \in \mathcal{K} \mid o = o' \text{ or } o < o'\}$.

The relation $<$ induces a partial order on $\mathcal{P}$ in the obvious way, that is, given $o_i = (oid(o_i), \Sigma(o_i))$ and $o_j = (oid(o_j), \Sigma(o_j))$, $o_i < o_j$ iff $oid(o_i) < oid(o_j)$ (read "o_i is more specific than o_j").

Informally, a knowledge base can be viewed as a set of *objects* embedding the definition of their properties specified through disjunctive logic rules, organized in an IS-A (inheritance) hierarchy (induced by the relation $<$). A program $\mathcal{P}$ for an object o on a knowledge base $\mathcal{K}$ consists of the portion of $\mathcal{K}$ "seen" from o looking up in the IS-A hierarchy. Thanks to the inheritance mechanism, $\mathcal{P}$ incorporates the knowledge explicitly defined for o plus the knowledge inherited from the higher objects.

If a knowledge base admits a *bottom* element (i.e., an object less than all the other objects, by the relation $<$), we usually refer to the knowledge base as "program", since it is equal to the program for the bottom element.

Moreover, we represent the *transitive reduction* of the relation $<$ on the objects.[1] An object o is denoted as $oid(o) : o_1, \ldots, o_n\ \Sigma(o)$,[2] where $(oid(o), o_1)$, $\ldots$, $(oid(o), o_n)$ are exactly those pairs of the transitive reduction of $<$, in which the first object identifier is $oid(o)$. o is referred to as *sub-object* of $o_1, \ldots, o_n$.

Example 7.1.1
Consider the following program $\mathcal{P}_{10}$*:*

$\texttt{o}_1 \quad \{\ \texttt{a} \vee \neg\, \texttt{b} \leftarrow \texttt{c}, \texttt{not d}. \quad \texttt{e} \leftarrow \texttt{b}!\ \}$
$\texttt{o}_2 : \texttt{o}_1\ \{\ \texttt{b}. \quad \neg\, \texttt{a} \vee \texttt{c}. \quad \texttt{c} \leftarrow \texttt{b}.\ \}$

[1] (a, b) is in the transitive reduction of $<$ iff $a < b$ and there is no c such that $a < c$ and $c < b$.

[2] The set $\Sigma(o)$ is denoted without commas as separators.

$\mathcal{P}_{10}$ consists of two objects $\mathtt{o}_1$ and $\mathtt{o}_2$. $\mathtt{o}_2$ is a sub-object of $\mathtt{o}_1$. According to the convention illustrated above, the knowledge base on which $\mathcal{P}_{10}$ is defined coincides with $\mathcal{P}_{10}$, and the object for which $\mathcal{P}_{10}$ is defined is $\mathtt{o}_2$ (the bottom object).

7.2 Semantics of $DLP^{<}$

In this section we assume that a knowledge base $\mathcal{K}$ is given and an object o has been fixed. Let $\mathcal{P}$ be the $DLP^{<}$program for o on $\mathcal{K}$. The *Universe* $U_{\mathcal{P}}$ and the *Base* $B_{\mathcal{P}}$ of a program $\mathcal{P}$ is defined as in Section 2.2. Let the ground instantiation $Ground(r)$ of a rule r be defined as in Section 2.2. We denote by $Ground(\mathcal{P})$ the (finite) multiset of all instances of the rules occurring in $\mathcal{P}$. The reason why $Ground(\mathcal{P})$ is a multiset is that a rule may appear in several different objects of $\mathcal{P}$, and we require that the respective ground instances are distinct. Hence, we can define a function obj_of from ground instances of rules in $Ground(\mathcal{P})$ into the set $\mathcal{O}$ of the object identifiers, associating with a ground instance $\overline{r}$ of r the (unique) object of r.

A subset of ground literals in $B_{\mathcal{P}}$ is said to be *consistent* if it does not contain a pair of complementary literals. An *interpretation* I is a consistent subset of $B_{\mathcal{P}}$. Given an interpretation $I \subseteq B_{\mathcal{P}}$, a ground literal (either positive or negative) L is *true* w.r.t. I if $L \in I$ holds, and L is *false* w.r.t. I otherwise.

Given a rule $r \in Ground(\mathcal{P})$, the head of r is *true* in I if at least one literal of the head is true w.r.t I. The body of r is *true* in I if: (1) every literal in $B^+(r)$ is true w.r.t. I, and (2) every literal in $B^-(r)$ is false w.r.t. I. A rule r is *satisfied* in I if either the head of r is true in I or the body of r is not true in I.

Next we introduce the concept of a *model* for a $DLP^{<}$-program. Different from traditional logic programming, the notion of satisfiability of rules is not sufficient for this goal, as it does not take into account the presence of explicit contradictions. Hence, we first present some preliminary definitions.

Given two ground rules r_1 and r_2 we say that r_1 *threatens* r_2 *on* a literal L if (1) $\neg.L \in H(r_1)$ and $L \in H(r_2)$, (2) $obj\_of(r_1) < obj\_of(r_2)$ and (3) r_2 is defeasible,

Definition 7.2.1
Given an interpretation I and two ground rules r_1 and r_2 such that r_1 threatens r_2 on L we say that r_1 *overrides* r_2 *on* L *in* I if: (1) $\neg.L \in I$, and (2) the body of r_2 is true in I.

A (defeasible) rule $r \in Ground(\mathcal{P})$ is *overridden in* I if for each $L \in H(r)$ there exists $r_1 \in Ground(\mathcal{P})$ such that r_1 overrides r on L in I.

Intuitively, the notion of overriding allows us to solve conflicts arising between rules with complementary heads. For instance, suppose that both

a and $\neg\, a$ are derivable in I from rules r_a and $r_{\neg\, a}$, respectively. If r_a is more specific than $r_{\neg\, a}$ in the inheritance hierarchy and $r_{\neg\, a}$ is not strict, then $r_{\neg\, a}$ is overridden, meaning that a should be preferred to $\neg\, a$ because it is derivable from a more trustable rule.

Observe that, by definition of overriding, strict rules cannot be overridden, since they are never threatened. Also note that strict rules can be emulated by non-strict rules, if all of them are put into an object which is isolated from all other objects.

Example 7.2.1
Consider the program $\mathcal{P}_{10}$ of Example 7.1.1. Let $\mathtt{I} = \{\neg\, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$ *be an interpretation. Rule* $\neg\, \mathtt{a} \vee \mathtt{c}.$ *in the object* $\mathtt{o}_2$ *overrides rule* $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}.$ *in* $\mathtt{o}_1$ *on the literal* $\mathtt{a}$ *in* $\mathtt{I}$*. Moreover, rule* $\mathtt{b}.$ *in* $\mathtt{o}_2$ *overrides rule* $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}.$ *in* $\mathtt{o}_1$ *on the literal* $\neg\, \mathtt{b}$ *in* $\mathtt{I}$*. Thus, the rule* $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}.$ *in* $\mathtt{o}_1$ *is overridden in* $\mathtt{I}$*.*

Example 7.2.2
Consider the following program $\mathcal{P}_{11}$:

$\mathtt{o}_1 \quad \{\ \neg\, \mathtt{a}! \quad \neg\, \mathtt{b}.\ \}$
$\mathtt{o}_2 : \mathtt{o}_1\ \{\ \mathtt{a} \leftarrow \mathtt{not\ b}. \quad \mathtt{b} \leftarrow \mathtt{not\ a}.\ \}$

Consider now the interpretations $\mathtt{M}_1 = \{\mathtt{a}, \neg\, \mathtt{b}\}$ *and* $\mathtt{M}_2 = \{\mathtt{b}, \neg\, \mathtt{a}\}$*. While the rule* $\neg\, \mathtt{b}.$ *is overridden in* $\mathtt{M}_2$*, the rule* $\neg\, \mathtt{a}!$ *cannot be overridden since it is a strict rule. Due to overriding, strict rules and defeasible rules are quite different from the semantic point of view. In our example, the overriding mechanism allows us to invalidate the defeasible rule* $\neg\, \mathtt{b}.$ *in favor of the more trustable one* $\mathtt{b} \leftarrow \mathtt{not\ a}.$ *(w.r.t. the interpretation* $\mathtt{M}_2$*). In words, the defeasible rule is invalidated in* $\mathtt{M}_2$ *because of a more specific contradictory rule and no inconsistency is generated. In other words, it is possible to find an interpretation containing the literal* $\mathtt{b}$ *(i.e., stating an exception for the rule* $\neg\, \mathtt{b}.$*) such that all the rules of $\mathcal{P}_{11}$ are either satisfied or overridden (i.e., invalidated) in it. Such an interpretation is just* $\mathtt{M}_2$*. This cannot happen for the strict rule* $\neg\, \mathtt{a}!$*. Indeed, no interpretation containing the literal* $\mathtt{a}$ *(i.e., stating an exception for the strict rule) can be found which satisfies all non-overridden rules of $\mathcal{P}_{11}$.* □

In the example above we have implicitly used the notion of *model* for a program $\mathcal{P}$ that we next formally provide. A model for a program is an interpretation satisfying all its non overridden rules.

Definition 7.2.2
Let I be an interpretation for $\mathcal{P}$. Then, I is a *model for* $\mathcal{P}$ if every rule in $Ground(\mathcal{P})$ is satisfied or overridden in I. Moreover, I is a *minimal model for* $\mathcal{P}$ if no (proper) subset of I is a model for $\mathcal{P}$. □

Note that strict rules must be satisfied in every model, since they cannot be overridden.

Example 7.2.3
It is easy to see that $\mathtt{M}_1 = \{\mathtt{a}, \neg\, \mathtt{b}\}$ *is not a model for the program* $\mathcal{P}_{11}$ *of Example 7.2.2 since the rule* $\neg\, \mathtt{a}!$ *is neither overridden nor satisfied. On the contrary,* $\mathtt{M}_2 = \{\mathtt{b}, \neg\, \mathtt{a}\}$ *is a model for* $\mathcal{P}_{11}$*, since* $\neg\, \mathtt{b}.$ *is overridden by the rule* $\mathtt{b} \leftarrow \mathtt{not\ a}$*. The latter rule is satisfied since* $\mathtt{b} \in \mathtt{M}_2$*. The rule* $\neg\, \mathtt{a}!$ *is satisfied as* $\neg\, \mathtt{a} \in \mathtt{M}_2$ *and the rule* $\mathtt{a} \leftarrow \mathtt{not\ b}.$ *is satisfied since both body and head are false w.r.t.* $\mathtt{M}_2$*.*

Next we define the transformation G_I based on which our semantics is defined. This transformation applied to a program $\mathcal{P}$ w.r.t. an interpretation I output a set of rules $G_I(\mathcal{P})$ with no negation by failure in the body. Intuitively, such rules are those remaining from $Ground(\mathcal{P})$ by (1) eliminating the rules overridden in the interpretation I, (2) deleting rules whose NAF part is not "true" in I (i.e., some literal negated by negation as failure occurs in I) and (3) deleting the NAF part of all the remainder rules. Since the transformation encodes the overriding mechanism, the distinction between strict rules and defeasible rules in $G_I(\mathcal{P})$ is meaningless (indeed, there is no difference between strict and defeasible rules except for the overriding mechanism where the upper rule is required to be defeasible). For this reason the syntax of rules in $G_I(\mathcal{P})$ can be simplified by dropping the symbol . from defeasible rules and the symbol ! from strict rules.

Definition 7.2.3
Given an interpretation I for $\mathcal{P}$, the *reduction of* $\mathcal{P}$ *w.r.t.* I, denoted by $G_I(\mathcal{P})$, is the set of rules obtained from $Ground(\mathcal{P})$ by (1) removing every rule overridden in I, (2) removing every rule r such that $B^-(r) \cap I \neq \emptyset$, (3) removing the NAF part from the bodies of the remaining rules. □

Example 7.2.4
Consider the program $\mathcal{P}_{10}$ *of Example 7.1.1. Let* $\mathtt{I}$ *be the interpretation* $\{\neg\, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$*. As shown in Example 7.2.1, rule* $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}, \mathtt{not\ d}.$ *is overridden in* $\mathtt{I}$*. Thus,* $G_I(\mathcal{P}_{10})$ *is the set of rules* $\{\neg\, \mathtt{a} \vee \mathtt{c}. \quad \mathtt{e} \leftarrow \mathtt{b}. \quad \mathtt{b}. \quad \mathtt{c} \leftarrow \mathtt{b}.\}$*. Consider now the interpretation* $\mathtt{M} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{e}\}$*. It is easy to see that* $G_M(\mathcal{P}_{10}) = \{\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}. \quad \neg\, \mathtt{a} \vee \mathtt{c}. \quad \mathtt{e} \leftarrow \mathtt{b}. \quad \mathtt{b}. \quad \mathtt{c} \leftarrow \mathtt{b}.\}$*.*

We observe that the reduction of a program is simply a set of ground rules. Given a set S of ground rules, we denote by $pos(S)$ the positive disjunctive program (called the *positive version of* S), obtained from S by considering each negative literal $\neg\, p(\bar{X})$ as a positive one with predicate symbol $\neg\, p$.

Definition 7.2.4
Let M be a model for $\mathcal{P}$. We say that M is a *($DLP^<$-)answer set* for $\mathcal{P}$ if M is a minimal model of the positive version $pos(G_M(\mathcal{P}))$ of $G_M(\mathcal{P})$. □

Note that interpretations must be consistent by definition, so considering $pos(G_M(\mathcal{P}))$ instead of $G_M(\mathcal{P})$ does not lose information in this respect.

Note that the notion of minimal model of Definition 7.2.2 cannot be used in Definition 7.2.4, as G_M is a set of rules and not a $DLP^<$ program.

Example 7.2.5
Consider the program $\mathcal{P}_{10}$ of Example 7.1.1:

It is easy to see that the interpretation I *of Example 7.2.4 is not an answer set for $\mathcal{P}_{10}$. Indeed, although* I *is a model for $pos(G_I(\mathcal{P}_{10}))$ it is not minimal, since the interpretation $\{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$ is a model for $pos(G_I(\mathcal{P}_{10}))$, too. Note that the interpretation $\mathtt{I}' = \{\mathtt{b}, \mathtt{c}, \mathtt{e}\}$ is not an answer set for $\mathcal{P}_{10}$. Indeed, $G_{I'}(\mathcal{P}_{10}) = \{\ \mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}. \quad \neg\, \mathtt{a} \vee \mathtt{c}. \quad \mathtt{e} \leftarrow \mathtt{b}. \quad \mathtt{b}. \quad \mathtt{c} \leftarrow \mathtt{b}.\ \}$ and $\mathtt{I}'$ is not a model for $pos(G_{I'}(\mathcal{P}_{10}))$, since the rule $\mathtt{a} \vee \neg\, \mathtt{b} \leftarrow \mathtt{c}.$ is not satisfied in $\mathtt{I}'$.*

On the other hand, the interpretation M *of Example 7.2.4 is an answer set for $\mathcal{P}_{10}$, since* M *is a minimal model for $pos(G_M(\mathcal{P}_{10}))$. Moreover, it can be easily realized that* M *is the only answer set for $\mathcal{P}_{10}$.*

Finally, the program $\mathcal{P}_{11}$ of Example 7.2.2 admits one consistent answer set $\mathtt{M}_2 = \{\mathtt{b}, \neg\, \mathtt{a}\}$. Note that if we replace the strict rule $\neg\, \mathtt{a}!$ by a defeasible rule $\neg\, \mathtt{a}.$, $\mathcal{P}_{11}$ admits two answer sets, namely $\mathtt{M}_1 = \{\mathtt{a}, \neg\, \mathtt{b}\}$ and $\mathtt{M}_2 = \{\mathtt{b}, \neg\, \mathtt{a}\}$. Asserting $\neg\, \mathtt{a}$ by a strict rule, prunes the answer set $\mathtt{M}_1$ stating the exception (truth of the literal a*) to this rule.*

It is worthwhile noting that if a rule r is not satisfied in a model M, then *all* literals in the head of r must be overridden in M.

Let $\mathcal{P}_1$ be the program

$\mathtt{o}_3 \quad \{\ \mathtt{a} \vee \mathtt{b}. \quad \leftarrow \mathtt{b}.\ \}$
$\mathtt{o}_2 : \mathtt{o}_3 \ \{\ \neg\, \mathtt{a}.\ \}$

and $\mathcal{P}_2$

$\mathtt{o}_1 \quad \{\ \mathtt{a}. \quad \leftarrow \mathtt{b}.\ \}$
$\mathtt{o}_2 : \mathtt{o}_1 \ \{\ \neg\, \mathtt{a}.\ \}$

Then, $\{\neg\, \mathtt{a}\}$ is not a model for program $\mathcal{P}_1$, because the head literal b in the head of $\mathtt{a} \vee \mathtt{b}.$ is not overridden in M. If we drop b from rule $\mathtt{a} \vee \mathtt{b}.$, then $\{\neg\, \mathtt{a}\}$ is a model of the resulting program $\mathcal{P}_2$.

Observe also that two programs having the same answer sets, such as $\mathtt{o}_1$ and $\mathtt{o}_3$ (both have the single answer set $\{\neg\, \mathtt{a}\}$), may get different answer sets even if we add the same object to both of them. Indeed, program $\mathcal{P}_1$ has no answer set, while program $\mathcal{P}_2$ has the answer set $\{\neg\, \mathtt{a}\}$.

This is not surprising, as a similar phenomenon also arises in normal logic programming where $\mathcal{P}_1 = \{\mathtt{a}.\}$ and $\mathcal{P}_2 = \{\mathtt{a} \leftarrow \mathtt{not}\ \mathtt{b}.\}$ have the same answer set $\{\mathtt{a}\}$, while $\mathcal{P}_1 \cup \{\mathtt{b}.\}$ and $\mathcal{P}_2 \cup \{\mathtt{b}.\}$ have different answer sets ($\{\mathtt{a}, \mathtt{b}\}$ and $\{\mathtt{b}\}$, respectively).

Finally we show that each answer set of a program $\mathcal{P}$ is also a minimal model of $\mathcal{P}$:

Proposition 7.2.1
If M is an answer set for $\mathcal{P}$, then M is a minimal model of $\mathcal{P}$.

Proof By contradiction suppose M' is a model for $\mathcal{P}$ such that $M' \subset M$.

First we show that M' is a model for $pos(G_M(\mathcal{P}))$ too, i.e., every rule in $pos(G_M(\mathcal{P}))$ is satisfied in M'. Recall that $pos(G_M(\mathcal{P}))$ is the positive version of the program obtained by applying the transformation G_M to the program $Ground(\mathcal{P})$. Consider a generic rule r of $Ground(\mathcal{P})$. Since M' is a model for $\mathcal{P}$ either (i) r is overridden in M' or (ii) is satisfied in M'.

In case (i), since $M' \subset M$, from Definition 7.2.1 immediately follows that r is overridden in M too. Thus, r does not occur in $G_M(\mathcal{P})$ since the transformation G_M removes all rules overridden in M.

In case (ii) (i.e., r is satisfied in M'), if r is such that $B^-(r) \cap M \neq \emptyset$, then the rule is removed by G_M. Otherwise, r is transformed by G_M into a rule r_1 obtained from r by dropping the NAF part from the body. Since r is satisfied in M', also r_1 is satisfied in M'. As a consequence, all the rules of $pos(G_M(\mathcal{P}))$ are satisfied in M', that is $M' \subset M$ is a model for $pos(G_M(\mathcal{P}))$. Thus, by Definition 7.2.4, M is not an answer set for $\mathcal{P}$ since it is not a minimal model of $pos(G_M(\mathcal{P}))$. The proof is hence complete.

7.3 Knowledge Representation with $DLP^<$

In this section, we present a number of examples which illustrate how knowledge can be represented using $DLP^<$. To start, we show the $DLP^<$ encoding of a classical example of nonmonotonic reasoning.

Example 7.3.1
Consider the following program $\mathcal{P}_{peng}$ with $\mathcal{O}(\mathcal{P}_{peng})$ consisting of three objects `bird`, `penguin` *and* `tweety`, *such that* `penguin` *is a sub-object of* `bird` *and* `tweety` *is a sub-object of* `penguin`*:*

```
bird             { flies. }
penguin : bird   { ¬ flies! }
tweety : penguin {  }
```

Unlike in traditional logic programming, our language supports two types of negation, that is strong negation *and* negation as failure. *Strong negation is useful to express negative pieces of information under the complete information assumption. Hence, a negative fact (by strong negation) is true only if it is explicitly derived from the rules of the program. As a consequence, the head of rules may contain also such negative literals and rules can be conflicting on some literals. According to the inheritance principles,*

the ordering relationship between objects can help us to assign different levels of reliability to the rules, allowing us to solve possible conflicts. For instance, in our example, the contradicting conclusion tweety both flies and does not fly *seems to be entailed from the program (as* `tweety` *is a* `penguin` *and* `penguin`*s are* `bird`*s, both* `flies` *and* $\neg$ `flies` *can be derived from the rules of the program). However, this is not the case. Indeed, the "lower" rule* $\neg$ `flies`. *specified in the object* `penguin` *is considered as a sort of refinement to the first general rule, and thus the meaning of the program is rather clear:* tweety does not fly, *as tweety is a penguin. That is,* $\neg$ `flies`. *is preferred to the default rule* `flies`. *as the hierarchy explicitly states the specificity of the former. Intuitively, there is no doubt that* $\mathtt{M} = \{\neg\, \mathtt{flies}\}$ *is the only reasonable conclusion.*

The next example, from the field of database authorizations, combines the use of both weak and strong negation.

Example 7.3.2
Consider the following knowledge base representing a set of security specification about a simple part-of *hierarchy of objects.*

$\mathtt{o}_1\{$
$\quad \mathtt{authorize(bob)} \leftarrow \mathtt{not\ authorize(ann)}.$ (7.1)
$\quad \mathtt{authorize(ann)} \vee \mathtt{authorize(tom)} \leftarrow \mathtt{not}\ \neg\, \mathtt{authorize(alice)}.$ (7.2)
$\quad \mathtt{authorize(amy)}!$ (7.3)
$\}$
$\mathtt{o}_2 : \mathtt{o}_1\{$
$\quad \neg\, \mathtt{authorize(alice)}!$ (7.4)
$\}$
$\mathtt{o}_3 : \mathtt{o}_1\{$
$\quad \neg\, \mathtt{authorize(bob)}!$ (7.5)
$\}$

Object $\mathtt{o}_2$ *is part-of the object* $\mathtt{o}_1$ *as well as* $\mathtt{o}_3$ *is part-of* $\mathtt{o}_1$. *Access authorizations to objects are specified by rules with head predicate* `authorize` *and subjects to which authorizations are granted appear as arguments. Strong negation is utilized to encode negative authorizations that represent explicit denials. Negation as failure is used to specify the absence of authorization (either positive or negative). Inheritance implements the automatic propagation of authorizations from an object to all its sub-objects. The overriding mechanism allows us to represent exceptions: for instance, if an object o inherits a positive authorization but a denial for the same subject is specified in o, then the negative authorization prevails on the positive one. Possible*

loss of control due to overriding mechanism can be avoided by using strict rules: strict authorizations cannot be overridden.

Consider the program $\mathcal{P}_{o_2} = \{(\mathtt{o_1}, \{(7.1), (7.2), (7.3)\}), (\mathtt{o_2}, \{(7.4)\})\}$ *for the object* $\mathtt{o_2}$ *on the above knowledge base. This program defines the access control for the object* $\mathtt{o_2}$*. Thanks to the inheritance mechanism, authorizations specified for the object* $\mathtt{o_1}$*, to which* $\mathtt{o_2}$ *belongs, are propagated also to* $\mathtt{o_2}$*. It consists of rules (7.1), (7.2) and (7.3) (inherited from* $\mathtt{o_1}$*) and (7.4). Rule (7.1) states that* `bob` *is authorized to access object* $\mathtt{o_2}$ *provided that no authorization for* `ann` *to access* $\mathtt{o_2}$ *exists. Rule (7.2) authorizes either* `ann` *or* `tom` *to access* $\mathtt{o_2}$ *provided that no denial for* `alice` *to access* $\mathtt{o_2}$ *is derived. The strict rule (7.3) grants to* `amy` *the authorization to access object* $\mathtt{o_1}$*. Such an authorization can be considered "strong", since no exceptions can be stated to it without producing inconsistency. As a consequence, all the answer sets of the program contain the authorization for* `amy`*. Finally, rule (7.4) defines a denial for* `alice` *to access object* $\mathtt{o_2}$*. Due to the absence of the authorization for* `ann`*, the authorization to* `bob` *of accessing the object* $\mathtt{o_2}$ *is derived (by rule (7.1)). Further, the explicit denial to access the object* $\mathtt{o_2}$ *for* `alice` *(rule (7.4)) allows to derive neither authorization for* `ann` *nor for* `tom` *(by rule (7.2)). Hence, the only answer set of this program is* $\{\mathtt{authorize(bob)}, \neg\, \mathtt{authorize(alice)}, \mathtt{authorize(amy)}\}$.

Consider now the program $\mathcal{P}_{o_3} = \{(\mathtt{o_1}, \{(7.1), (7.2)\}), (\mathtt{o_3}, \{(7.5)\})\}$ *for the object* $\mathtt{o_3}$*. Rule (7.5) defines a denial for* `bob` *to access object* $\mathtt{o_3}$*. The authorization for* `bob` *(defined by rule (7.1)) is no longer derived. Indeed, even if rule (7.1) allows to derive such an authorization due to the absence of authorizations for* `ann`*, it is overridden by the explicit denial (rule (7.5)) defined in the object* $\mathtt{o_3}$ *(i.e., at a more specific level). The body of rule (7.2) inherited from* $\mathtt{o_1}$ *is true for this program since no denial for alice can be derived, and it entails a mutual exclusive access to object* $\mathtt{o_3}$ *for* `ann` *and* `tom` *(note that no other head contains* `authorize(ann)` *or* `authorize(bob)`*). The program* $\mathcal{P}_{o_3}$ *admits two answer sets, namely* $\{\mathtt{authorize(ann)}, \neg\, \mathtt{authorize(bob)}, \mathtt{authorize(amy)}\}$ *and* $\{\mathtt{authorize(tom)}, \neg\, \mathtt{authorize(bob)}, \mathtt{authorize(amy)}\}$ *representing two alternative authorization sets to grant the access to the object* $\mathtt{o_3}$.

Solving the Frame Problem

The frame problem has first been addressed by McCarthy and Hayes [94], and in the meantime a lot of research has been conducted to overcome it (see e.g. [116] for a survey).

In short, the frame problem arises in planning, when actions and fluents are specified: An action affects some of the fluents, but all unrelated fluents should remain as they are. In most formulations using classical logic, one must specify for every pair of actions and unrelated fluents that the fluent remains unchanged. Clearly this is an undesirable overhead, since with n

actions and m fluents, $n \times m$ clauses would be needed.

Instead, it would be nice to be able to specify for each fluent that it *"normally remains valid"* and that only actions which explicitly entail the contrary can change them.

Indeed, this goal can be achieved in a very elegant way using $DLP^{<}$: One object contains the rules which specify *inertia* (the fact that fluents normally do not change). Another object inherits from it and specifies the actions and the effects of actions — in this way a very natural, straightforward and effective representation is achieved, which avoids the frame problem.

Example 7.3.3
As an example we show how the famous Yale Shooting Problem, which is due to Hanks and McDermott [72], can be represented and solved with $DLP^{<}$*:*

The scenario involves an individual, who can be shot with a gun (or who can be photographed, in a less violent version). There are two fluents, `alive` *and* `loaded`*, which intuitively mean that the individual is alive and that the gun is loaded, respectively. There are three actions,* `load`*,* `wait` *and* `shoot`*. Loading has the effect that the gun is loaded afterwards, shooting with the loaded gun has the effect that the individual is no longer alive afterwards (and also that the gun is unloaded, but this not really important), and waiting has no effects.*

The problem involves temporal projection*: It is known that initially the individual is alive, and that first the gun is loaded, and after waiting, the gun is shot with. The question is: Which fluents hold after these actions and between them?*

In our encoding, the `inertia` *object contains the defaults for the fluents, the* `domain` *object additionally specifies the effects of actions, while the* `yale` *object encodes the problem instance.*

For the time framework we use the `DLV` *bounded integer built-ins: The upper bound* `n` *of positive integers is specified by either adding the fact* `#maxint = n.` *to the program or by passing the option* `-N = n` *on the command-line (this overrides any* `#maxint = n.` *statement). It is then possible to use the built-in constant* `#maxint`*, which evaluates to the specified upper bound, and several built-in predicates, of which in this work we just use* `#succ(N, N1)` *, which holds if* `N1` *is the successor of* `N` *and* `N1` $\leq$ `#maxint`*.*

```
inertia
  {
    alive(T1) ← alive(T), #succ(T, T1).                    (7.6)
    ¬ alive(T1) ← ¬ alive(T), #succ(T, T1).                (7.7)
    loaded(T1) ← loaded(T), #succ(T, T1).                  (7.8)
    ¬ loaded(T1) ← ¬ loaded(T), #succ(T, T1).              (7.9)
  }
```

```
domain : inertia
  {                                                                    (7.10)
     loaded(T1) ← load(T), #succ(T, T1)!                               (7.11)
     ¬ loaded(T1) ← shoot(T), loaded(T), #succ(T, T1)!                 (7.12)
     ¬ alive(T1) ← shoot(T), loaded(T), #succ(T, T1)!                  (7.13)
  }
yale : domain
  {                                                                    (7.14)
     load(0)!  wait(1)!  shoot(2)!  alive(0)!                          (7.15)
  }
```

The only answer set for this program (and #maxint = 3*) contains, besides the facts of the* yale *object,* loaded(1), loaded(2), alive(0), alive(1), alive(2) *and* ¬ loaded(3), ¬ alive(3)*. That is, the individual is alive until the shoot action is taken, and no longer alive afterwards, and the gun is loaded between loading and shooting.*

We want to point out that this formalism is equally suited for solving problems which involve finding a plan (i.e. a sequence of actions) rather than doing temporal projection (determining the effects of a given plan) as in the Yale Shooting Problem: We have to add a rule

$$\texttt{action(T)} \vee \neg\, \texttt{action(T)} \leftarrow \#\texttt{succ(T, T1)}.$$

for every action, and we have to specify the goal state by a query, e.g. ¬ alive(3), ¬ loaded(3)?.

Below we consider a classical plan-finding example: The blocksworld domain and the Sussman anomaly as a concrete problem.

Example 7.3.4
In [61], several planning problems, including the blocksworld problems, are encoded using disjunctive datalog.

In general, planning problems can be effectively specified using action languages (e.g. [69, 42, 71, 86]). Then, a translation from these languages to another language (in our case $DLP^{<}$) is applied.

We omit the step of describing an action language and the associated translation, and directly show the encoding of an example planning domain in disjunctive datalog. This encoding is rather different from the one presented in [61].

The objects in the blocksworld are one table *and an arbitrary number of labeled cubic* blocks*. Together, they are referred to as* locations*.*

The state of the blocksworld at a particular time can be fully specified by the fluent on(B, L, T)*, which specifies that block* B *resides on location* L *at time* T*.*

So, first we state in the object `bw_inertia` *that the fluent* `on` *is inertial.*

```
bw_inertia
  {
     on(B, L, T1) ← on(B, L, T), #succ(T, T1).                    (7.16)
  }
```

We continue to define the blocksworld domain in the object `bw_domain`*, which inherits from the inertia object:*

```
bw_domain : bw_inertia
  {
     move(B, L, T) ∨ ¬ move(B, L, T) ← block(B), loc(L), #succ(T, T1)!  (7.17)
     on(B, L, T1) ← move(B, L, T), #succ(T, T1)!                        (7.18)
     ¬ on(B, L, T1) ← move(B, L1, T), on(B, L, T), #succ(T, T1)!        (7.19)
     ← move(B, L, T), on(B1, B, T).                                     (7.20)
     ← move(B, B1, T), on(B2, B1, T), block(B1).                        (7.21)
     ← move(B, B, T).                                                   (7.22)
     ← move(B, L, T), move(B1, L1, T), B <> B1.                         (7.23)
     ← move(B, L, T), move(B1, L1, T), L <> L1.                         (7.24)
     loc(table)!                                                        (7.25)
     loc(B) ← block(B)!                                                 (7.26)
  }
```

There is one action, which is moving a block from one location to another location. A move is started at one point in time, and it is completed before the next time. Rule (7.17) expresses that at any time `T`*, the action of moving a block* `B` *to location* `L` *may be initiated (*`move(B, L, T)`*) or not (*¬ `move(B, L, T)`*).*

Rules (7.18) and (7.19) specify the effects of the move action: The moved block is at the target location at the next time, and no longer on the source location.

(7.20) – (7.24) are constraints, and their semantics is that in any answer set the conjunction of their literals must not be true.[3] *Their respective meanings are: (7.20): A moved block must be clear. (7.21): The target of a move must be clear if it is a block (the table may hold an arbitrary number of blocks). (7.22) A block may not be on itself. (7.23) and (7.24): No two move actions may be performed at the same time.*

The time-steps are again represented by `DLV`*'s integer built-in predicates and constants.*

[3] We use constraints for clarity, but they can be eliminated by rewriting $\leftarrow B.$ to `p` $\leftarrow B,$ `not p.`, where `p` is a new symbol which does not appear anywhere else in the program.

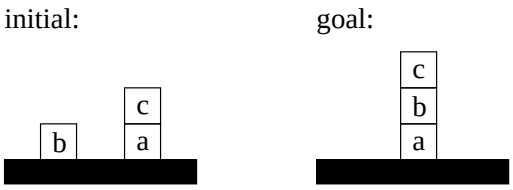

Figure 7.1: The Sussman Anomaly

What is left is the concrete problem instance, in our case the so-called Sussman Anomaly (see Figure 7.1):

```
sussman : bw_domain
  {
    block(a)!   block(b)!   block(c)!                          (7.27)
    on(b, table, 0)!   on(c, a, 0)!   on(a, table, 0)!         (7.28)
  }
on(c, b, #maxint), on(b, a, #maxint), on(a, table, #maxint)?   (7.29)
```

Since different problem instances may involve different numbers of blocks, the blocks are defined as facts (7.27) together with the problem instance.[4]

We give the initial situation by facts (7.28), while the goal situation is specified by query (7.29). This query enforces that only those answer sets are computed, in which the conjunction of the query literals is true.

7.4 Computational Complexity

As for the classical nonmonotonic formalisms [89, 91, 109], two important decision problems, corresponding to two different reasoning tasks, arise in $DLP^{<}$:

Brave Reasoning: Given a $DLP^{<}$ program $\mathcal{P}$ and a ground literal L, decide whether there exists an answer set M for $\mathcal{P}$ such that L is true w.r.t. M.

Cautious Reasoning: Given a $DLP^{<}$ program $\mathcal{P}$ and a ground literal L, decide whether L is true in all answer sets for $\mathcal{P}$.

We next prove that the complexity of reasoning in $DLP^{<}$ is exactly the same as in traditional disjunctive logic programming. That is, inheritance comes for free, as the addition of inheritance does not cause any computational overhead. We consider the propositional case, i.e., we consider ground $DLP^{<}$ programs.

[4]Note that usually the instance will be separated from the domain definition.

Lemma 7.4.1
Given a ground $DLP^<$ program $\mathcal{P}$ and an interpretation M for $\mathcal{P}$, deciding whether M is an answer set for $\mathcal{P}$ is in coNP.

Proof We check in NP that M **is not** an answer set of $\mathcal{P}$ as follows. Guess a subset I of M, and verify that: (1) M is not a model for $pos(G_M(\mathcal{P}))$, or (2) I is a model for $pos(G_M(\mathcal{P}))$ and $I \subset M$. The construction of $pos(G_M(\mathcal{P}))$ (see Definition 7.2.3) is feasible in polynomial time, and the tasks (1) and (2) are clearly tractable. Thus, deciding whether M is not an answer set for $\mathcal{P}$ is in NP, and, consequently, deciding whether M is an answer set for $\mathcal{P}$ is in coNP. □

Theorem 7.4.1
Brave Reasoning on $DLP^<$ programs is Σ_2^P-complete.

Proof Given a ground $DLP^<$ program $\mathcal{P}$ and a ground literal L, we verify that L is a brave consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is true w.r.t. M. Task (2) is clearly polynomial; while (1) is in coNP, by virtue of Lemma 7.4.1. The problem therefore lies in Σ_2^P.

Σ_2^P-hardness follows from Theorem 7.5.1 and the results in [43, 56]. □

Theorem 7.4.2
Cautious Reasoning on $DLP^<$ programs is Π_2^P-complete.

Proof Given a ground $DLP^<$ program $\mathcal{P}$ and a ground literal L, we verify that L is not a cautious consequence of $\mathcal{P}$ as follows. Guess a set $M \subseteq B_\mathcal{P}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is not true w.r.t. M. Task (2) is clearly polynomial; while (1) is in coNP, by virtue of Lemma 7.4.1. Therefore, the complement of cautious reasoning is in Σ_2^P, and cautious reasoning is in Π_2^P.

Π_2^P-hardness follows from Theorem 7.5.1 and the results in [43, 56]. □

7.5 Related Work

7.5.1 Answer Set Semantics

Answer Set Semantics, proposed by Gelfond and Lifschitz in [68], is the most widely acknowledged semantics for disjunctive logic programs with strong negation. For this reason, while defining the semantics of our language, we took care of ensuring full agreement with Answer Set Semantics (on inheritance-free programs).

Theorem 7.5.1
Let $\mathcal{P}$ be a $DLP^<$ program consisting of a single object $o = \langle oid(o), \Sigma(o)\rangle$.[5] *Then, M is an answer set of $\mathcal{P}$ if and only if it is a consistent answer set of $\Sigma(o)$ (as defined in [68]).*

Proof First we show that $G_M(\mathcal{P})$ is equal to $\Sigma(o)^M$ (as defined in [68]):

Deletion rule (1) of Definition 7.2.3 never applies, since for every literal L and any two rules r_1, $r_2 \in Ground(\mathcal{P})$, $obj\_of(r_1) \not< obj\_of(r_2)$ holds, thus violating condition (1) in Definition 7.2.1 and therefore no rule can be overridden. It is evident that the deletion rules (2) and (3) of Definition 7.2.3 are equal to deletion rules (i) and (ii) of the definition of Π^S in §7 in [68], respectively. The first ones delete rules, where some NAF literal is contained in M, while the second ones delete all NAF literals of the remaining rules.

Next, we show that the criteria for a consistent set M of literals being an answer set of a positive (i.e. NAF free) program (as in [68]) is equal to the notion of satisfaction:

Since the set is consistent, condition (ii) in §7 of [68] does not apply. Condition (i) says: $L_{k+1}, \ldots, L_m \in M$ (the body is true) implies that the head is true. This is logically equivalent to "The body is not true or the head is true", which is the definition of rule satisfaction.

In total we have that the minimal models of $pos(G_M(\mathcal{P}))$ are equal to the consistent answer sets of $\Sigma(o)^M$, since answer sets are minimal by definition.

Additionally, we require in Definition 7.2.4 that M is also a model of $\mathcal{P}$, while in [68] there is no such requirement. However, all minimal models of $pos(G_M(\mathcal{P}))$ are also models of $\mathcal{P}$: All rules in $G_M(\mathcal{P})$ are satisfied, and only the deletion rules (2) and (3) of Definition 7.2.3 have been applied (as shown above). So, for any rule r, which has been deleted by (2), some literal in $B^-(r)$ is in M, so r's body is not true, and thus r is satisfied in M. If a rule r, which has been transformed by (3), is satisfied without $B^-(r)$, then either $H(r)$ is true or $B^+(r)$ is not true, so adding any NAF part to it does not change its satisfaction status. □

Theorem 7.5.1 shows that the set of rules contained in a single object of a $DLP^<$ program has precisely the same answer sets (according to the definition in [68]) as the single object program (according to Definition 7.2.4).

For a $DLP^<$ program $\mathcal{P}$ consisting of more than one object, the answer sets (as defined in [68]) of the collection of all rules in $\mathcal{P}$ in general do not coincide with the answer sets of $\mathcal{P}$.

For instance the program

```
o      { p.  }
o1 : o { ¬ p.  }
```

[5] On inheritance-free programs, there is no difference between strict and defeasible rules. Therefore, without loss of generality we assume that rules are of only one type here. This allows us to drop the symbol ('.' or '!') at the end of the rules of single object programs.

has the answer set $\{\neg\, \mathtt{p}\}$, while the disjunctive logic program $\{\mathtt{p}.\ \neg\, \mathtt{p}.\}$ does not have a consistent answer set.

Nevertheless, in Section 7.6 we will show that each $DLP^<$ program $\mathcal{P}$ can be translated into a disjunctive logic program $\mathcal{P}'$, the semantics of which is equivalent to the semantics of $\mathcal{P}$. However, this translation requires the addition of a number of extra predicates.

7.5.2 Disjunctive Ordered Logic

Disjunctive Ordered Logic ($\mathcal{DOL}$) is an extension of Disjunctive Logic Programming with strong negation and inheritance (without default negation) proposed in [22, 23]. The $DLP^<$ language incorporates some ideas taken from $\mathcal{DOL}$. However, the two languages are very different in several respects. Most importantly, unlike with $DLP^<$, even if a program belongs to the common fragment of $\mathcal{DOL}$ and of the language of [68] (i.e., it contains neither inheritance nor default negation), $\mathcal{DOL}$ semantics is completely different from Answer Set Semantics, because of a different way of handling contradictions.[6] In short, we observe the following differences between $\mathcal{DOL}$ and $DLP^<$:

- $\mathcal{DOL}$ does not include default negation `not` , while $DLP^<$ does.
- $\mathcal{DOL}$ and $DLP^<$ have different semantics on the common fragment. Consider a program $\mathcal{P}$ consisting of a single object $o = \langle oid(o), \Sigma(o)\rangle$, where $\Sigma(o) = \{\mathtt{p}.\quad \neg\, \mathtt{p}.\}$. Then, according to $\mathcal{DOL}$, the semantics of $\mathcal{P}$ is given by two models, namely, $\{\mathtt{p}\}$ and $\{\neg\, \mathtt{p}\}$. On the contrary, $\mathcal{P}$ has no answer set according to $DLP^<$ semantics.
- $DLP^<$ generalizes (consistent) Answer Set Semantics to disjunctive logic programs with inheritance, while $\mathcal{DOL}$ does not.

7.5.3 Prioritized Logic Programs

$DLP^<$ can be also seen as an attempt to handle priorities in disjunctive logic programs (the lower the object in the inheritance hierarchy, the higher the priority of its rules).

There are several works on preference handling in logic programming [35, 18, 70, 100, 78, 107, 112]. However, we are aware of only one previous work on priorities in **disjunctive** programs, namely, the paper by Sakama and Inoue [112]. This interesting work can be seen as an extension of Answer Set Semantics to deal with priorities. Comparing the two approaches under the perspective of priority handling, we observe the following:

[6] Actually, this was a main motivation for the authors to look for a different language.

- On priority-free programs, the two languages yield essentially the same semantics, as they generalize Answer Set Semantics and Consistent Answer Set Semantics, respectively.

- In [112], priorities are defined among **literals**, while priorities concern program **rules** in $DLP^{<}$.

- The different kind of priorities (on rules vs. literals) and the way how they are dealt with in the two approaches imply different complexity in the respective reasoning tasks. Indeed, from the simulation of abductive reasoning in the language of [112], and the complexity results on abduction reported in [55], it follows that brave reasoning is Σ_3^P-complete for the language of [112]. On the contrary, brave reasoning is "only" Σ_2^P-complete in $DLP^{<}$[7].

[35] deals with non-disjunctive programs, but the authors note that their semantics-defining transformation "is also applicable to disjunctive logic programs". In this formalism, the preference relation is defined by regular atoms (with a set of constants representing the rules), allowing the definition of dynamic preferences. However, the semantics of the preferences is based on the order of rule application (or defeating) and thus seems to be quite different from our approach.

A comparative analysis of the various approaches to the treatment of preferences in ($\vee$-free) logic programming has been carried out in [18].

7.5.4 Inheritance Networks

From a different perspective, the objects of a $DLP^{<}$ program can also be seen as the nodes of an inheritance network.

We next show that $DLP^{<}$ satisfies the basic semantic principles which are required for inheritance networks in [119].

The book [119] constitutes a fundamental attempt to present a formal mathematical theory of multiple inheritance with exceptions. The starting point of this work is the consideration that an intuitively acceptable semantics for inheritance must satisfy two basic requirements:

1. Being able to reason with redundant statements, and

2. not making unjustified choices in ambiguous situations.

Touretzky illustrates this intuition by means of two basic examples.

The former requirement is presented by means of the *Royal Elephant* example, in which we have the following knowledge: "Elephants are gray.", "Royal elephants are elephants.", "Royal elephants are not gray.", "Clyde is a royal elephant.", "Clyde is an elephant."

[7] We refer to the complexity in the propositional case here.

The last statement is clearly redundant; however, since it is consistent with the others there is no reason to rule it out. Touretzky shows that an intuitive semantics should be able to recognize that Clyde is not gray, while many systems fail in this task.

Touretzky's second principle is shown by the *Nixon diamond* example, in which the following is known: "Republicans are not pacifists.", "Quakers are pacifists.", "Nixon is both a Republican and a quaker."

According to our approach, he claims that a good semantics should draw no conclusion about the question whether Nixon is a $pacifist$.

The proposed solution for the problems above is based on a topological relation, called *inferential distance ordering*, stating that an individual A is "closer" to B than to C iff A has an inference path *through* B to C. If A is "closer" to B than to C, then as far as A is concerned, information coming from B must be preferred w.r.t. information coming from C. Therefore, since Clyde is "closer" to being a royal elephant than to being an elephant, he states that Clyde is not gray. On the contrary no conclusion is taken on Nixon, as there is not any relationship between quaker and republican.

The semantics of $DLP^<$ fully agrees with the intuition underlying the *inferential distance ordering*.

Example 7.5.1
Let us represent the Royal Elephant example in our framework:

`elephant`	{`gray`.}
`royal_elephant` : `elephant`	{¬ `gray`.}
`clyde` : `elephant`, `royal_elephant`	{ }

The only answer set of the above $DLP^<$ program is {¬ `gray`}.

The Nixon Diamond example can be expressed in our language as follows:

`republican`	{¬ `pacifist`.}
`quaker`	{`pacifist`.}
`nixon` : `republican`, `quaker`	{ }

This $DLP^<$ program has no answer set, and therefore no conclusion is drawn.

7.5.5 Updates in Logic Programs

The definition of the semantics of updates in logic programs is another topic where $DLP^<$ could potentially be applied. Roughly, a simple formulation of the problem is the following: Given a ($\vee$-free) logic program P and a sequence $U_1, \cdots, U_n$ of successive updates (insertion/deletion of ground atoms), determine what is or is not true in the end. Expressing the insertion (deletion) of an atom A by the rule $A \leftarrow (\neg\ A \leftarrow)$, we can represent this problem by a $DLP^<$ knowledge base $\{\langle t_0, P\rangle, \langle t_1, \{U_1\}\rangle, \cdots, \langle t_n, \{U_n\}\rangle\}$ (t_i intuitively represents the instant of time when the update U_i

has been executed), where $t_n < \cdots < t_0$.[8] The answer sets of the program for t_k can be taken as the semantics of the execution of $U_1, \cdots, U_k$ on P. For instance, given the logic program $\mathtt{P} = \{\mathtt{a} \leftarrow \mathtt{b}, \mathtt{not\ c}.\}$ and the updates $\mathtt{U_1} = \{\mathtt{b}.\}$, $\mathtt{U_2} = \{\mathtt{c}.\}$, $\mathtt{U_3} = \{\neg\ \mathtt{b}.\}$, we build the $DLP^<$ program

$\mathtt{t_0}$ $\quad \{\ \mathtt{a} \leftarrow \mathtt{b}, \mathtt{c}, \mathtt{not\ d}.\ \}$
$\mathtt{t_1} : \mathtt{t_0}$ $\quad \{\ \mathtt{b}.\ \}$
$\mathtt{t_2} : \mathtt{t_1}$ $\quad \{\ \mathtt{c}.\ \}$
$\mathtt{t_3} : \mathtt{t_2}$ $\quad \{\ \neg\ \mathtt{b}.\ \}$.

The answer set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ of the program for $\mathtt{t_2}$ gives the semantics of the execution of $\mathtt{U_1}$ and $\mathtt{U_2}$ on $\mathtt{P}$; while the answer set $\{\mathtt{c}\}$ of the program for $\mathtt{t_3}$ expresses the semantics of the execution of $\mathtt{U_1}$, $\mathtt{U_2}$ and $\mathtt{U_3}$ on $\mathtt{P}$ in the given order.

The semantics of updates obtained in this way is very similar to the approach adopted for the ULL language in [79]. Further investigations are needed on this topic to see whether $DLP^<$ can represent update problems in more general settings like those treated in [90] and in [2]. A comparative analysis of various approaches to updating logic programs is presented in [53, 54], showing that the semantics of common fragments coincide.

7.6 From $DLP^<$ to Plain DLP

In this section we show how a $DLP^<$ program can be translated into an equivalent plain disjunctive logic program (with strong negation, but without inheritance). The translation allows us to exploit existing disjunctive logic programming (DLP) systems for the implementation of $DLP^<$.

Notation.

1. Let $\mathcal{P}$ be the input $DLP^<$ program.
2. We denote a literal by $\phi(\bar{X})$, where $\bar{X}$ is the tuple of the literal's arguments, and ϕ represents an *adorned predicate*, that is either a predicate symbol p or a strongly negated predicated symbol $\neg\ p$. Two adorned predicates are *complementary* if one is the negation of the other (e.g., q and $\neg\ q$ are complementary). $\neg.\phi$ denotes the complementary adorned predicate of the adorned predicate ϕ.
3. An adorned predicate ϕ is *conflicting* if both $\phi(\bar{X})$ and $\neg.\phi(\bar{Y})$ occur in the heads of rules in $\mathcal{P}$.
4. Given an object o in $\mathcal{P}$, and a head literal $\phi(\bar{X})$ of a defeasible rule in $\Sigma(o)$, we say that ϕ is *threatened in* o if a literal $\neg.\phi(\bar{Y})$ occurs in the head of a rule in $\Sigma(o')$ where $o' < o$. A defeasible rule r in $\Sigma(o)$ is *threatened in* o if all its head literals are threatened in o.

[8] In this context, $<$ should be interpreted as "more recent".

The rewriting algorithm translating $DLP^{<}$ programs in plain disjunctive logic programs with constraints is shown in Figure 7.2.

An informal description of how the algorithm proceeds is the following:

- $DLP(\mathcal{P})$ is initialized to a set of facts with head predicate $prec'$ representing the partial ordering among objects (statement 1).
- Then, for each object o in $\mathcal{O}(\mathcal{P})$:
  - For each threatened literal $\phi(\bar{X})$ appearing in o, rules defining when the literal is overridden are added (statements 3–7).
  - For each rule r belonging to o:
    1. If r is threatened, then the rule is rewritten, such that the head literals include information about the object in which they have been derived, and the body includes a literal which satisfies the rule if it is overridden. In addition, a rule is added which encodes when the rule is overridden (statements 9–12).
    2. Otherwise (i.e., if r is not threatened) just the rule head is rewritten as described above, since these rules cannot be overridden (statements 13–15).
- For all adorned predicates in the program, we add a rule which states that an atom with this predicate holds, no matter in which object it has been derived (statements 19–23). The information in which object an atom has been derived is only needed for determination of overriding.
- Finally, statements 24–28 add a constraint for each adorned predicate, which prevents the generation of inconsistent sets of literals.

$DLP(\mathcal{P})$ is referred to as the *DLP version* of the program $\mathcal{P}$.[9]
We now give an example to show how the translation works:

Example 7.6.1
The datalog version $DLP(\mathcal{P}_{10})$ of the program $\mathcal{P}_{10}$ of Example 7.1.1 is:

(**1**) **rules expliciting partial order among objects** :
$\mathtt{prec'}(\mathtt{o_2}, \mathtt{o_1})$.

(**2**) **rules for threatened adorned predicates in** o_1 :
$\mathtt{ovr'}(\mathtt{a}, \mathtt{o_1}) \leftarrow \neg\ \mathtt{a'}(\mathtt{X}), \mathtt{prec'}(\mathtt{X}, \mathtt{o_1})$.
$\mathtt{ovr'}(\neg\ \mathtt{b}, \mathtt{o_1}) \leftarrow \mathtt{b'}(\mathtt{X}), \mathtt{prec'}(\mathtt{X}, \mathtt{o_1})$.

[9] $DLP(\mathcal{P})$ is a function-free disjunctive logic program. Allowing functions could make the algorithm notation more compact, but would not give any computational benefit.

ALGORITHM
INPUT: a $DLP^{<}$-program $\mathcal{P}$
OUTPUT: a plain disjunctive logic program with constraints $DLP(\mathcal{P})$

1: $DLP(\mathcal{P}) \Leftarrow \{prec'(o, o_1) \leftarrow \ \mid o < o_1\}$
2: **for** each object $o \in \mathcal{O}(\mathcal{P})$ **do**
3: **for** each threatened adorned predicate ϕ in o **do**
4: Add the following rule to $DLP(\mathcal{P})$:
5: $ovr'(\phi, o, X_1, \ldots, X_n) \leftarrow \neg.\phi'(X, X_1, \ldots, X_n), prec'(X, o)$
6: where n is the arity of ϕ and $X, X_1, \ldots, X_n$ are distinct variables.
7: **end for**
8: **for** each rule r in $\Sigma(o)$, say $\phi_1(\bar{X}_1) \vee \ldots \vee \phi_n(\bar{X}_n) \leftarrow BODY$, **do**
9: **if** r is threatened **then**
10: Add the following two rules to $DLP(\mathcal{P})$:
11: $\phi'_1(o, \bar{X}_1) \vee \ldots \vee \phi'_n(o, \bar{X}_n) \leftarrow BODY$, `not` $ovr'(r, o, \bar{X}_1, \ldots, \bar{X}_n)$
12: $ovr'(r, o, \bar{X}_1, \ldots, \bar{X}_n) \leftarrow ovr'(\phi_1, o, \bar{X}_1), \ldots, ovr'(\phi_n, o, \bar{X}_n)$
13: **else**
14: Add the following rule to $DLP(\mathcal{P})$:
15: $\phi'_1(o, \bar{X}_1) \vee \ldots \vee \phi'_n(o, \bar{X}_n) \leftarrow BODY$
16: **end if**
17: **end for**
18: **end for**
19: **for** each adorned predicate ϕ appearing in $\mathcal{P}$ **do**
20: Add the following rule to $DLP(\mathcal{P})$:
21: $\phi(X_1, \ldots, X_n) \leftarrow \phi'(X_0, X_1, \ldots, X_n)$
22: where n is the arity of ϕ and $X_0, \ldots, X_n$ are distinct variables.
23: **end for**
24: **for** each conflicting adorned predicate ϕ appearing in $\mathcal{P}$ **do**
25: Add the following constraint to $DLP(\mathcal{P})$:
26: $\leftarrow \phi(X_1, \ldots, X_n), \neg.\phi(X_1, \ldots, X_n)$
27: where n is the arity of ϕ and $X_1, \ldots, X_n$ are distinct variables.
28: **end for**

Figure 7.2: A Rewriting Algorithm

(**3**) **rewriting of rules in** o_1 :
$\mathtt{a}'(\mathtt{o}_1) \vee \neg\, \mathtt{b}'(\mathtt{o}_1) \leftarrow \mathtt{c}, \mathtt{not\ d}, \mathtt{not\ ovr}'(\mathtt{r}_1, \mathtt{o}_1)$.
$\mathtt{ovr}'(\mathtt{r}_1, \mathtt{o}_1) \leftarrow \mathtt{ovr}'(\mathtt{a}, \mathtt{o}_1), \mathtt{ovr}'(\neg\, \mathtt{b}, \mathtt{o}_1)$.
$\mathtt{e}'(\mathtt{o}_1) \leftarrow \mathtt{b}$.

(**4**) **rewriting of rules in** o_2 :
$\neg\, \mathtt{a}'(\mathtt{o}_2) \vee \mathtt{c}'(\mathtt{o}_2)$.
$\mathtt{b}'(\mathtt{o}_2)$.
$\mathtt{c}'(\mathtt{o}_2) \leftarrow \mathtt{b}$.

(**5**) **projection rules** :
$\mathtt{a} \leftarrow \mathtt{a}'(\mathtt{X})$. $\quad \neg\, \mathtt{a} \leftarrow \neg\, \mathtt{a}'(\mathtt{X})$.
$\mathtt{b} \leftarrow \mathtt{b}'(\mathtt{X})$. $\quad \neg\, \mathtt{b} \leftarrow \neg\, \mathtt{b}'(\mathtt{X})$.
$\mathtt{c} \leftarrow \mathtt{c}'(\mathtt{X})$. $\quad \mathtt{d} \leftarrow \mathtt{d}'(\mathtt{X})$.
$\mathtt{e} \leftarrow \mathtt{e}'(\mathtt{X})$.

(**6**) **constraints** :
$\leftarrow \mathtt{a}, \neg\, \mathtt{a}$.
$\leftarrow \mathtt{b}, \neg\, \mathtt{b}$.

Given a model M for $DLP(\mathcal{P})$, $\pi(M)$ is the set of literals obtained from M by eliminating all the literals with a "primed" predicate symbol, i.e. a predicate symbol in the set $\{prec', ovr'\} \cup \{\phi' \mid \exists$ *an adorned literal* $\phi(\bar{X})$ *appearing in* $\mathcal{P}\}$. $\pi(M)$ is the set of literals without all atoms which were introduced by the translation algorithm.

The DLP version of a $DLP^<$-program $\mathcal{P}$ can be used in place of $\mathcal{P}$ in order to evaluate answer sets of $\mathcal{P}$. The result supporting the above statement is the following:

Theorem 7.6.1
Let $\mathcal{P}$ be a $DLP^<$-program. Then, for each answer set M for $\mathcal{P}$ there exists a consistent answer set M' for $DLP(\mathcal{P})$ such that $\pi(M') = M$. Moreover, for each consistent answer set M' for $DLP(\mathcal{P})$ there exists an answer set M for $\mathcal{P}$ such that $\pi(M') = M$.

Proof First we show that given an answer set M for $\mathcal{P}$ there exists a consistent answer set M' for $DLP(\mathcal{P})$ such that $\pi(M') = M$. We proceed by constructing the model M'. Let $K_1 = \{prec'(o, o_1) \mid o < o_1\}$. Let K_2 be the set of ground literals $ovr'(r, o, \bar{X})$ such that there exists a (defeasible) rule $r \in Ground(\mathcal{P})$ with $obj\_of(r) = o$ such that r is overridden in M and $\bar{X}$ is the tuple of arguments appearing in the head of r. Let K_3 be the set of ground literals $ovr'(L, o)$ such that there exist two rules $r, r' \in Ground(\mathcal{P})$ such that $L \in H(r)$, r is defeasible and r' overrides r in L. Let denote by $\mathcal{K}$ the collection of sets of ground literals such that each element $K \in \mathcal{K}$ satisfies the following properties:

1. for each literal $\phi(\bar{X}) \in M$ there is a literal $\phi'(o, \bar{X})$ in K, for some object identifier o,

2. for each $r \in G_M(\mathcal{P})$ such that the body of r is true in M, for at least one literal $\phi(\bar{X})$ of the head of r a corresponding literal $\phi'(obj\_of(r), \bar{X})$ occurs in K,

3. $K \subseteq \{\phi'(o, \bar{X}) \mid \phi(\bar{X})$ is an adorned predicate appearing in $\mathcal{P} \wedge o \in \mathcal{O}\}$,

4. K is a consistent set of literals.

First observe that the family $\mathcal{K}$ is not empty (i.e., there is at least a set of consistent sets of literals satisfying items (1), (2) and (3) above). This immediately follows from the fact that M is an answer set of the program $\mathcal{P}$.

Let $M_K = K_1 \cup K_2 \cup K_3 \cup K \cup M$, for a generic $K \in \mathcal{K}$. It is easy to show that $G_{M_K}(DLP(\mathcal{P}))$ is independent on which $K \in \mathcal{K}$ is chosen. Indeed, no literals from K appear in the NAF part of the rules in $Ground(DLP(\mathcal{P}))$.

Now we examine which rules the program $pos(G_{M_K}(DLP(\mathcal{P})))$ contains (for any set $K \in \mathcal{K}$).

Both the rules with head predicate $prec'$ and ovr' and the rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ (added by statement 21 of Figure 7.2) appear unchanged in $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, these rules do not contain a NAF part (recall that the GL transformation can modify only rules in which a NAF part occurs). Each constraint of $DLP(\mathcal{P})$ (added by statement 26 of the algorithm), that is a rule of the form $b \leftarrow \phi(\bar{X}), \neg.\phi(\bar{X}), \texttt{not}\ b$ (where b is a literal not occurring in M_K) is translated into the rule $b \leftarrow \phi(\bar{X}), \neg.\phi(\bar{X})$.

The other rules in $pos(G_{M_K}(DLP(\mathcal{P})))$ originate from rules of $Ground(DLP(\mathcal{P}))$ obtained by rewriting rules of $Ground(\mathcal{P})$ (see statements 11 and 15 of the algorithm).

Thus, consider a rule r of $Ground(\mathcal{P})$.

If r is defeasible and overridden in M then the corresponding rule in $Ground(DLP(\mathcal{P}))$ (generated by statement 11 of the algorithm) contains a NAF part not satisfied in M_K, by construction of K_2 and K_3. Hence, such a rule appears neither in $pos(G_M(\mathcal{P}))$ nor in $pos(G_{M_K}(DLP(\mathcal{P})))$.

The other case we have to consider is that the rule r is either a strict rule or a defeasible rule not overridden in M.

First suppose that r is a strict rule or is a defeasible rule not threatened in M (recall that a rule not threatened in M is certainly not overridden in M). In this case, the corresponding rule, say r' in $Ground(DLP(\mathcal{P}))$ (generated by statement 15 of the algorithm) has the same body of r and the head modified by renaming predicates (from ϕ to ϕ') and by adding the object o (from which the rule r comes) as first argument in each head literal. Since the body of r' does not contain literals from K_1, K_2, K_3 and K, and further $M \subseteq M_K$ (for each $K \in \mathcal{K}$), r' is eliminated by the GL transformation w.r.t. M_K if and only if r is eliminated by the GL transformation w.r.t. M. Moreover, in case r' is not eliminated by the GL transformation w.r.t

M_K, since the body of r' does not contain literals from K_1, K_2, K_3 and K, the GL transformation w.r.t. M_K modifies the body of r' in the same way the GL transformation w.r.t. M modifies the body of r. Thus, each rule r in $pos(G_M(\mathcal{P}))$ has a corresponding rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ with the same body and a rewritten head.

Now suppose that r is a threatened defeasible rule that is not overridden in M. In this case, the corresponding rule, say r' in $Ground(DLP(\mathcal{P}))$ (generated by statement 11 of the algorithm) has the head modified by renaming predicates (from ϕ to ϕ') and by adding the object o as first argument in each head literal and a body obtained by adding to the body of r a literal of the form $\texttt{not}\ ovr'(r, o, \bar{X})$, where o is the object from which r comes, and $\bar{X}$ represents the tuple of terms appearing in the head literals of r. Since the rule r is not overridden in M, the literal $ovr'(r, o, \bar{X})$ cannot belong to K_2 and hence cannot belong to M_K. Thus, the GL transformation w.r.t M_K eliminates the NAF part of the rule r'. As a consequence, also in this case, each rule in $pos(G_M(\mathcal{P}))$ has a corresponding rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ with

Now we prove that, for any $K \in \mathcal{K}$, M_K is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, rules with head predicate $prec'$ are clearly satisfied. Further, rules with head predicate ovr' are satisfied by construction of set K_2, K_3 and K. Moreover, rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ are satisfied since $M \subseteq M_K$ and by construction of K. Rules of $pos(G_{M_K}(DLP(\mathcal{P})))$ of the form $b \leftarrow \phi(\bar{X}), \texttt{not}\ \phi(\bar{X})$, originated by the translation of the constraints, are satisfied since M_K is a consistent set of literals.

Consider now the remaining rules (those corresponding to rules of $pos(G_M(\mathcal{P}))$). Let r be a rule of $pos(G_{M_K}(DLP(\mathcal{P})))$ and r' the rule of $pos(G_M(\mathcal{P}))$ corresponding to r. As shown earlier, the two rules have the same body. Thus, if the body of r is true w.r.t. M_K, the body of r' is true w.r.t. M, since no literal of $M_K \setminus M$ can appear in the body of the rule r' (and hence of the rule r). As a consequence, by property (2) of the collection $\mathcal{K}$ to which the set K belongs, the head of the rule r is true in M_K.

Thus, M_K is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$.

Now we prove the following claim:

Claim 1. Let $\bar{M}$ be a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Then, $\pi(\bar{M})$ is a model for $pos(G_M(\mathcal{P}))$.

Proof By contradiction suppose that $\pi(\bar{M})$ is not a model for $pos(G_M(\mathcal{P}))$. Thus, there exists a rule $r \in pos(G_M(\mathcal{P}))$ with body true in $\pi(\bar{M})$ and head false in $\pi(\bar{M})$. Since, as shown earlier, the rule r has a corresponding rule r' in $pos(G_{M_K}(DLP(\mathcal{P})))$ with the same body of r and the head obtained by replacing each literal $\phi(\bar{X})$ of the head of r by the corresponding literal $\phi'(o, \bar{X})$, where o is the object from which r comes. Since $\pi(\bar{M})$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$, at least one of the "$\phi'$ literal" of the head of r' must be true in $\pi(\bar{M})$. Then, due to the presence of the rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ in $pos(G_{M_K}(DLP(\mathcal{P})))$, $\pi(\bar{M})$ must contain also the "ϕ

corresponding literal" belonging to the head of r (contradiction) $\Box$

Moreover we prove that each model for $pos(G_{M_K}(DLP(\mathcal{P})))$

(1) contains M, and

(2) belongs to $\mathcal{K}$.

To prove item (1), suppose by contradiction $\bar{M}$ is a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ such that $M \cap \bar{M} \neq M$. Thus, $\pi(\bar{M}) \subset M$. On the other hand, by Claim 1, $\pi(\bar{M})$ is a model for $pos(G_M(\mathcal{P}))$. But since M is an answer set for $\mathcal{P}$ and then a minimal model for $pos(G_M(\mathcal{P}))$, a contradiction arises.

Now we have to prove the item (2) above. First observe that the properties 3. and 4. of the family $\mathcal{K}$ are trivially verified by the models of $pos(G_{M_K}(DLP(\mathcal{P})))$. Thus, by contradiction suppose there exists a model $\bar{M}$ of $pos(G_{M_K}(DLP(\mathcal{P})))$ such that it does not satisfy one of the properties 1. or 2. characterizing the family $\mathcal{K}$.

First suppose that property 1. is not satisfied by $\bar{M}$, that is, there is a literal $\phi(\bar{X})$ in M such that no corresponding literal $\phi'(o, \bar{X})$ occurs in $\bar{M}$, for some object identifier o. Let $\bar{M}'$ be the set obtained by $\bar{M}$ by eliminating all such literals $\phi(\bar{X})$. It is easy to see that $\bar{M}'$ is still a model for $pos(G_{M_K}(DLP(\mathcal{P})))$. Indeed, rules of the form $\phi(\bar{X}) \leftarrow \phi'(X, \bar{X})$ are satisfied since literals $\phi(\bar{X})$ dropped from $\bar{M}$ do not have corresponding "ϕ' literals" by hypothesis. Further, no other rule in $pos(G_{M_K}(DLP(\mathcal{P})))$ contains a literal from M in the head. On the other hand, by Claim 1, $\pi(M')$ is a model for $pos(G_M(\mathcal{P}))$. But this is a contradiction, since $\pi(M') \subset M$ and M is an answer set for $\mathcal{P}$.

Suppose now that property 2. is not satisfied by $\bar{M}$, that is, there is a rule $r \in pos(G_M(\mathcal{P}))$ such that the body of r is true in M and no "ϕ' literals" corresponding to literals of the head occur in $\bar{M}$. Since, $M \subseteq \bar{M}$ (see item (1) above), the corresponding rule of $pos(G_{M_K}(DLP(\mathcal{P})))$ is not satisfied. But this implies that $\bar{M}$ can not be a model for $pos(G_{M_K}(DLP(\mathcal{P})))$ (contradiction).

A consequence of the fact that every model of $pos(G_{M_K}(DLP(\mathcal{P})))$ contains M is that every model of $pos(G_{M_K}(DLP(\mathcal{P})))$ must contain the sets K_1, K_2 and K_3. because of the rules added by statements 1,5 and 12 of the algorithm.

Thus, the model $M' = M_{K'}$ for $K' \in \mathcal{K}$ such that no set $\bar{K} \in \mathcal{K}$ exists such that $\bar{K} \subset K'$ is a minimal model for $pos(G_{M_K}(DLP(\mathcal{P}))$, that is, a consistent answer set for $DLP(\mathcal{P})$. Hence, the first part of the proof is concluded, since $\pi(M') = M$.

Now, we prove that given a consistent answer set M' for $DLP(\mathcal{P})$, $M = \pi(M')$ is an answer set for $\mathcal{P}$.

First we prove that a literal $\phi(\bar{X})$ belongs to M if and only if there exits a literal $\phi'(o, \bar{X})$ in M', for some object identifier o.

Indeed, $\phi(\bar{X}) \in M$ implies that $\phi(\bar{X}) \in M'$. But since M' is a minimal model for $pos(G_{M'}(DLP(\mathcal{P}))$, there must exits a rule in $pos(G_{M'}(DLP(\mathcal{P}))$ with head containing the literal $\phi(\bar{X})$ and body true w.r.t. M' (otherwise the literal $\phi(\bar{X})$ could be dropped from M' without invalidate any rule of $pos(G_{M'}(DLP(\mathcal{P}))$ and thus M' would not be minimal). Conversely, if $\phi'(o, \bar{X}) \in M'$, for some object identifier o, the literal $\phi(\bar{X})$ belongs to M', since M' is a model for $pos(G_{M'}(DLP(\mathcal{P}))$ and the rule $\phi(\bar{X}) \leftarrow \phi'(o, \bar{X})$ belongs to $pos(G_{M'}(DLP(\mathcal{P}))$. Thus, $\phi(\bar{X}) \in M$.

Moreover, we prove that every rule of $pos(G_{M'}(DLP(\mathcal{P}))$ has a corresponding rule in $pos(G_M(\mathcal{P}))$ with same body and a head obtained by replacing the ϕ' literals with the ϕ corresponding ones and by eliminating the object argument from these literals. Indeed, from the above result, the GL transformation deletes a rule r from $Ground(DLP(\mathcal{P}))$ if either the corresponding rule belonging to $Ground(\mathcal{P})$ is overridden in M (due the the literal `not` $ovr'(r, o, \bar{X})$ occurring in the body of r) or some negated (by negation `not`) literal is false in M'. But this literal is false in M' if and only if it is false in M. On the other hand, in case the rule is not deleted by the GL transformation, its body is rewritten in the same way of the corresponding rule appearing in $pos(G_M(\mathcal{P}))$.

As a consequence, M is a model for $pos(G_M(\mathcal{P}))$. Indeed, if the body of a rule r of $pos(G_M(\mathcal{P}))$ is true w.r.t. M, the corresponding rule r' of $pos(G_{M'}(DLP(\mathcal{P}))$ has the body true w.r.t. M'. Hence, at least one of the head literals of r' must be true in M'. Let $\phi'(o, \bar{X})$ such a literal. As shown earlier, this implies that $\phi(\bar{X})$ belongs to M. But $\phi(\bar{X})$ appears in the head of r and hence r is satisfied in M.

Now we prove that M is minimal. By contradiction, suppose that $\bar{M} \subset M$ is a model for $pos(G_M(\mathcal{P}))$. Consider the literals belonging to the set $M \setminus \bar{M}$. Because of the correspondence between the rules of $pos(G_M(\mathcal{P}))$ and the rules of $pos(G_{M'}(DLP(\mathcal{P}))$, the set of literals obtained from M' by eliminating all the literals $\phi(\bar{X})$ belonging to the set $M \setminus \bar{M}$ as well as the corresponding ϕ' literals is still a model for $pos(G_{M'}(DLP(\mathcal{P}))$. But this is a contradiction, since M' is a consistent answer set for $DLP(\mathcal{P})$.

Since M is a model for $pos(G_M(\mathcal{P}))$ and is minimal, $M = \pi(M')$ is an answer set for the program $\mathcal{P}$. Hence the proof is concluded. □

Example 7.6.2
Consider the program $\mathcal{P}_{10}$ of Example 7.1.1. It is easy to see that $DLP(\mathcal{P}_{10})$ (see Example 7.6.1) admits one consistent answer set $M = \{prec'(o_2, o_1), a'(o_1), e'(o_1), b'(o_2),\ c'(o_2),\ ovr'(\neg\ b, o_1),\ a,\ e,\ b,\ c\}$. Thus, $\pi(M) = \{a, b, c, e\}$. On the other hand, $\pi(M)$ is the only answer set for $\mathcal{P}_{10}$, as shown in Example 7.2.5.
□

Chapter 8

Diagnosis

In this chapter, we present the Diagnosis Front-end of the DLV system. Diagnostic reasoning deals with the problems of finding explanations for some observed behavior of a system, given a theory or a model of this system and some hypotheses, which can constitute these explanations. Usually, but not necessarily, the observed behavior is some failure.

The chapter is organized as follows. To begin with, we describe in Sections 8.1 and 8.2 the different formal models of diagnosis that are realized in the front-end. It supports both abductive diagnosis [105, 29], adapted to the semantics of logic programming [74, 55], and consistency-based diagnosis [110, 34].

In our abductive model of diagnosis (Section 8.1), the theory is a disjunctive logic program, whose semantics is given by the set of its stable models or answer sets [68]. This form of diagnostic reasoning has been recently proved to be highly expressive, as it is able to represent even problems located at the third level of the polynomial hierarchy [55]. On the other hand, the consistency-based diagnosis model (Section 8.2) refers to theories of first-order logic under classical semantics, similarly to the original definition by Reiter [110]. For each of the two diagnosis modalities (abduction and consistency-based), general diagnoses, single-error diagnoses and subset minimal diagnoses are considered in our model. We then illustrate the various kinds of diagnostic reasoning supported in the DLV system by a number of examples, which demonstrate the power and the simplicity of our model of diagnosis.

In Section 8.3, we present six translation algorithms (one for each kind of diagnostic reasoning problem emerging from the discussion above) which rewrite diagnosis problems into disjunctive logic programs (DLPs), such that each answer set of such a program corresponds to precisely one diagnosis, and each diagnosis has some corresponding answer set. In particular, for each answer set a unique corresponding diagnosis exists. On the other hand, for each diagnosis a corresponding answer set of the program exists. Running

such a program with DLV, we obtain all diagnoses of the input diagnosis problem.

8.1 Abductive Diagnosis over DLP Theories

Abductive diagnostic reasoning has been widely studied in the literature in the last years (see, e.g., [105, 103, 75, 29, 74, 55]).

Intuitively, a diagnostic problem consists of a description of the system to be diagnosed, observations of the actual state of the system, and possible reasons for effects, which are usually errors.

Definition 8.1.1
We define an abductive diagnostic problem (ADP) $\mathcal{P}$ *as a triple* $\langle H, T, O \rangle$, *where:*

- H *is a set of ground classical literals referred to as* hypotheses.
- T *is a Datalog program as described in Section 2.1 and is referred to as* theory.
- O *is a set of ground NAF literals and is referred to as* observations.

The solutions to an ADP are referred to as abductive diagnoses. In the following subsections, we describe several notions of reasoning over abductive diagnostic problems.

8.1.1 General Abductive Diagnosis

In the case of finding general diagnoses, any subset of the hypotheses which "explains" the observations based on the system description is considered a diagnosis. By an explanation in the abductive case we understand that the diagnosis together with the system model gives reason for the observations.

Definition 8.1.2
Given an ADP $\mathcal{P} = \langle H, T, O \rangle$, *a* general diagnosis *is a set* $\Delta \subseteq H$, *such that* $T \cup \Delta \models_b q_O$ *holds, where* q_O *is* $\mathtt{o_1}, \ldots, \mathtt{o_n}$? *if* $O = \{o_1, \ldots, o_n\}$.

Note that our definition incorporates the brave reasoning consequence operator. In the literature some definitions are based on cautious reasoning, i.e., O must be true in *all* (stable) models (answer sets) of $T \cup \Delta$. See for example [55] for a variety of definitions.

Our first example shows how to use disjunction in the system description:

Example 8.1.1
Consider an electric circuit, as shown in Fig. 8.1, consisting of a simple stove with two hot-plates wired in parallel and a control light, which is on if at

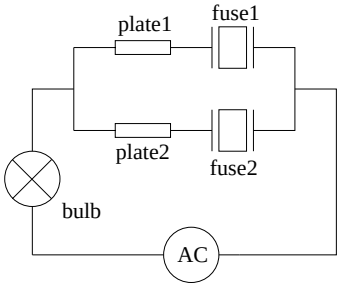

Figure 8.1: Simplified circuit diagram of a stove.

least one of the plates is in operation. Each plate has a fuse, and it is known that one of them cannot stand much current and will melt if the current gets high, but it is not known which.

We assume that there are several equally likely causes of malfunction: The power supply may fail (in this case $-$`power` *holds), the bulb may be broken (*`broken_bulb` *holds), or the current may be (unusually) high (*`high_curr` *holds). So our hypotheses are:*

$$H_{st} = \{\texttt{broken\_bulb}, -\texttt{power}, \texttt{high\_curr}\}$$

Now for the system description T_{stove}*:*

$$\begin{aligned}
&-\texttt{light} \leftarrow -\texttt{power}.\\
&-\texttt{light} \leftarrow \texttt{broken_bulb}.\\
&-\texttt{light} \leftarrow \texttt{melted_fuse}_1, \texttt{melted_fuse}_2.\\
&\texttt{melted_fuse}_1 \vee \texttt{melted_fuse}_2 \leftarrow \texttt{high_curr}.\\
&\texttt{hot_plate}_1 \leftarrow \texttt{not melted_fuse}_1, \texttt{not } -\texttt{power}.\\
&\texttt{hot_plate}_2 \leftarrow \texttt{not melted_fuse}_2, \texttt{not } -\texttt{power}.
\end{aligned}$$

The first three rules describe situations under which the control light (emitted by the bulb) is off, namely, if the power fails, the bulb is broken, or both fuses are melted.

The fourth rule states that, on high current, a fuse will melt.

The last rules state that a hot-plate is heated if there is no power failure and the fuse is not melted. [1]

We observe that no light is emitted and that plate 1 is not hot:

$$O_{st} = \{-\texttt{light}, \texttt{not hot\_plate}_1\}$$

The goal is to find diagnoses to the ADP $\mathcal{P}_{st} = \langle H_{st}, T_{st}, O_{st}\rangle$*, which explain the current situation.*

[1] To keep the example simple, we refrain from modeling switches for the hot-plates.

Consider now $\Delta_1 = \{-\texttt{power}\}$. *Then* $\Delta_1 \cup T_{st}$ *has one answer set:*

$$A_1 = \{-\texttt{power}, -\texttt{light}\}$$

$-\texttt{light}$ *is in* A_1, *and* $\texttt{hot\_plate}_1$ *is not, so* Δ_1 *is a valid diagnosis.*

Let us examine whether $\Delta_2 = \{\texttt{broken\_bulb}\}$ *is a valid diagnosis.* $\Delta_2 \cup T_{st}$ *also has only one answer set:*

$$\begin{aligned} A_2 = \{&\texttt{broken\_bulb}, -\texttt{light}, \\ &\texttt{hot\_plate}_1, \texttt{hot\_plate}_2\} \end{aligned}$$

So Δ_2 *is not a valid diagnosis, since there is no answer set of* $\Delta_2 \cup T_{st}$ *in which* $\texttt{hot\_plate}_1$ *is not contained.*

On the other hand, if we augment Δ_2 *to*

$$\Delta_3 = \{\texttt{broken\_bulb}, \texttt{high\_curr}\}$$

the program $\Delta_3 \cup T_{st}$ *has two answer sets:*

$$\begin{aligned} A_3 = \Delta_3 \cup \{&-\texttt{light}, \texttt{melted\_fuse}_1, \\ &\texttt{hot\_plate}_2\} \\ A_4 = \Delta_3 \cup \{&-\texttt{light}, \texttt{melted\_fuse}_2, \\ &\texttt{hot\_plate}_1\} \end{aligned}$$

One answer set (A_3) *exists in which* $-\texttt{light}$ *is contained, but* $\texttt{hot\_plate}_1$ *is not, therefore* Δ_3 *is a valid diagnosis. Note that the observations do not hold in* A_4, *albeit it is sufficient that one answer set explains the observations since we consider brave reasoning.*

Other diagnoses for $\mathcal{P}_{st}$ *are:*

$$\begin{aligned} &\{-\texttt{power}, \texttt{high\_curr}\} \\ &\{-\texttt{power}, \texttt{broken\_bulb}\} \\ &\{-\texttt{power}, \texttt{high\_curr}, \texttt{broken\_bulb}\} \end{aligned}$$

The system description of the next example utilizes the notion of reachability which is easily defined in logic programming (through a recursive predicate) while it cannot be defined at all in classical first order logic.

Example 8.1.2
Consider a computer network as depicted in Fig. 8.2 (the nodes denote computers and the arcs represent point-to-point connections).

The network is described by facts of the form $\texttt{connected}(\texttt{x}, \texttt{y})$, *which state that there is a connection between node* x *and node* y. *The main part of the system description is the notion of* reachability*: A node is reachable from another node if there is a path between them and all computers on that path are online (*$\texttt{not off}(\texttt{X})$*).*

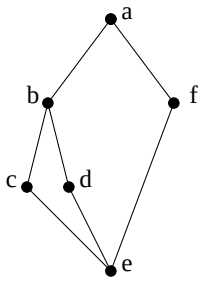

Figure 8.2: Topology of a point-to-point network.

$$\begin{aligned}
\texttt{reaches}(\texttt{X},\texttt{X}) \leftarrow\ & \texttt{node}(\texttt{X}), \texttt{not}\ \texttt{off}(\texttt{X}) \\
\texttt{reaches}(\texttt{X},\texttt{Z}) \leftarrow\ & \texttt{reaches}(\texttt{X},\texttt{Y}), \\
& \texttt{connected}(\texttt{Y},\texttt{Z}), \\
& \texttt{not}\ \texttt{off}(\texttt{Z})
\end{aligned}$$

The system description T_{net} consists of the facts of `connected` *and the above rules for* `reaches`.

The hypotheses are given by the assumptions that particular machines are offline:

$$H_{net} = \{\texttt{off}(\texttt{a}), \texttt{off}(\texttt{b}), \texttt{off}(\texttt{c}), \texttt{off}(\texttt{d}), \texttt{off}(\texttt{e}), \texttt{off}(\texttt{f})\}$$

If we work on computer `a` *and fail to reach* `e`*, the observations are given as follows:*

$$O_{net} = \{\texttt{not}\ \texttt{off}(\texttt{a}), \texttt{not}\ \texttt{reaches}(\texttt{a},\texttt{e})\}$$

Valid diagnoses are:

{off(e)}
{off(e), off(b)}
{off(e), off(c)}
{off(e), off(d)}
{off(e), off(f)}
{off(e), off(b), off(c)}
{off(e), off(b), off(d)}
{off(e), off(b), off(f)}
{off(e), off(b), off(c), off(d)}
{off(e), off(b), off(c), off(f)}
{off(e), off(b), off(d), off(f)}
{off(e), off(b), off(c), off(d), off(f)}
{off(e), off(c), off(d)}
{off(e), off(c), off(f)}

{off(e), off(c), off(d), off(f)}
{off(e), off(d), off(f)}
{off(b), off(f)}
{off(b), off(f), off(c)}
{off(b), off(f), off(d)}
{off(b), off(f), off(c) off(d)}
{off(c), off(d), off(f)}

Example 8.1.2 shows that in general many diagnoses are in a sense redundant: If one error is enough to explain the observations, one may additionally assume several other hypotheses and still be able to explain the observations, but there is no information gained. So in general it is preferable to require some additional minimality criterion.

8.1.2 Single Error Abductive Diagnosis

A simple, but commonly used minimality criterion is to require that diagnoses are made up of exactly one hypothesis each.

Definition 8.1.3
Given an ADP $\mathcal{P} = \langle H, T, O \rangle$, a general diagnosis Δ of $\mathcal{P}$ is a single error *diagnosis, if $|\Delta| = 1$ holds.*

Example 8.1.3
Reconsider Example 8.1.1:
The only single error diagnosis for this example is $\Delta_1 = \{-\texttt{power}\}$.
Now reconsider Example 8.1.2:
The only single error diagnosis for this example is {off(e)}.

At least in the case of Example 8.1.2 it is obvious that with the single error assumption some potentially important diagnoses are missed (those which do not follow directly from a single error diagnosis by assuming additional errors).

Even worse, in many cases (for instance if the observations include not off(e)) a single error diagnosis does not exist.

8.1.3 Subset Minimal Abductive Diagnosis

A more sensible version of minimality criterion is to require that only those diagnoses should be considered, which do not contain any subset that is a diagnosis itself.

Definition 8.1.4
Given an ADP $\mathcal{P} = \langle H, T, O \rangle$, a general diagnosis Δ of $\mathcal{P}$ is a subset minimal *diagnosis, if for every diagnosis Δ^* of $\mathcal{P}$ $\Delta^* \not\subset \Delta$ holds.*

Example 8.1.4
If we reconsider Example 8.1.1, we see that there are two subset minimal diagnoses: $\{-\texttt{power}\}$ *and* $\{\texttt{broken\_bulb}, \texttt{high\_curr}\}$.

Now consider Example 8.1.2: There are three subset minimal diagnoses:

$$\{\texttt{off(e)}\},$$
$$\{\texttt{off(b)}, \texttt{off(f)}\}$$
$$\{\texttt{off(c)}, \texttt{off(d)}, \texttt{off(f)}\}$$

8.2 Consistency-Based Diagnosis

Consistency-based diagnosis has been defined e.g. in [110, 34].

As in the abductive case, we first introduce the general problem formulation and in the following subsections we define several notions of consistency-based diagnosis.

Definition 8.2.1
A consistency-based diagnostic problem (CDP) $\mathcal{P}$ *is a triple* $\langle H, T, O\rangle$*, where:*

- H *is a set of ground atoms with predicate name ab (standing for* abnormal*) and is referred to as* hypotheses*.*
- T *is a set of first-order sentences and is referred to as* theory*.*
- O *is a set of ground NAF literals and is referred to as* observations*.*

In this work, only Herbrand interpretations (resp. models) and function-free theories where sentences are universally quantified are considered. Further, we assume that these sentences are written in clausal form so that they can be represented by Datalog programs.

H consists of atoms of the form $\texttt{ab(c)}$, meaning that the component `c` behaves abnormally.

8.2.1 General Consistency-Based Diagnosis

Definition 8.2.2
Given a CDP $\mathcal{P} = \langle H, T, O\rangle$*, a* general diagnosis *is a set* $\Delta \subseteq H$*, such that* $T \cup O \cup \Delta \cup \{\neg h | h \in H \setminus \Delta\}$ *is consistent in the classical sense, i.e. it has a (two-valued) model.*

Note that the consistency-based approach requires a weaker property on the diagnoses compared to the abductive approach. Indeed, Definition 8.2.2 requires that $T \cup O \cup \Delta \cup \{\neg h | h \in H \setminus \Delta\}$ is consistent in the classical sense (i.e., it has a model), while Definition 8.1.2 requires the existence of a *stable* model (answer set) of $T \cup \Delta$ which entails O. Thus, every diagnosis of

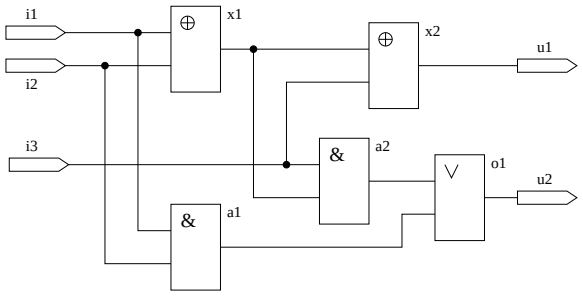

Figure 8.3: Circuit diagram of a full adder

an ADP $\mathcal{P} = \langle H, T, O \rangle$ is also a diagnosis of the corresponding consistency-based diagnosis problem (provided that hypotheses do not occur in the heads of the rules), while the converse does not hold in general.

For comparisons between the abductive and consistency-based approaches, see [106, 77, 30].

Example 8.2.1
Consider the circuit description of a full adder (depicted in Fig. 8.3)[2]*:*

For the system description T_{add}, we first give definitions for the gates involved.

The and *gates:*

$$\begin{aligned}
&\texttt{out(X,1)} \leftarrow \texttt{and(X)}, \texttt{in1(X,1)}, \texttt{in2(X,1)}, \texttt{not ab(X)}.\\
&\texttt{out(X,0)} \leftarrow \texttt{and(X)}, \texttt{in1(X,0)}, \texttt{not ab(X)}.\\
&\texttt{out(X,0)} \leftarrow \texttt{and(X)}, \texttt{in2(X,0)}, \texttt{not ab(X)}.
\end{aligned}$$

The or *gates:*

$$\begin{aligned}
&\texttt{out(X,0)} \leftarrow \texttt{or(X)}, \texttt{in1(X,0)}, \texttt{in2(X,0)}, \texttt{not ab(X)}.\\
&\texttt{out(X,1)} \leftarrow \texttt{or(X)}, \texttt{in1(X,1)}, \texttt{not ab(X)}.\\
&\texttt{out(X,1)} \leftarrow \texttt{or(X)}, \texttt{in2(X,1)}, \texttt{not ab(X)}.
\end{aligned}$$

The xor *gates:*

$$\begin{aligned}
&\texttt{out(X,1)} \leftarrow \texttt{xor(X)}, \texttt{in1(X,1)}, \texttt{in2(X,0)}, \texttt{not ab(X)}.\\
&\texttt{out(X,1)} \leftarrow \texttt{xor(X)}, \texttt{in1(X,0)}, \texttt{in2(X,1)}, \texttt{not ab(X)}.\\
&\texttt{out(X,0)} \leftarrow \texttt{xor(X)}, \texttt{in1(X,Y)}, \texttt{in2(X,Y)}, \texttt{not ab(X)}.
\end{aligned}$$

Then we specify that all in- and outputs are binary-valued:

[2]This example is taken from [110].

$$\texttt{gate(X)} \leftarrow \texttt{xor(X)}.$$
$$\texttt{gate(X)} \leftarrow \texttt{and(X)}.$$
$$\texttt{gate(X)} \leftarrow \texttt{or(X)}.$$

$$\texttt{in1(X,0)} \vee \texttt{in1(X,1)} \leftarrow \texttt{gate(X)}.$$
$$\texttt{in2(X,0)} \vee \texttt{in2(X,1)} \leftarrow \texttt{gate(X)}.$$
$$\texttt{out(X,0)} \vee \texttt{out(X,1)} \leftarrow \texttt{gate(X)}.$$

Also, we ensure that an input or output must have only one associated truth value:

$$\leftarrow \texttt{out(X,0)}, \texttt{out(X,1)}.$$
$$\leftarrow \texttt{in1(X,0)}, \texttt{in1(X,1)}.$$
$$\leftarrow \texttt{in2(X,0)}, \texttt{in2(X,1)}.$$

After these general definitions, we model the circuit wiring:

$$\texttt{in1(a1,S)} \leftarrow \texttt{in1(x1,S)}.$$
$$\texttt{in1(x1,S)} \leftarrow \texttt{in1(a1,S)}.$$

$$\texttt{in2(a1,S)} \leftarrow \texttt{in2(x1,S)}.$$
$$\texttt{in2(x1,S)} \leftarrow \texttt{in2(a1,S)}.$$

$$\texttt{in2(x2,S)} \leftarrow \texttt{in1(a2,S)}.$$
$$\texttt{in1(a2,S)} \leftarrow \texttt{in2(x2,S)}.$$

$$\texttt{out(x1,S)} \leftarrow \texttt{in1(x2,S)}.$$
$$\texttt{in1(x2,S)} \leftarrow \texttt{out(x1,S)}.$$

$$\texttt{out(x1,S)} \leftarrow \texttt{in2(a2,S)}.$$
$$\texttt{in2(a2,S)} \leftarrow \texttt{out(x1,S)}.$$

$$\texttt{out(a1,S)} \leftarrow \texttt{in2(o1,S)}.$$
$$\texttt{in2(o1,S)} \leftarrow \texttt{out(a1,S)}.$$

$$\texttt{out(a2,S)} \leftarrow \texttt{in1(o1,S)}.$$
$$\texttt{in1(o1,S)} \leftarrow \texttt{out(a2,S)}.$$

*The hypotheses H_{add} are that each of the gates is abnormal (*ab*):*

$$H_{add} = \{\texttt{ab(x1)}, \texttt{ab(x2)}, \texttt{ab(a1)}, \texttt{ab(a2)}, \texttt{ab(o1)}\}$$

It is observed that for the input $\texttt{i1} : 1, \texttt{i2} : 0, \texttt{i3} : 1$ *the output is* $\texttt{u1} : 1, \texttt{u2} : 0$.[3]

[3]To save space, we did not model the input and output elements separately. Instead, we use the uppermost connections to each input and output in Fig. 8.3.

$$O_{add} = \{\texttt{in1(x1, 1)}, \texttt{in2(x1, 0)}, \texttt{in1(a2, 1)}, \texttt{out(x2, 1)}, \texttt{out(o1, 0)}\}$$

The smallest general consistency-based diagnoses of the problem $\mathcal{P}_{add} = \langle H_{add}, T_{add}, O_{add} \rangle$ *are*

$$\Delta_1 = \{\texttt{ab(x1)}\}$$
$$\Delta_2 = \{\texttt{ab(x2)}, \texttt{ab(o1)}\}$$
$$\Delta_3 = \{\texttt{ab(x2)}, \texttt{ab(a2)}\}$$

These are in fact the subset minimal diagnoses; for general diagnoses, an addition of any number of hypotheses to the diagnoses above still yields a diagnosis.

Note that an abnormal component does not necessarily need to exhibit behavior which is the exact complement of its specified behavior, since we do not model any fault modes. So assuming one additional component to be faulty does not necessarily mean that its output is reversed. If this was true, there would be fewer additional diagnoses.

The full set of all general diagnoses can be described as:

$$\{\Delta \cup X \mid \Delta \in \{\Delta_1, \Delta_2, \Delta_3\}, X \subseteq H\}$$

Again it is clear that many diagnoses are irrelevant. Indeed, the original definition in [110] includes a (subset) minimality criterion.

8.2.2 Single Error Consistency-Based Diagnosis

As in the abductive case, it is often feasible to look for single error diagnoses, i.e. diagnoses which label exactly one component as abnormal.

Definition 8.2.3
Given a CDP $\mathcal{P} = \langle H, T, O \rangle$*, a general diagnosis* Δ *of* $\mathcal{P}$ *is a* single error *diagnosis, if* $|\Delta| = 1$ *holds.*

Example 8.2.2
Reconsider Example 8.2.1: The only single error diagnosis for $\mathcal{P}_{add}$ *is* $\{\texttt{ab(x1)}\}$.

8.2.3 Subset Minimal Consistency-Based Diagnosis

In the field of consistency-based diagnostic reasoning, most definitions of diagnoses include a subset minimality criterion.

Definition 8.2.4
Given a CDP $\mathcal{P} = \langle H, T, O \rangle$*, a general diagnosis* Δ *of* $\mathcal{P}$ *is a* subset minimal *diagnosis, if for every diagnosis* Δ^* *of* $\mathcal{P}$ $\Delta^* \not\subset \Delta$ *holds.*

Example 8.2.3
As already stated in Example 8.2.1, the subset minimal diagnoses for $\mathcal{P}_{add}$ *are* Δ_1*,* Δ_2 *and* Δ_3.

8.3 From Diagnoses to DLP Problems

In this section, we provide several algorithms which transform diagnostic problems (as defined in Section 8.1) into Datalog programs (as defined in Section 2.1).

Here we only show algorithms for transforming abductive diagnostic problems; the transformation algorithms for consistency-based diagnostic problems use similar techniques and are reported in Appendix B.

We remark that, in some cases, we impose syntactic restrictions on the diagnosis theory in order to achieve better performance in the implementation resp. due to complexity issues (i.e., cases which are harder than Σ_2^P and thus cannot be expressed by Datalog).

A different approach to the transformation of abductive logic programs into disjunctive logic programs can be found in [73]. Sakama and Inoue describe in [111] a transformation from abductive logic programs to disjunctive logic programs under the possible model semantics, which exploits a similar idea as our transformation in the general case. To our knowledge, other translations mapping subset minimal diagnoses to answer sets of disjunctive logic programs have not been discussed elsewhere.

The basic abductive translation is performed by the function in Fig. 8.4. A special atom $\_sol(i)$ is used to represent membership of hypothesis h_i in a solution. Rules connecting $\_sol(i)$ to h_i and constraints for the observations are added to $\mathcal{P}$.

Description of Algorithm 1: Intuitively, if $\_sol(i)$ is included in an answer set, the i-th hypothesis is assumed to hold (i.e., it is part of the diagnosis), whereas if $\_notsol(i)$ is included, the i-th hypothesis is not part of the diagnosis. Due to the minimality of answer sets and because $\_sol$ and $\_notsol$ do not occur in other heads, it is impossible that both $\_sol(i)$ and $\_notsol(i)$ occur in one answer set.

The rules in instruction (4) of the function in Fig. 8.4 generate a correspondence between the atoms $\_sol(i)$[4] and the hypotheses h_i. If $\_sol(i)$ belongs to an answer set of the program, also h_i will be deduced.

The constraints generated by instructions (8) and (10) of the function in Fig. 8.4 ensure that any solution Δ entails, through the initial theory T, the observations (recall that Δ is an abductive diagnosis if $\Delta \cup T \models_b q_O$). □

General abductive diagnoses are then computed through the program which is generated by Algorithm 1. More formally:

[4]Predicate names starting with an underscore are reserved for DLV internal usage and cannot be used by the regular user.

Function $AbductionToDLP(\mathcal{P}) : DLP$
Input: An Abductive Problem $\mathcal{P} = \langle H, T, O \rangle$.
Output: A Datalog program.

```
var  i, j: Integer;
     Q : DLP;
begin
(1)   Q := T;
(2)   Let H = ⟨h_1, ..., h_n⟩;
(3)   for i := 1 to n do
(4)       Q := Q ∪ {h_i ← _sol(i).};
(5)   Let O = {o_1, ..., o_m};
(6)   for j := 1 to m do
(7)       if o_j is a positive literal "a" then
(8)           Q := Q ∪ {← not a.};
(9)       else (* o_j is a negative literal "not a" *)
(10)          Q := Q ∪ {← a.};
(11)  return Q;
end
```

Figure 8.4: Basic Abductive Diagnosis Translation

Algorithm 1 General Abductive Diagnosis

Function $GeneralAbductionToDLP(\mathcal{P}) : DLP$
Input: An Abductive Problem $\mathcal{P} = \langle H, T, O \rangle$.
Output: A Datalog program.

```
var  Q : DLP;
begin
(1)   Q := AbductionToDLP(P);
(2)   Let H = ⟨h_1, ..., h_n⟩;
(3)   for i := 1 to n do
(4)       Q := Q ∪ {_sol(i) ∨ _notsol(i).};
(5)   return Q;
end
```

Function $AnswerSetToDiagnosis(A) : set\ of\ atoms$
Input: An answer set A of a program transformed from
a diagnostic problem $\mathcal{P} = \langle H, T, O \rangle$
Output: A set of atoms (the corresponding diagnosis).

var $S : setofatoms$;
begin
(1) $S := \emptyset$;
(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) **for** $i := 1$ **to** n **do**
(4) **if** $\_sol(i) \in A$ **then**
(5) $S := S \cup \{h_i\}$;
(6) **return** $\mathcal{Q}$;
end

Figure 8.5: Translating answer sets to diagnoses

Theorem 8.3.1
Let $\mathcal{P} = \langle H, T, O \rangle$ be an abductive problem and $DLP(\mathcal{P})$ the logic program generated by Algorithm 1 with input $\mathcal{P}$. Each answer set of $DLP(\mathcal{P})$ corresponds to precisely one diagnosis of $\mathcal{P}$, and each diagnosis of $\mathcal{P}$ has some corresponding answer set of $DLP(\mathcal{P})$. In particular, if A is a answer set of $DLP(\mathcal{P})$, then $AnswerSetToDiagnosis(A)$ (see Fig. 8.5) is a diagnosis *for $\mathcal{P}$.*
□

Proof The rules from instruction (4) of Algorithm 1 generate all possible diagnosis candidates by assuming either $\_sol(i)$ or $\_notsol(i)$ for each hypothesis h_i. These candidates are then checked by the rules from Fig. 8.4 as described above. □ □

The program for computing a single error diagnosis is generated by Algorithm 2. It is similar to the previous case but introduces a different choice rule.

Theorem 8.3.2
Let $\mathcal{P} = \langle H, T, O \rangle$ be an abductive problem and $DLP(\mathcal{P})$ the logic program generated by Algorithm 2 with input $\mathcal{P}$. Each answer set of $DLP(\mathcal{P})$ corresponds to precisely one single error diagnosis of $\mathcal{P}$, and each single error diagnosis of $\mathcal{P}$ has some corresponding answer set of $DLP(\mathcal{P})$. In particular, if A is an answer set of $DLP(\mathcal{P})$, then $AnswerSetToDiagnosis(A)$ (see Fig. 8.5) is a single error diagnosis *for $\mathcal{P}$.*
□

Proof This case is very similar to the general one (Algorithm 1 and

Algorithm 2 Single Error Abductive Diagnosis

Function $SingleErrorAbductionToDLP(\mathcal{P}) : DLP$

Input: An Abductive Problem $\mathcal{P} = \langle H, T, O \rangle$.
Output: A Datalog program.

var $\mathcal{Q} : DLP$;
begin
(1) $\mathcal{Q} := AbductionToDLP(\mathcal{P})$;
(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) $\mathcal{Q} := \mathcal{Q} \cup \{\_sol(1) \vee \ldots \vee \_sol(n).\}$;
(4) **return** $\mathcal{Q}$;
end

Theorem 8.3.1), the only difference being the generation of the "guess": Instead of evaluating all possible combinations of hypotheses (including the empty set), by virtue of instruction (3) in Algorithm 2 only those instances are generated (and checked) where exactly one hypothesis is assumed.

Indeed, by the minimality of answer sets, one and only one of the $\_sol(i)$ – which do not appear elsewhere in the head – can be made true. □ □

Algorithm 3 Subset Minimal Abductive Diagnosis

Function $SubsetMinimalAbductionToDLP(\mathcal{P})$
: DLP

Input: An Abductive Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive and non-disjunctive.
Output: A Datalog program.

var $\mathcal{Q} : DLP$;
begin
(1) $\mathcal{Q} := GeneralAbductionToDLP(\mathcal{P})$;
(2) $\mathcal{Q} := \mathcal{Q} \cup AbductiveSubsetMinimality(\mathcal{P})$;
(3) **return** $\mathcal{Q}$;
end

The program for computing subset minimal abductive diagnoses is generated by Algorithm 3. Basically, it consists of a part which generates a solution, and a part which checks for minimality. This part is produced by the function in Fig. 8.6, which is the most complicated function considered here.

Note that the theory is restricted to be non-disjunctive and nonnegative

Function *AbductiveSubsetMinimality*($\mathcal{P}$) : DLP
Input: An Abductive Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive and non-disjunctive.
Output: A Datalog program.

```
var  i, j: Integer;
     Q : DLP;
begin
(1)   Q := ∅;
(2)   Let H = ⟨h_1, ..., h_n⟩;
(3)   for i := 1 to n do
(4)       Q := Q ∪ {_hyp(i).};
(5)   Q := Q ∪ {_sol1(K, I) ← _sol(K), _hyp(I), <> (K, I).};
(6)   for i := 1 to n do
(7)       Q := Q ∪ {__h_i(x̄, I) ← _sol1(i, I).};
(8)   for each r : a(x̄) ← b_1(x̄_1), ..., b_n(x̄_n). ∈ T do
(9)       Q := Q ∪ {__a(x̄, I) ← _hyp(I), __b_1(x̄_1, I), ..., __b_n(x̄_n, I).};
(10)  Let O = {o_1, ..., o_m};
(11)  for j := 1 to m do
(12)      Q := Q ∪ {_notobs(I) ← _hyp(I), not __o_j(x̄, I).};
(13)  Q := Q ∪ {_isntsol(0).};
(14)  Q := Q ∪ {_isntsol(I) ← _isntsol(K), #succ(K, I), _notobs(I).};
(15)  Q := Q ∪ {_isntsol(I) ← _isntsol(K), #succ(K, I), not _sol(I).};
(16)  Q := Q ∪ {← not _isntsol(n)..};
(17)  return Q;
end;
```

Figure 8.6: Subset Minimality Check for Abductive Diagnosis

because the full subset minimal abductive diagnosis problem over Datalog theories is in the complexity class Σ_3^P, and thus not expressible in Datalog (cf. [55]). Subset minimal abductive diagnosis over Datalog programs would be expressible, but is not considered in order to keep the algorithm compact.

Theorem 8.3.3
Let $\mathcal{P} = \langle H, T, O \rangle$ be an abductive problem and $DLP(\mathcal{P})$ the logic program generated by Algorithm 3 with input $\mathcal{P}$. Each answer set of $DLP(\mathcal{P})$ corresponds to precisely one subset minimal diagnosis of $\mathcal{P}$, and each subset minimal diagnosis of $\mathcal{P}$ has some corresponding answer set of $DLP(\mathcal{P})$. In particular, if A is an answer set of $DLP(\mathcal{P})$, then $AnswerSetToDiagnosis(A)$ (see Fig. 8.5) is a subset minimal diagnosis *for $\mathcal{P}$.*
□

Proof

First of all note that $<>$ and $\#succ$ are built-in predicates directly supported by DLV, where $<>$ stands for inequality and $\#succ(A, B)$ denotes $A + 1 = B$, i.e., "B is the successor of A".

To compute minimal diagnoses, we extend the general case (Algorithm 1) with an additional minimality check that is performed by a set of rules generated by the function *AbductiveSubsetMinimality* (depicted in Fig. 8.6).

Basically, this minimality check works as follows: Given a diagnosis Δ (which is a subset of the hypotheses satisfying the constraints) with n elements, all the subsets of Δ having $n - 1$ elements are generated. If one of these subsets itself is a diagnosis, then Δ is rejected.

The rules from instruction (4) of the function in Fig. 8.6 generate a set of facts $\_hyp(1), \ldots, \_hyp(n)$ that will be used to enumerate all hypotheses later on.

The rule from instruction (5) generates the subsets of Δ as follows: For any fixed index I $(1 \leq I \leq n)$, $\Delta_I = \{\_sol1(K, I) : h_K \in \Delta, K \neq I\}$ represents the set of all elements of Δ except the one corresponding to the I-th hypothesis h_I. In other words, $\_sol1(K, I)$ is true in some answer set A if and only if $\_sol(K)$ is true in A and $I \neq K$.

For each of these subsets Δ_I of Δ, we need to check whether it is a diagnosis, that is, whether Δ_I entails the observations. If so, we can reject Δ because it is not minimal. By means of instructions (8) and (9) we generate a new version of the theory T parameterized with respect to the index I of Δ_I.

The rules generated by instruction (7) install the correspondence between the elements of Δ_I (stored in $\_sol1(K, I)$) and the hypotheses parameterized with respect to I (i.e., $h(\overline{x})$ becomes $\_\_h(\overline{x}, I)$, where $\overline{x}$ stands for the parameter list of h.)

The rules from instruction (12) check whether the observations are really entailed by Δ_I by means of the parameterized copy of the theory T that has

been generated by instructions (8) and (9). If an observation is not entailed by Δ_I, then $\_notobs(I)$ becomes true. This means that Δ_I is not a diagnosis and it thus does not invalidate the minimality of Δ.

If all the subsets Δ_I of Δ verify this condition (that is, $\_notobs(I)$ is true for every I) then Δ is indeed a minimal diagnosis, otherwise it is rejected.

This final check is performed by the rules from instructions (13), (14), (15) and (16): If Δ is a minimal solution, then for each $K \in \{1, \ldots, n\}$, it must hold that either (*) $h_K \in \Delta$ and Δ_K is not a solution, or (**) $h_K \notin \Delta$. Thus, for each K, $\_isnotsol(K)$ must be true.

The conditions (*) and (**) are expressed by the rules (14) and (15), respectively. They process the hypotheses along their ordering, and are only applicable if the checks for all preceeding hypotheses were successful. Rule (13) is the initialization, and rule (16) enforces that the check is true for all hypotheses. □ □

Chapter 9

Further Front-Ends

A number of additional front-ends for DLV have been developed, and we give here only a brief account on some of them.

9.1 Planning Front-End

The DLV $^{\mathcal{K}}$ Planning System is a knowledge based planning system. Its language is $\mathcal{K}$, which is similar in spirit to the language $\mathcal{C}$ [71, 86, 84]. $\mathcal{K}$ offers the following distinguishing features:

- handling of incomplete knowledge
- parallel actions
- nondeterministic effects
- secure (conformant) planning

We have implemented an operational prototype supporting the $\mathcal{K}$ language as a front-end on top of the DLV system [60, 45]. This front-end transforms $\mathcal{K}$ files, the syntax of which is described in the following, and (optionally) also background knowledge in the form of stratified datalog programs to answer set programs such that the answer sets of the transformed programs correspond to $\mathcal{K}$ plans.

A detailed account of the language $\mathcal{K}$ is given in [52], the front-end itself is the topic of [51]. A preliminary report on the front-end can be found in [48]; system descriptions have been published as [50] and [49].

9.2 SQL3 Front-End

DLV also offers an SQL3 front-end, based on the ISO/ANSI SQL standard of 1999. This front-end translates a subset of SQL3 query expressions to

datalog queries and supports recursive query processing such as the list-of-materials problem, where we have items that consist of other items that again consist of other items and so forth, and have to transform this tree structure into a list of all dependencies.

Since there are no column names in datalog, we introduced the construct

```
DATALOG SCHEMA Relationname(Columnname, ...);
```

which creates a connection between the parameters of a datalog predicate and the column names of an SQL table.

Example 9.2.1 (List of Materials)
Consider the canonical list of materials query:

```
DATALOG SCHEMA consists_of(major,minor);

WITH RECURSIVE listofmaterials(major,minor) AS
     (
     SELECT c.major, c.minor FROM consists_of AS c
     UNION
     SELECT c1.major, c2.minor
       FROM consists_of AS c1, listofmaterials AS c2
       WHERE c1.minor = c2.major
     )
SELECT major, minor FROM listofmaterials;
```

This query is translated into

```
listofmaterials(A,B) ← consists_of(A,B).
listofmaterials(A,B) ← consists_of(A,C), listofmaterials(C,B).
sql2dl__intern0(A,B) ← listofmaterials(A,B).
```

and the elements of `sql2dl__intern0` are printed in tabular form as the result.

9.3 Meta-Interpreter Front-Ends

In [46, 47] we have presented a kind of meta-interpretation technique for prioritized logic programs, which can be used to compute preferred and weakly preferred answer sets as defined in [19]. In this setting, a special program – the meta-interpreter – is used in conjunction with a suitably transformed prioritized program, which forms a fact base. The answer sets of the meta-interpreter and the transformed input then correspond to the preferred answer sets.

Note that this is a very lightweight kind of front-end, as the initial transformation is quite simple and the meta-interpreter is the same for any input. The output transformation is constituted by a simple filter.

In [47] we have elaborated on this approach and have provided additional meta-interpreters for variants of preferred answer set semantics, based on a unifying characterization defined in [113].

9.4 Third-Party Front-Ends

Also other researchers have created front-ends for DLV. The most notable of these, the *plp* system, has been described in [36] and is available on its web-page at `http://www.cs.uni-potsdam.de/~torsten/plp/`. It can use DLV or Smodels as a back-end for computing a particular notion of preferred answer sets.

Another foreign front-end has recently been implemented. It is called *nlp* and transforms logic programs with nested expressions to Answer Set Programs. DLV is then used as the kernel to compute answer sets, which are then post-processed to yield the answer sets of the original program with nested expressions. A description of this front-end can be found in [102] and on its web-page `http://www.cs.uni-potsdam.de/~torsten/nlp/`, from which it can be downloaded.

Chapter 10

Conclusions and Future Work

In this thesis, we have presented methods for enhancing the efficiency and expressiveness of ASP systems, and in particular of the ASP system DLV, which is based on disjunctive logic programming.

In the first part we have shown how to improve the efficiency of DLV, by first analyzing some modules in the system, which are critical for performance (Deterministic Consequences and Choice inside the Model Generator) and have subsequently defined and evaluated advanced methods for these modules.

In particular, we have defined a method for computing the deterministic consequences, which builds on well-known techniques from satisfiability checking and was tailored to fit the peculiarities of ASP. A main difference between satisfiability checking and ASP is the supportedness principle, and consequently our method exploits knowledge about this ASP feature.

We have also defined several methods for choice heuristics. Here, we have again ported ideas and heuristics from satisfiability checking to the ASP setting, and in addition we have defined heuristics based on methods used in the Smodels system. Moreover, we have also defined a novel heuristics which also exploits the supportedness principle. These approaches have been evaluated experimentally, suggesting that the supportedness-oriented heuristics is competitive.

All of the presented heuristics involve look-aheads, which yields very powerful heuristics, but often becomes the dominating factor (in a negative sense) for efficency. Therefore, we have conceived two methods for reducing the computational overhead of these look-aheads. We have experimentally assessed the benefit of these techniques and have also shown that a combination of both approaches yields the best results.

In the second part, we have demonstrated how to enhance the expressiveness (seen from the viewpoints of both complexity and knowledge repre-

sentation) of the ASP system DLV. We have defined various extensions and front-ends to the basic language, DLP.

The first extension introduces a new construct, weak constraints, allowing for a flexible way of expressing optimality criteria among answer sets. The associated semantics chooses those answer sets which are optimal with respect to the weak constraints among all answer sets of the weak constraint free program. Weak constraints enhance expressiveness both in the context of knowledge representation and of complexity.

The second extension introduces inheritance hierarchies. It turns out that this extension yields a more expressive language in terms of knowledge representation, but computationally it stays in the same complexity class as the original DLP formalism. It is therefore possible to implement the resulting language as a front-end to DLV, which has indeed been realized in the system.

Next, we have shown how to transform diagnostic reasoning — a task, which is quite different from the basic language — to ASP. We have defined a front-end which utilizes DLV as a computational kernel for solving various diagnostic reasoning problems.

Finally, we have mentioned other front-ends, some of which have been realized in the scope of the DLV project, such as the Planning Front-End, the SQL3 Front-End, and Front-Ends for computing preferred answer sets, and some of which have been implemented by other researchers.

In total, we have shown the methods and techniques which make DLV the state-of-the-art system for full (disjunctive) Answer Set Programming.

The results assert that ASP is a paradigm which allows for straightforward and clear encodings (by following the Guess & Check principle), and at the same time for efficient evaluation. We have shown how to speed up this computation effectively, which is important on the one hand for computing solutions to large problem instances and on the other hand for easing rapid prototyping, for which a short response-time of the system is critical.

Furthermore, our work shows that an ASP system is very well-suited as a kernel system which performs the basic computations for enhanced ASP formalisms as well as for formalisms which are quite different from ASP, but for which a computationally easy transformation can be formulated. We believe that having such front-ends is very important for the success of ASP, as this technique allows for "hiding" of the logic programming rules. In this way people can make use of ASP without having to learn a new language — they can use the language they are used to, which is transparently transformed to the ASP formalism. Also for ASP researchers, this technique proves to be very useful, as DLV can serve as a testbed for experimenting with new languages and semantics by means of (usually straightforward to implement) front-ends.

Future work will focus on refining the heuristics even more. In particular, it should be investigated whether feasible heuristics do exist, which do not

rely on look-aheads, can be feasible. Recent results in satisfiability checking (manifested in the Chaff solver [97]) suggest this possibility.

But also the other crucial module, computing deterministic consequences, needs further considerations. In particular, some techniques exist, which have not been incorporated into DetCons previously, because their computational complexity is prohibitive in general. However, one can identify syntactical subclasses of programs, for which an efficient implementation of these operators is feasible and allows the computation of more information. Preliminary work on this issue has been reported in [25].

Concerning front-ends, many important formalisms are known to be candidates for front-ends to `DLV`. Currently our main focus is on the Planning Front-End, for which we plan to implement many refinements and extensions.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] José Júlio Alferes, João Alexandre Leite, Luís Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. Dynamic Logic Programming. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 98–111. Morgan Kaufmann, 1998.

[3] José Júlio Alferes and Luís Moniz Pereira. On Logic Program Semantics with Two Kinds of Negation. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'92)*, pages 574–588. MIT Press, 1992.

[4] José Júlio Alferes, Luís Moniz Pereira, and Teodor C. Przymusinski. Strong and Explicit Negation in Non-Monotonic Reasoning and Logic Programming. In *European Workshop on Logics in Artificial Intelligence (JELIA'96)*, pages 143–163. Springer, 1996.

[5] José Júlio Alferes, Luís Moniz Pereira, and Teodor C. Przymusinski. 'Classical' Negation in Nonmonotonic Reasoning and Logic Programming. *Journal of Automated Reasoning*, 20(1/2):107–142, 1998.

[6] Christian Anger, Kathrin Konczak, and Thomas Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. 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 406–410. Springer Verlag, September 2001.

[7] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[8] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. DisLoP: A Research Project on Disjunctive Logic Programming. *AI Communications – The European Journal on Artificial Intelligence*, 10(3/4):151–165, 1997.

[9] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[10] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[11] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[12] R. Ben-Eliyahu and L. Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 39–50, 1994.

[13] Nicole Bidoit and Christine Froidevaux. General Logical Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation*, 91:15–54, 1991.

[14] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. In V. W. Marek, A. Nerode, and M. Truszczyński, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, number 928 in Lecture Notes in AI (LNAI). Springer, June 1995.

[15] Stefan Brass and Jürgen Dix. Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation. In Leon Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming*, pages 199–213, Tokyo, June 1995. MIT Press.

[16] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. Extended Abstract appeared as [14].

[17] Stefan Brass and Jürgen Dix. Semantics of (Disjunctive) Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, 38(3):167–213, 1999. Extended Abstract appeared as [15].

[18] Gerhard Brewka and Thomas Eiter. Preferred Answer Sets for Extended Logic Programs. In A.G. Cohn, L. Schubert, and S.C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 86–97. Morgan Kaufmann, June 2–4 1998.

[19] Gerhard Brewka and Thomas Eiter. Preferred Answer Sets for Extended Logic Programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

[20] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive Logic Programs with Inheritance. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 79–93, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[21] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3), May 2002.

[22] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Disjunctive Ordered Logic: Semantics and Expressiveness. 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 418–429. Morgan Kaufmann Publishers, 1998.

[23] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Semantics and Expressiveness of Disjunctive Ordered Logic. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):311–337, November 1999.

[24] 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.

[25] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pruning Operators for Answer Set Programming Systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pages 200–209, April 2002.

[26] Weidong Chen and David Scott Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[27] Paweł Cholewiński, V. Wiktor Marek, Artur Mikitiuk, and Mirosław Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.

[28] Paweł Cholewiński, V. Wiktor Marek, and Mirosław Truszczyński. Default Reasoning System DeReS. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pages 518–528, Cambridge, Massachusetts, USA, 1996. Morgan Kaufmann Publishers.

[29] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[30] Luca Console and Pietro Torasso. A Spectrum of Logical Definitions of Model-based Diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[31] James M. Crawford and Larry D. Auton. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence*, 81(1–2):31–57, March 1996.

[32] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5:394–397, 1962.

[33] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[34] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing Diagnoses and Systems. *Artificial Intelligence*, 56(2–3):197–222, 1992.

[35] Jim Delgrande, Torsten Schaub, and Hans Tompits. Logic Programs with Compiled Preferences. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, pages 392–398. IOS Press, 2000.

[36] 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.

[37] J. Dix, G. Gottlob, and W. Marek. Causal Models for Disjunctive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 290–302, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[38] Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, 22:227–255, 1995.

[39] Jürgen Dix. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, 22:257–288, 1995.

[40] Jürgen Dix, Georg Gottlob, and V. Wiktor Marek. Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations. *Fundamenta Informaticae*, 28:87–100, 1996. (This is a full version of [37].).

[41] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[42] Phan Minh Dung. Representing Actions in Logic Programming and Its Applications in Database Updates. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 222–238, 1993.

[43] 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.

[44] T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-94)*, pages 267–278. ACM Press, May 1994.

[45] 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.

[46] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing Preferred and Weakly Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. 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 45–52, Stanford, California, USA, March 2001. AAAI Press.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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*, 5(2), April 2004.

[53] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. Considerations on Updates of Logic Programs. In M. Ojeda-Aciego, I. P. de Guzmán, G. Brewka, and L. Moniz Pereira, editors, *Proceedings European Workshop on Logics in Artificial Intelligence – Journées Européennes sur la Logique en Intelligence Artificielle (JELIA 2000), Malaga, Spain, September 29–October 2, 2000*, number 1919 in LNCS, pages 2–20. Springer, 2000.

[54] 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.

[55] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[56] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[57] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix and Ulrich Furbach and Anil Nerode, editor, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Dagstuhl, Germany, July 1997. Springer.

[58] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The Architecture of a Disjunctive Deductive Database System. In Moreno Falaschi, Marisa Navarro, and Alberto Policriti, editors, *Proceedings Joint Conference on Declarative Programming (APPIA-GULP-PRODE '97)*, pages 141–151, June 1997.

[59] 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.

[60] 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.

[61] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[62] Wolfgang Faber. Disjunctive Datalog with Strong and Weak Constraints: Representational and Computational Issues. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, 1998.

[63] Wolfgang Faber. `DLVi` homepage, 1999. `http://www.dlvsystem.com/inheritance/`.

[64] 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.

[65] J.A. Fernández and J. Minker. Semantics of Disjunctive Deductive Databases. In *Proceedings 4th Intl. Conference on Database Theory (ICDT-92)*, pages 21–50, Berlin, October 1992.

[66] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[67] 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.

[68] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[69] Michael Gelfond and Vladimir Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17:301–321, 1993.

[70] Michael Gelfond and Tran Cao Son. Reasoning with Prioritized Defaults. In *Proceedings of the Workshop of Logic Programming and Knowledge Representation (LPKR '97)*, pages 164–223. Springer, 1997.

[71] 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.

[72] Steve Hanks and Drew McDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[73] Katsumi Inoue and Chiaki Sakama. Transforming Abductive Logic Programs to Disjunctive Programs. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 335–353, 1993.

[74] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 1993.

[75] A.C. Kakas and P. Mancarella. Database Updates Through Abduction. In *Proceedings of the 16th VLDB Conference*, pages 650–661, Brisbane, Australia, 1990.

[76] Christoph Koch and Nicola Leone. Stable Model Checking Made Easy. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999*, pages 70–75, Stockholm, Sweden, August 1999. Morgan Kaufmann Publishers.

[77] Kurt Konolige. Abduction Versus Closure in Causal Theories. *Artificial Intelligence*, 53:255–272, 1992.

[78] Robert A. Kowalski and Fariba Sadri. Logic Programs with Exceptions. In *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, pages 598–616. MIT Press, 1990.

[79] Nicola Leone, Luigi Palopoli, and Massimo Romeo. A Language for Updating Logic Programs and its Implementation. *Journal of Logic Programming*, 23(1):1–61, 1995.

[80] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Declarative and Fixpoint Characterizations of Disjunctive Stable Models. In *Proceedings of the International Logic Programming Symposium – ILPS'95*, pages 399–413, Portland, Oregon, December 1995. MIT Press.

[81] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[82] 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.

[83] 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.

[84] 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.

[85] Vladimir Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[86] 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.

[87] 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.

[88] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[89] Victor W. Marek and Mirosław Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[90] V.W. Marek and M. Truszczyński. Revision Specifications by Means of Programs. In *European Workshop on Logics in Artificial Intelligence (JELIA'94)*, pages 122–136. Springer, 1994.

[91] W. Marek and M. Truszczyński. Modal Logic for Default Reasoning. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):275–302, 1990.

[92] Norman McCain. The Causal Calculator Homepage, 1999. `http://www.cs.utexas.edu/users/tag/cc/`.

[93] John McCarthy. *Formalization of Common Sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.

[94] 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 [93].

[95] Jack Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[96] Michel Minoux. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29:1–12, 1988.

[97] 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.

[98] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In Michael J. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96)*, pages 289–303, Bonn, Germany, September 1996. MIT Press.

[99] Ilkka Niemelä and Patrik Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[100] Donald Nute. Defeasible Logic. In Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 353–395. Oxford University Press, 1994.

[101] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[102] David Pearce, Vladimir Sarsakov, Torsten Schaub, Hans Tompits, and Stefan Woltran. A Polynomial Translation of Logic Programs with Nested Expressions into Disjunctive Logic Programs: Preliminary Report. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, 2002.

[103] Yun Peng and James A. Reggia. *Abductive Inference Models for Diagnostic Problem Solving*. Symbolic Computation – Artificial Intelligence. Springer, 1990.

[104] Gerald Pfeifer. *DLV: Evaluation Algorithms and Efficient Implementation*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, Wien, Österreich, 2000.

[105] D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5(1):97–110, 1989.

[106] David Poole. Normality and Faults in Logic-Based Diagnosis. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI '89)*, pages 1304–1310, Detroit, Michigan, USA, 1989.

[107] Shekhar Pradhan and Jack Minker. Using Priorities to Combine Knowledge Bases. *International Journal of Cooperative Information Systems*, 5(2&3):333–364, 1996.

[108] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[109] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[110] Raymond Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32:57–95, 1987.

[111] Chiaki Sakama and Katsumi Inoue. On the Equivalence between Disjunctive and Abductive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 489–503, 1994. Extended version "On the Relation between Abductive and Disjunctive Logic Programming" (1998).

[112] Chiaki Sakama and Katsumi Inoue. Representing Priorities in Logic Programs. In *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming (JICSLP'96)*, pages 82–96. MIT Press, 1996.

[113] Torsten Schaub and Kewen Wang. A Comparative Study of Logic Programs with Preference. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 597–602, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[114] Dietmar Seipel and Helmut Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In Antoni Olivé, editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politecnica de Catalunya (UPC), 1994.

[115] Bart Selman and Henry Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.

[116] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[117] Patrik Simons. Towards Constraint Satisfaction through Logic Programs and the Stable Model Semantics. Technical Report 47, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Otaniemi, Finland, August 1997.

[118] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[119] David S. Touretzky. *The Mathematics of Inheritance Systems*. Pitman, London, 1986.

[120] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

Appendix A

Instantiation of the Hamiltonian Path Program

Here are the rules of the Hamiltonian path program in Figure 5.5, instantiated with example graph 2 of Figure 5.8 on page 47, the encoding of which is given in Figure 5.9 on page 47:

```
                                    ← inPath(a,b), inPath(d,b).
← inPath(a,b), inPath(a,c).         ← inPath(d,b), inPath(a,b).
← inPath(a,b), inPath(a,d).         ← inPath(a,c), inPath(b,c).
← inPath(a,b), inPath(a,e).         ← inPath(b,c), inPath(a,c).
← inPath(a,c), inPath(a,b).         ← inPath(c,d), inPath(a,d).
← inPath(a,c), inPath(a,d).         ← inPath(a,d), inPath(c,d).
← inPath(a,c), inPath(a,e).         ← inPath(d,e), inPath(a,e).
← inPath(a,d), inPath(a,b).         ← inPath(a,e), inPath(d,e).
← inPath(a,d), inPath(a,c).
← inPath(a,d), inPath(a,e).         ← not reached(b).
← inPath(d,b), inPath(d,e).         ← not reached(c).
← inPath(d,e), inPath(d,b).         ← not reached(d).
                                    ← not reached(e).

inPath(a,b) ∨ outPath(a,b).   reached(b) ← reached(a), inPath(a,b).
inPath(a,c) ∨ outPath(a,c).   reached(b) ← reached(d), inPath(d,b).
inPath(a,d) ∨ outPath(a,d).   reached(c) ← reached(a), inPath(a,c).
inPath(a,e) ∨ outPath(a,e).   reached(c) ← reached(b), inPath(b,c).
inPath(b,c) ∨ outPath(b,c).   reached(d) ← reached(c), inPath(c,d).
inPath(c,d) ∨ outPath(c,d).   reached(d) ← reached(a), inPath(a,d).
inPath(d,b) ∨ outPath(d,b).   reached(e) ← reached(d), inPath(d,e).
inPath(d,e) ∨ outPath(d,e).   reached(e) ← reached(a), inPath(a,e).
```

Appendix B

Consistency-Based Diagnosis

In this appendix we present the algorithms that rewrite CDPs (as defined in Section 8.2) to disjunctive logic programs (as defined in Section 2.1). As in the case of ADPs, we will first give the algorithm for generic consistency-based diagnoses, then for single error and finally for subset minimal diagnoses.

For consistency based problems, we impose some syntactic restrictions: As stated after Definition 8.2.1, hypotheses should consist only of the predicate `ab`, and negative literals in the body are only allowed if the involved atoms are hypotheses.

Additionally, for subset minimal diagnostic problems we do not allow disjunctions and hypotheses may only occur negatively (i.e. as `not ab(...)`).

The basic consistency-based translation is performed by the function in Fig. B.1, which is similar to the basic transformation in the abductive case (see Fig. 8.4). The difference is that positive observations `a` are translated to facts `a` ← . instead of constraints ← `not a`. (lines (10) and (12) in the function in Fig. B.1).

← `not a`. is introduced in the abductive case to disallow models in which the observation `a` does not hold. In the consistency-based case positive observations need not be entailed. Rather, if we assume them to hold (by adding the corresponding fact to the theory), they must not give reason to inconsistencies. Negative observations need not be inserted as facts, because of the concept of negation as failure. Rather, we must ensure that the positive variants of negative observations are not derived (since this would represent an inconsistency). Therefore we add ← `not a`..

Additionally we have to include constraints for each hypothesis, which forbids the hypothesis to become true unless the corresponding `_sol(i)` is also true (line (6)). If `_sol(i)` does not hold, the corresponding hypothesis is assumed to be false (cf. Definition 8.2.2). If the same hypothesis is then derived (i.e. becomes true), the diagnosis leads to an inconsistency. These constraints could be omitted if we disallowed hypotheses to occur in the

Function $ConsBasedToDLP(\mathcal{P}) : DLP$

Input: A Consistency Based Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive (except for hypotheses) and non-disjunctive.

Output: A Datalog program.

var i, j: $Integer$;
$q : DLP$;

begin

(1) $q := T$;

(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) **for** $i := 1$ **to** n **do**
(4) $q := q \cup \{h_i \leftarrow \_sol(i).\}$;
(5) **for** $i := 1$ **to** n **do**
(6) $q := q \cup \{\leftarrow h_i, \texttt{not } \_sol(i).\}$;

(7) Let $O = \{o_1, \ldots, o_m\}$;
(8) **for** $j := 1$ **to** m **do**
(9) **if** o_j is a positive literal "a" **then**
(10) $q := q \cup \{a.\}$;
(11) **else** ($*$ o_j is a negative literal "`not` a" $*$)
(12) $q := q \cup \{\leftarrow a.\}$;

(13) **return** q;

end

Figure B.1: Basic Consistency Based Diagnosis Translation

head of rules in the theory, because then the only way in which hypotheses could become true is via the corresponding `_sol(i)`.

Algorithm 4 General Consistency-Based Diagnosis

Function $GeneralConsBasedToDLP(\mathcal{P}) : DLP$

Input: A Consistency Based Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive (except for hypotheses) and non-disjunctive.

Output: A Datalog program.

var $q : DLP$;
begin
(1) $q := ConsBasedToDLP(\mathcal{P})$;
(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) **for** $i := 1$ **to** n **do**
(4) $q := q \cup \{\_sol(i) \vee \_notsol(i).\}$;
(5) **return** q;
end

Theorem B.0.1
Let $\mathcal{P} = \langle H, T, O \rangle$ be a consistency-based problem and $DLP(\mathcal{P})$ the logic program generated by Algorithm 4 with input $\mathcal{P}$. There exists a many to one correspondence between the stable models of $DLP(\mathcal{P})$ and the diagnoses of $\mathcal{P}$. In particular, if M is a stable model of $DLP(\mathcal{P})$, then $ModelToDiagnosis(M)$ (see Fig. 8.5) is a diagnosis *for $\mathcal{P}$.*
□

Demonstration: As in Algorithm 1, the rules in instruction (4) of Algorithm 4 generate all possible diagnosis candidates by assuming either $\_sol(i)$ or $\_notsol(i)$ for each hypothesis h_i. These candidates are then checked by the rules from the function in Fig. B.1, as described above. □

Single error diagnoses of CDPs are translated in analogy to Algorithm 2. The only difference is that the basic translation is accomplished by the function $ConsBasedToDLP$ (see Fig. B.1).

Theorem B.0.2
Let $\mathcal{P} = \langle H, T, O \rangle$ be a consistency-based diagnostic problem and $DLP(\mathcal{P})$ the logic program generated by Algorithm 5 with input $\mathcal{P}$. There exists a many to one correspondence between the stable models of $DLP(\mathcal{P})$ and a single error diagnosis of $\mathcal{P}$ and vice versa. In particular, if M is a stable model of $DLP(\mathcal{P})$, then $ModelToDiagnosis(M)$ is a single error diagnosis *for $\mathcal{P}$.*
□

Algorithm 5 Single Error Consistency Based Diagnosis

Function $SingleErrorConsBasedToDLP(\mathcal{P}) : DLP$

Input: A Consistency Based Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive (except for hypotheses) and non-disjunctive.

Output: A Datalog program.

var $q : DLP$;
begin
(1) $q := ConsBasedToDLP(\mathcal{P})$;
(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) $q := q \cup \{\_sol(1) \vee \ldots \vee \_sol(n).\}$;
(4) **return** q;
end

Demonstration: This case is very similar to the general one (Algorithm 4 and Theorem B.0.1), and as in Algorithm 2, line (3) ensures that exactly one hypothesis can become true because of the minimality of stable models. □

Algorithm 6 Subset Minimal Consistency Based Diagnosis

Function $SubsetMinimalConsBasedToDLP(\mathcal{P})$
$: DLP$

Input: A Consistency Based Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive (except for hypotheses) and non-disjunctive.

Output: A Datalog program.

var $q : DLP$;
begin
(1) $q := GeneralConsBasedToDLP(\mathcal{P})$;
(2) $q := q \cup ConsBasedSubsetMinimality(\mathcal{P})$;
(3) **return** q;
end

The program for computing subset minimal consistency-based diagnoses is generated by Algorithm 6. As Algorithm 3, it consists of a part which generates a solution, and a part which checks for minimality. This part is produced by the function in Fig. B.2, which in turn is similar to the one depicted in Fig. 8.6.

Theorem B.0.3
Let $\mathcal{P} = \langle H, T, O \rangle$ be a consistency-based diagnostic problem and $DLP(\mathcal{P})$ the logic program given by Algorithm 6 with input $\mathcal{P}$. There exists a many

Function $ConsBasedSubsetMinimality(\mathcal{P}) : DLP$

Input: A Consistency Based Problem $\mathcal{P} = \langle H, T, O \rangle$
where T is positive (except for hypotheses) and non-disjunctive,
and hypotheses may only occur as negative literals.

Output: A Datalog program.

var i, j: $Integer$;
q : DLP;

begin

(1) $q := \emptyset$;

(2) Let $H = \langle h_1, \ldots, h_n \rangle$;
(3) **for** $i := 1$ **to** n **do**
(4) $q := q \cup \{\_hyp(i)\}$;

(5) $q := q \cup \{\_sol1(K, I) \leftarrow \_sol(K), \_hyp(I), <> (K, I).\}$;

(6) **for** $i := 1$ **to** n **do**
(7) $q := q \cup \{\_\_h_i(\overline{x}, I) \leftarrow \_sol1(i, I).\}$;

(8) **for** each $r : a(\overline{x}) \leftarrow b_1(\overline{x_1}), \ldots, b_n(\overline{x_n}). \in T$ **do**
(9) $q := q \cup \{\_\_a(\overline{x}, I) \leftarrow \_hyp(I), \_\_b_1(\overline{x_1}, I), \ldots, \_\_b_n(\overline{x_n}, I).\}$;

(10) **for** each $c :\leftarrow c_1(\overline{x_1}), \ldots, c_n(\overline{x_n}). \in T$ **do**
(11) $q := q \cup \{\_notobs(I) \leftarrow \_\_c_1(\overline{x_1}, I), \ldots, \_\_c_n(\overline{x_n}, I), \_hyp(I).\}$;

(12) Let $O = \{o_1, \ldots, o_m\}$;
(13) **for** $j := 1$ **to** m **do**
(14) **if** o_j is a positive literal "$a(\overline{x})$" **then**
(15) $q := q \cup \{a(\overline{x}, I) \leftarrow \_hyp(I).\}$;
(16) **else** ($*$ o_j is a negative literal "`not` $a(\overline{x})$" $*$)
(17) $q := q \cup \{\_notobs(I) \leftarrow \_hyp(I), a(\overline{x}, I).\}$;

(18) $q := q \cup \{\_isntsol(0).\}$;
(19) $q := q \cup \{\_isntsol(I) \leftarrow \_isntsol(K), \#succ(K, I), \_notobs(I).\}$;
(20) $q := q \cup \{\_isntsol(I) \leftarrow \_isntsol(K), \#succ(K, I), \texttt{not}\ \_sol(I).\}$;
(21) $q := q \cup \{\leftarrow \texttt{not}\ \_isntsol(n).\}$;

(22) **return** q;

end;

Figure B.2: Subset Minimality Check for Consistency Based Diagnosis

to one correspondence between a stable models of $DLP(\mathcal{P})$ and the subset minimal diagnoses of $\mathcal{P}$ and vice versa. In particular, if M is a stable model of $DLP(\mathcal{P})$, then $ModelToDiagnosis(M)$ (see Fig. 8.5) is a subset minimal diagnosis *for $\mathcal{P}$.*
□

Demonstration: The algorithm is based on the same ideas as its abductive counterpart, as demonstrated after Theorem 8.3.3. Only minor changes have to be applied to the function which generates the minimality check (depicted in Fig. 8.6), which stem from the different definition of diagnoses and from the different syntactical restrictions.

The differences are how the observations are translated (lines (14) to (17)) and that constraints (and not only rules) have to be parameterized w.r.t. hypotheses of a diagnosis which has to be checked for subset minimality (lines (10) and (11)).

The observations are translated in analogy to the basic consistency-based translation in Fig. B.1.

The reason why constraints must be parameterized in the consistency-based minimality check, but not in the abductive one is due to the restrictions which are imposed on the theories:

In the abductive case, we only allow positive, non-disjunctive theories. If some hypothesis is omitted from a diagnosis, no additional atoms can be derived (because the program is positive and non-disjunctive). Therefore, if a constraint was satisfied for a solution, at least one of its atoms[1] was false, and it must also be false for the smaller diagnosis candidate.

In the consistency-based case, hypotheses may occur negatively in bodies. For this reason, if a hypothesis is omitted from a valid diagnosis, a formerly satisfied constraint may become violated (for example, if the hypothesis which is omitted from the diagnosis occurs negatively in a constraint), so parameterized constraints must be included. These constraints are rewritten to rules — if one of their bodies becomes true, an atom is derived which represents the fact that this subdiagnosis is not a valid diagnosis (`_notobs(I)`).
□

[1]Since only positive theories are allowed, a constraint contains only atoms.

Curriculum Vitae et Studiorum

Wolfgang Faber

Personal Data

Degree:	Diplom-Ingenieur (M.Sc.)
Date and Place of Birth:	8. April 1974, Vienna, Austria
Citizenship:	Austrian
Family Status:	unmarried, no children
Current Position:	Universitätsassistent (teaching and 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
Office Phone:	+43 1 58801 18465
Office Fax:	+43 1 58801 18493
E-mail:	faber@kr.tuwien.ac.at
Homepage:	`http://www.wfaber.com`
Home Address:	Harkortstraße 9/3, A–1020 Vienna, Austria
Home Phone:	+43 1 7202972

Education

since October 1998	Doktoratsstudium, Studienzweig Informatik (PhD course in Computer Science) at Vienna University of Technology.
June 1998	Graduation with honors. Thesis "Disjunctive Datalog with Strong and Weak Constraints: Representational and Computational Issues." [62] advised by Prof. N. Leone.
October 1992 – June 1998	Diplomstudium Informatik (Diploma Study in Computer Science) at Vienna University of Technology.
June 1992	Graduation from secondary school (with honors).

Working Experience

since October 1999	Universitätsassistent (Teaching and Research Assistant) at the Knowledge Based Systems Group, Vienna University of Technology.
July 1998 – September 1999	Research Assistant in the project P-11580-MAT *"Design and Implementation of a Query System for Disjunctive Deductive Databases"*, funded by FWF, the Austrian Science Fund.
March 1997 – June 1998	Student Fellow in the project P11580-MAT.
1995 – 1996	Studienassistent (Student Teaching Assistant) at the Institute of Computer Languages, Vienna University of Technology.
1994 – 1998	Tutor for the lab course "Logic-oriented Programming Languages".

Research Interests

- Knowledge Representation and Reasoning
- Logic Programming and Nonmonotonicity
- Declarative Languages and Semantics
- Planning, Diagnostic Reasoning
- Implementation and Evaluation of Systems (in the areas mentioned above)

Teaching Experience

- Lecture "Deductive Databases".
- Organization of a Logic Programming lab course.
- Organization of practica.
- Organization of seminars.
- Student advisor for a Logic Programming lab course.

Activities in the Scientific Community

Committee Membership

- 8th European Conference on Logics in Artificial Intelligence (JELIA'02)

Conference Organization

- Publicity Chair of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)
- Publicity and Local Organization for the Joint German/Austrian Conference on Artificial Intelligence (KI-2001)

Peer Reviews

Journals and Collections

- Information Systems
- Information Fusion
- Theory and Practice of Logic Programming
- Computational Logic: From Logic Programming into the Future (In honour of Bob Kowalski)

Conferences and Workshops

- STarting Artificial Intelligence Researchers Symposium (STAIRS 2002)
- 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)
- 18th International Conference on Logic Programming (ICLP'01)
- 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)
- 17th International Joint Conference on Artificial Intelligence (IJCAI'01)
- First International Joint Conference on Automated Reasoning (IJCAR'01)
- Fourth International Conference on Multi-Agent Systems (ICMAS-2000)
- International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2000)
- First International Conference on Computational Logic (CL2000)
- 7th International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)

- 8th Inernational Workshop on Non-Monotonic Reasoning (NMR'2000)
- 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)
- 10th International Conference and Workshop on Database and Expert Systems Applications (DEXA'99)

Publications

[A] Books and Collections

[1] T. Eiter, W. Faber, and M. aw 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). Springer Verlag, 2001.

[2] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[B] Journals

[3] F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3), May 2002.

[4] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[C] Conferences and Workshops

[5] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning Operators for Answer Set Programming Systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, April 2002.

[6] F. Calimeri, W. Faber, N. Leone, S. Perri, and G. Pfeifer. DLV - Declarative Problem Solving using Answer Set Programming. In *Proceedings of the Seventh Congress of the Italian Association for Artificial Intelligence AI*IA 2001*, Bari, Italy, 2001.

[7] W. Faber, N. Leone, and G. Pfeifer. Optimizing the Computation of Heuristics for Answer Set Programming Systems. In T. Eiter, W. Faber, and M. aw 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 288–301. Springer Verlag, September 2001.

[8] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System Description: `DLV`. In T. Eiter, W. Faber, and M. aw 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 409–412. Springer Verlag, September 2001.

[9] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. System Description: The DLV$^{\mathcal{K}}$ Planning System. In T. Eiter, W. Faber, and M. aw 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.

[10] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

[11] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. The DLV$^{\mathcal{K}}$ Planning System. In A. Cimatti, H. Geffner, E. Giunchiglia, and J. Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 76–81, Aug. 2001.

[12] W. Faber, N. Leone, and G. Pfeifer. A Comparison of Heuristics for Answer Set Programming. In *Proceedings of the 5th Dutch-German Workshop on Nonmonotonic Reasoning Techniques and their Applications (DGNMR 2001)*, pages 64–75, Apr. 2001.

[13] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing Preferred and Weakly Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. In A. Provetti and S. T. Cao, editors, *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 45–52, Stanford, CA, March 2001. AAAI Press.

[14] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. 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.

[15] T. Eiter, W. Faber, C. Koch, N. Leone, and G. Pfeifer. DLV – A System for Declarative Problem Solving. In C. Baral and M. Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[16] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Using the `dlv` system for planning and diagnostic reasoning. In F. Bry, U. Geske, and D. 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.

[17] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In M. Gelfond, N. Leone, and G. 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 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

[18] F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. In D. D. Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 79–93, Las Cruces, New Mexico, USA, Nov. 1999. The MIT Press.

[19] W. Faber, N. Leone, and G. Pfeifer. dlv: a DLP System for AI. In E. Lamma and P. Mello, editors, *Sixth Congress of the Italian Association for Artificial Intelligence (AIIA '99)*, pages 511–514, Bologna, Italy, September 1999. Pitagora Editrice Bologna.

[20] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In I. O. 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.

[21] T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[22] W. Faber, N. Leone, and G. Pfeifer. Representing school timetabling in a disjunctive logic programming language. In U. Egly and H. Tom-

pits, editors, *Proceedings of the 13th Workshop on Logic Programming (WLP'98)*, pages 43–52, Vienna, Austria, October 1998.

[23] R. Bihlmeyer, W. Faber, C. Koch, N. Leone, C. Mateis, and G. Pfeifer. `dlv` – an overview. In U. Egly and H. Tompits, editors, *Proceedings of the 13th Workshop on Logic Programming (WLP'98)*, pages 65–67, Vienna, Austria, October 1998.

[24] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The `dlv` System: Model Generator and Application Frontends. In F. Bry, B. Freitag, and D. Seipel, editors, *Proceedings of the 12th Workshop on Logic Programming (WLP'97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.

[D] Papers under Review

[25] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. Technical Report INFSYS RR-1843-02-01, Institut für Informationssysteme, Technische Universität Wien, Jan. 2002.

[26] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. Technical Report INFSYS RR-1843-01-12, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001.

[27] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. Technical Report INFSYS RR-1843-01-11, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001.

[E] Further Publications and Reports

[28] W. Faber. Disjunctive datalog with strong and weak constraints: Representational and computational issues. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, 1998.

[29] W. Faber, N. Leone, and G. Pfeifer. Computing Consistent Preferred Answer Sets. Technical Report DBAI-TR-99-28a, Institut für Informationssysteme, Technische Universität Wien, Austria, April 1999.

[F] Web Pages

[30] W. Faber and G. Pfeifer. `dlv` homepage, since 1996. `<URL:http://www.dbai.tuwien.ac.at/proj/dlv/>`.

[31] W. Faber. `dlvi` homepage, 1999. `<URL:http://www.dbai.tuwien.ac.at/proj/dlv/inheritance/>`.