# Extension and Efficient Evaluation of Disjunctive Logic Programs

**Francesco Calimeri**

*Dipartimento di Matematica,*
*Università della Calabria*
87036 *Rende, Italy*
*email : calimeri@mat.unical.it*

*A mamma, papà, Maria, Teresa. A chi mi vuole bene.*

## Sommario

La Programmazione Logica Disgiuntiva (PLD, o DLP, all'inglese), è un approccio dichiarativo alla programmazione, che è stato proposto di recente nell'area del ragionamento non monotono e della programmazione logica. Può essere anche vista come una alternativa di programmazione logica alla programmazione basata su SAT, che è largamente usata, e con successo, nell'area dell'Intelligenza Artificiale [Kautz and Selman, 1992].

La DLP è molto ''espressiva'': cattura interamente, sotto la semantica degli *Answer Set* (o *Modelli Stabili*), l'intera classe $\Sigma_2^P$ ($\mathrm{NP}^{\mathrm{NP}}$). Tuttavia, il suo alto potere espressivo comporta un elevato costo sul piano computazionale, e questo, per molti anni, ha comportato il fatto che la ricerca sulla DLP stessa è stata condotta sostanzialmente sul solo piano teorico, e la difficoltà nella valutazione dei programmi logici disgiuntivi ha scoraggiato a lungo l'implementazione di sistemi reali.

Sono stati fatti molti sforzi nell'intento di realizzare sistemi DLP efficienti, e dopo alcuni lavori pionieristici [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], diversi moderni sistemi sono ormai disponibili. Tra questi, il sistema DLV consente di sfruttare la DLP per risolvere problemi reali in diverse aree applicative, come ad esempio la pianificazione, la schedulazione (*scheduling*), la correzione automatica di dati da censimento, la manipolazione di dati complessi [Eiter *et al.*, 2000; Simons, 2000; Franconi *et al.*, 2001].

Tuttavia, applicazioni pratiche in diverse aree emergenti, come la gestione della conoscenza (*Knowledge Management*) o l'integrazione delle informazioni (*Information Integration*), richiedono prestazioni sempre più elevate. Di conseguenza, la progettazione e l'implementazione di appropriate tecniche di ottimizzazione sono fondamentali per l'efficienza di sistemi come DLV. Senza contare che, nonostante il già citato potere espressivo della DLP, ci sono alcuni tipi di problemi che non possono essere codificati in modo naturale, e pertanto i programmi risultanti sono spesso complicati e macchinosi.

In questa tesi ci concentriamo sulle questioni succitate: da un lato proponiamo nuove tecniche mirate ad aumentare l'efficienza del sistema DLV; e dall'altro proponiamo nuove estensioni della Programmazione Logica Disgiuntiva che ne aumentano le capacità di modellare conoscenza.

Qui di seguito riportiamo brevemente i principali contributi di questa tesi.

1. Studiamo la DLP e ne analizziamo la complessità e l'utilizzo per la rappresentazione della conoscenza e il ragionamento non monotono.

2. Studiamo i metodi di ‘‘taglio’’ (‘‘pruning’’) dello spazio di ricerca, aspetto cruciale per l’efficienza dei sistemi DLP. Analizziamo due operatori per il pruning, con riferimento alla loro efficienza ed efficacia. Mettiamo a punto una strategia efficiente per combinare i due operatori, che consente di sfruttare i vantaggi di entrambi.

3. Progettiamo algoritmi molto efficienti per l’implementazione delle strategie suddette, e, contestualmente, studiamo nuove tecniche per la maggiore ‘‘localizzazione’’ possibile del calcolo delle ‘‘parti’’ di un programma logico davvero utili per raggiungere una soluzione. Implementiamo l’approccio presentato nel sistema DLV e conduciamo diversi esperimenti.

4. Introduciamo un framework formale, per favorire l’introduzione di predicati esterni nella DLP, chiamato DLP-EX. Questo consente di definire funzioni aggregate estese ‘‘esterne’’ al sistema stesso (non predefinite), definibili dall’utente finale e raggruppabili in librerie, e la cui estensione non è specificata per mezzo di un programma logico, bensì calcolata attraverso del codice esterno.

5. Integriamo il framework DLP-EX stesso nel sistema DLV, e conduciamo alcuni esperimenti; mostriamo inoltre come DLP-EXfaciliti l’applicazione della Programmazione Logica Disgiuntiva in alcuni ambiti di rilievo e, cosa molto importante, come apra la strada ad un modo per simulare i simboli di funzione in uno scenario in cui la nozione di termine rimane semplice.

6. Introduciamo il paradigma ‘‘template’’ nella DLP. L’estensione proposta è chiamata DLP$^T$. Il framework proposto introduce il concetto di predicato ‘‘template’’, la cui definizione può essere sfruttata dovunque sia necessario o desiderato, attraverso il ‘‘binding’’ con predicati usuali, permettendo così di definire moduli riutilizzabili, nuovi costrutti e/o aggregati, e senza alcuna limitazione sintattica.

7. Implementiamo il paradigma ‘‘template’’ all’interno del sistema DLV; tuttavia, esso non si basa su caratteristiche specifiche di DLV, ed è facilmente generalizzabile ad altri sistemi. Esso garantisce l’aumento della dichiaratività del codice, la possibilità di raggruppare i template in librerie, la capacità di introdurre velocemente nuovi costrutti predefiniti e di ottenere una prototipazione rapida di nuove estensioni.

# Contents

## Abstract

Disjunctive Logic Programming (DLP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. It can be view also as a logic programming alternative to SAT-based programming, which is successfully and widely used in the area of Artificial Intelligence [Kautz and Selman, 1992].

DLP is very expressive: it allows to express every property of finite structures that is decidable in the complexity class $\Sigma_2^P$ ($\mathrm{NP}^{\mathrm{NP}}$). However, the high expressiveness of Disjunctive Logic Programming comes at the price of a high computational cost. For many years the research on DLP has been carried out only on the theoretical side, because the hardness of the evaluation of DLP programs has discouraged the implementation of DLP engines for quite some time.

Several efforts have been made in the direction of implementing efficient DLP systems. After some pioneering work on stable model computation [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], a number of modern DLP systems are now available. Among them, the DLV system allows to use DLP for solving real-world problems in a number of application areas, including planning, scheduling, automatic correction of census data, as well as for complex data manipulations [Eiter *et al.*, 2000; Simons, 2000; Franconi *et al.*, 2001].

However, practical applications in many emerging areas, such as Knowledge Management or Information Integration, require always higher performances, and therefore the design and the implementation of suitable optimization techniques are fundamental for the efficiency of systems like DLV. Moreover, despite the high expressive power of DLP, there are some kinds of problems that cannot be encoded in a natural way and then the resulting programs are often complex and tricky.

In this thesis we focus on the issues above: from the one hand we propose new techniques aiming at improving the efficiency of DLV; on the other hand, we propose new extensions of Disjunctive Logic Programming for enhancing its knowledge modelling abilities.

Briefly, the main contributions of the thesis are the following:

1. We study DLP, analyze its complexity and its exploitation for knowledge representation and reasoning.

2. We describe the main steps of the computational process performed by DLP systems with a focus on search space pruning, which is crucial for effi-

ciency. We analyze the properties of the disjunctive extensions of two well-known pruning operators for logic programming. We design an intelligent strategy for combining the two pruning operators cited above, which exploits the advantages of both.

3. We implement our approach in the DLV system, taking care of efficiency issues and respecting the known complexity bounds, and reporting experimental results on a number of benchmark problems to assess the impact of our approach.

4. We introduce a formal framework, for accommodating *external predicates* in the context of Disjunctive Logic Programming. We show that the framework enhances the applicability of DLP to a variety of problems such as string and algebraic manipulation.

5. We discuss implementation issues, and show how we have integrated the support for *external predicate* in the DLV system, which is, this way, enabled with the possibility of using external sources of computation.

6. We introduce the *template* paradigm into DLP, providing syntax and giving a clear operational semantics. We discuss theoretical properties of the extension and its main advantages.

7. We extend the DLV system with the capability to support the *template* paradigm.

# Introduction

## Disjunctive Logic Programming

Disjunctive Logic Programming (DLP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. It can be view also as a logic programming alternative to SAT-based programming, which is successfully and widely used in the area of Artificial Intelligence [Kautz and Selman, 1992]. In SAT-based programming, a given computational problem $P$ is encoded as a propositional CNF formula the models of which correspond to solutions of $P$; a SAT solver is then used to find such models (and thus solutions of $P$). In Disjunctive Logic Programming, a given computational problem $P$ is represented by a DLP program whose stable models correspond to solutions; a DLP system is then used to find such solutions to $P$ [Lifschitz, 1999].

One of the main features of Disjunctive Logic Programming is the higher expressiveness of its language, which enjoys the knowledge modeling power of logic programming features like variables, negation as failure, and disjunction. Indeed, the knowledge representation language of DLP consists of function-free logic programs where disjunction is allowed in the heads and nonmonotonic negation may occur in the bodies of the rules. The DLP language supports the representation of problems of high computational complexity, and, importantly, the DLP encoding of a large variety of problems is often very concise, simple, and elegant [Eiter *et al.*, 2000].

Such encodings are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [Baral and Gelfond, 1994; Lobo *et al.*, 1992; Wolfinger, 1994; Eiter *et al.*, 1999; Gelfond and Lifschitz, 1991; Lifschitz, 1996; Minker, 1994; Baral, 2002]. For instance, one of the attractions of Disjunctive Logic Programming (DLP) is its capability of allowing the natural modeling of incomplete knowledge [Baral and Gelfond, 1994; Lobo *et al.*, 1992].

Much research has been spent on the semantics of disjunctive logic programs, and several alternative semantics have been proposed [Brass and Dix, 1995; Eiter *et al.*, 1997e; Gelfond and Lifschitz, 1991; Minker, 1982; Przymusinski, 1990; 1991; 1995; Ross, 1990; Sakama, 1989] (see [Apt and Bol, 1994; Dix, 1995; Lobo *et al.*, 1992; Minker, 1994; 1996] for comprehensive surveys). The most widely accepted semantics is the *answer sets semantics* proposed by Gelfond and Lifschitz [Gelfond and Lifschitz, 1991], as an extension of the stable model semantics of normal logic programs [Gelfond and Lifschitz, 1988]. According to this semantics, a disjunctive logic program may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world.

Disjunctive Logic Programming under answer sets semantics (also called Answer Set Programming (ASP)) is very expressive. It has been shown ([Eiter *et al.*, 1997c; Gottlob, 1994]) that, under this semantics, disjunctive logic programs capture the complexity class $\Sigma_2^P$ (i.e., they allow us to express, in a precise mathematical sense, *every* property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP). As Eiter *et al.* [Eiter *et al.*, 1997c] showed, the expressiveness of Disjunctive Logic Programming has practical implications, since relevant practical problems can be represented by disjunctive logic programs, while they cannot be expressed by logic programs without disjunctions, given current complexity beliefs. In addition, even problems of lower complexity can be often expressed more naturally by disjunctive programs than by programs without disjunction.

# The DLV System

Several efforts have been made in the direction of implementing efficient DLP systems. After some pioneering work on stable model computation [Bell *et al.*, 1994; Subrahmanian *et al.*, 1995], a number of modern DLP systems are now available. The most widespread DLP systems are DLV [Leone *et al.*, 2005], `GnT` [Janhunen *et al.*, 2005], and recently also Cmodels-3 [Lierler, 2005]. Many other systems support various fragments of the DLP language [Anger *et al.*, 2001; Aravindan *et al.*, 1997; Babovich, since 2002; Chen and Warren, 1996; Cholewiński *et al.*, 1996; 1999; East and Truszczyński, 2000; 2001b; 2001a; Egly *et al.*, 2000; Lierler and Maratea, 2004; Lin and Zhao, 2004; McCain and Turner, 1998; Niemelä and Simons, 1997; Rao *et al.*, 1997; Seipel and Thöne, 1994; Simons *et al.*, 2002].

In this thesis we focus on the DLV system, which is generally recognized to

be the state-of-the-art implementation of Disjunctive Logic Programming. DLV is widely used by researchers all over the world, and it is competitive, also from the viewpoint of efficiency, with the most advanced systems in this area.

The development of DLV started in 1996 at the Vienna University of Technology, in a research project funded by the Austrian Science Funds (FWF); at present, DLV is the subject of an international cooperation between the University of Calabria and the Vienna University of Technology. It is widely used for educational purposes in courses on databases and on AI, both in European and American universities, and has been employed at CERN, the European Laboratory for Particle Physics located near Geneva, for an advanced deductive database application that involves complex knowledge manipulation on large-sized databases.

The industrial exploitation of DLV in the emerging areas of Knowledge Management and Information Integration has been the subject of two international projects funded by the European Commission, namely, INFOMIX (Boosting Information Integration, project IST-2002-33570) and ICONS (Intelligent Content Management System, project IST-2001-32429).

We believe that the strengths of DLV – its expressivity and solid implementation – make it attractive for such hard applications.

## Main Contribution

This thesis concerns the study and the extension of Disjunctive Logic Programming, and the optimization of the DLV system, which implements the DLP itself.

Some evidence raised during our studies.

- in the latest years, the availability of reliable DLP systems induced many people to start exploiting DLP in several application areas, but the current systems are not efficient enough for many of these applications;
- despite its expressiveness, DLP is, in some relevant cases, still unable to model knowledge in a natural way (for instance, is not capable to easily deal with some data types).

Our work faces both the above issues, aiming at overcoming these limitations by:

(i) increasing the efficiency of the DLP systems, and of the DLV system in particular, through design and implementation of new optimization techniques;

(ii) extending DLP in order to enhance its knowledge modelling abilities.

**Optimizing the Evaluation of Disjunctive Logic Programming.** The core of a DLP system is model generation, where a model of the program is produced, which is then subjected to a model check. For the generation of models, DLP systems typically employ procedures which are similar to Davis-Putnam procedures used in SAT solvers. As for SAT solvers, two factors are fundamentally important for the efficiency of model generation in DLP: (i) the *heuristic* (branching rule) for the selection of the branching literal, i.e., the criterion determining the literal to be assumed true at a given stage of the computation; and (ii) the *pruning operator*, i.e., the operator computing the consequences deterministically derivable from the program rules and the interpretation at hand, which enlarges the set of known facts pruning the search space.

Much work has to be done to make DLP systems fully satisfactory for modern knowledge-based applications, and the design of new optimization techniques and smart algorithms for the computation of DLP programs is of utmost importance. Our work goes in this direction, focusing on search space pruning, an extremely critical problem for the performance of DLP systems. Thus:

- We describe the main steps of the computational process performed by DLP systems with a focus on search space pruning, which is crucial for efficiency. We analyze the properties of the disjunctive extensions of two well-known pruning operators for logic programming, *Fitting*'s operator and the *Well-founded* operator. We carry out an in-depth discussion on their strengths and weaknesses w.r.t. efficiency and effectiveness, deriving new properties on these operators which are fundamental for their concrete exploitation in DLP systems.

- We design an intelligent strategy for combining these two pruning operators, which exploits the advantages of both, starting from several known results established in previous works and focusing on modularity properties [Lifschitz and Turner, 1994; Eiter *et al.*, 1997c; Leone *et al.*, 1997], head-cycle free programs [Ben-Eliyahu and Dechter, 1994], acyclic programs [Fages, 1994], disjunctive unfounded sets and complexity [Leone *et al.*, 1997], combining these in a smart way.

- We implement our approach in the DLV system, taking care of efficiency issues and respecting the known complexity bounds. Indeed, the fixpoints

of Fitting's operator are computed in linear time, as are the Greatest Unfounded Sets (which contribute the negative inferences in the Well-founded operator).

- We report experimental results on a number of benchmark problems to assess the impact of our approach. The experiments show that the choice of the pruning operator has a strong influence on the performance of DLP systems, and specifically that our techniques considerably improve the efficiency of the DLV system.

**Extending Disjunctive Logic Programming.** Despite its high expressiveness, there are several problems that DLP cannot encode in a natural way, and thus even the state-of-the-art DLP systems hardly deal with these. For instance, it's quite artful to deal with data types such as strings, natural and real numbers. Although simple, this data types bring two kinds of technical problems: first, they range over infinite domains; second, they need to be manipulated with primitive constructs which can be encoded in logic programming at the cost of compromising efficiency and declarativity. Furthermore, interoperability with other software is nowadays important, especially in the context of those Semantic Web applications aimed at managing external knowledge.

In addition, it is very likely that this new generation of DLP applications require the introduction of repetitive pieces of standard code. Indeed, a major need of complex and huge DLP applications such as [Nogueira *et al.*, 2001] is dealing efficiently with large pieces of such a code and with complex data structures, more sophisticated than the simple, native DLP data types. Indeed, the nonmonotonic reasoning community has continuously produced, in the past, several extensions of nonmonotonic logic languages, aimed at improving readability and easy programming through the introduction of new constructs, employed in order to specify classes of constraints, search spaces, data structures, new forms of reasoning, new special predicates [Cadoli *et al.*, 1999; Eiter *et al.*, 1997a; Kuper, 1990], such as aggregate predicates [Calimeri *et al.*, 2005]. Nonetheless, code reusability has never been considered as a priority in the DLP field, despite the fact that modular logic programming has been widely studied in the general case [Bugliesi *et al.*, 1994; Eiter *et al.*, 1997d].

Our work tries to start filling these gaps.

- We introduce a formal framework, named DLP-EX, for accommodating *external predicates* in the context of Disjunctive Logic Programming.

- DLP-EX includes the explicit possibility of invention of new values from external sources: since this setting could lead to non-termination of any conceivable evaluation algorithm, we tailor specific cases where decidability is preserved.
- We show that DLP-EX enhances the applicability of Disjunctive Logic Programming to a variety of problems such as string and algebraic manipulation. Also the framework paves the way for simulating function symbols in a setting where the notion of term is kept simple (Skolem terms are not necessary).
- We discuss implementation issues, and show how we have integrated DLP-EX in the DLV system, which is, this way, enabled with the possibility of using external sources of computation.
- We present the DLP$^T$ framework, which introduces the template paradigm into DLP, providing syntax and giving a clear operational semantics through a proper ''explosion'' pseudo-algorithm.
- We discuss theoretical properties of DLP$^T$ and its main advantages.
- We present an implementation of the DLP$^T$ language on top of the DLV system.

# Organization of the Thesis

This thesis consists of three parts. First, we introduce Disjunctive Logic Programming, and then discuss points (i) and (ii) above. The organization of the thesis is described more in details as follows.

[ I ] The first part introduces Disjunctive Logic Programming under the answer sets semantics, analyzes its computational complexity and addresses some knowledge representation issues.

[ II ] The second part describes the main steps of the computational process performed by DLP systems, and the DLV system in particular, focusing on search space pruning, and then provides new techniques aimed at improving the overall efficiency.

[ III ] Finally, the third part presents two extensions of the DLP, namely *template predicates* and *external predicates*, and the implementation of their support into the DLV system.

# Part I

# Disjunctive Logic Programming

In this part we present Disjunctive Logic Programming[1]. In particular, we first define the syntax of this language and its associated semantics, the *Answer Set Semantics*. This programming framework is also referred to as Answer Set Programming (ASP). Then, we analyze the computation complexity of this language and we illustrate the usage of Disjunctive Logic Programming for knowledge representation and reasoning.

The part is organized as follows:

- Chapter 1 provides a formal definition of the syntax and the semantics of Disjunctive Logic Programming[2].
- In Chapter 2 we give a detailed analysis of the computational complexity of disjunctive logic programs. We recall the main decisional problems arising in the context of DLP and discuss their computational complexity.
- Finally, in Chapter 3, we illustrate the usage of this language for knowledge representation and reasoning, describing a new declarative programming methodology which allows one to encode complex problems (up to $\Delta_3^P$-complete problems) in a declarative fashion.

[1]From now on, when talking about DLP we actually refer to a rather recent extension of DLP itself by *weak constraints* [Buccafurri *et al.*, 2000] which are a powerful tool to express optimization problems.

[2]The semantics presented here is a slight generalization of the original semantics proposed for weak constraints in [Buccafurri *et al.*, 2000]. In particular, the original definition of weak constraints allowed only "prioritized weak constraints", while here we allow both priority levels (layers) and weights for weak constraints.

# Chapter 1

# The Language

In this chapter, we provide a formal definition of the syntax and the semantics of Disjunctive Logic Programming. For further background, see [Lobo *et al.*, 1992; Eiter *et al.*, 1997c; Gelfond and Lifschitz, 1991].

## 1.1 Syntax

A *term* is either a variable or a constant. An *atom* is an expression $p(t_1, \ldots, t_n)$, where $p$ is a *predicate* of arity $n$ and $t_1, \ldots, t_n$ are terms. A *literal* is a *positive literal* $p$ or a *negative literal* $\text{not } p$, where $p$ is an atom.

A *disjunctive rule* (*rule*, for short) $r$ is a formula

$$a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \text{not } b_{k+1}, \cdots, \ \text{not } b_m. \tag{1.1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n$ is called *head* of $r$, while the conjunction $b_1, \cdots, b_k$, $\text{not } b_{k+1}, \cdots, \ \text{not } b_m$ is the *body* of $r$. We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set of the body literals. In particular, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and we usually omit the " :– " sign.

An *(integrity) constraint* is a rule without head literals (i.e. $n = 0$)

$$\texttt{:-} \ b_1, \cdots, b_k, \ \text{not } b_{k+1}, \cdots, \ \text{not } b_m. \tag{1.2}$$

A weak constraint $wc$ is an expression of the form

$$\texttt{:}\sim \ b_1, \ldots, b_k, \ \text{not } b_{k+1}, \ldots, \ \text{not } b_m. \ [w : l] \tag{1.3}$$

where for $m \geq k \geq 0$, $b_1, \ldots, b_m$ are atoms, while $w$ (the *weight*) and $l$ (the *level*, or *layer*) are positive integer constants or variables. For convenience, $w$ and/or $l$ might be omitted and are set to 1 in this case.

The sets $B(wc)$, $B^+(wc)$, and $B^-(wc)$ of a weak constraint $wc$ are defined in the same way as for integrity constraints.

A *disjunctive logic program* (often simply DLP program) $\mathcal{P}$ is a finite set of rules (possibly including integrity constraints) and weak constraints. $WC(\mathcal{P})$ denotes the set of weak constraints in $\mathcal{P}$, and $Rules(\mathcal{P})$ denotes the set of rules (including integrity constraints) in $\mathcal{P}$. A $\mathrm{not}$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*, and a $\mathtt{v}$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *normal logic program*. A program that does not contain weak constraints (i.e., such that $WC(\mathcal{P})$=$\emptyset$) is called *regular*.

A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe, if each of its rules is safe, and in the following we will only consider safe programs.

A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. A ground program is also called a *propositional* program.

## 1.2 Semantics

The semantics provided in this section extends the Answer Set Semantics of regular disjunctive logic programs, originally defined in [Gelfond and Lifschitz, 1991], to deal with weak constraints.

Let $\mathcal{P}$ be a disjunctive logic program. The Herbrand Universe of $\mathcal{P}$, denoted as $U_\mathcal{P}$, is the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant $\psi$ is added to $U_\mathcal{P}$. The Herbrand Base of $\mathcal{P}$, denoted as $B_\mathcal{P}$, is the set of all ground atoms constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$.

For any rule $r$, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions $\sigma$ from the variables in $r$ to elements of $U_\mathcal{P}$. In a similar way, given a weak constraint $w$, $Ground(w)$ denotes the set of weak constraints obtained by applying all possible substitutions $\sigma$ from the variables in $w$ to elements of $U_\mathcal{P}$. For any program $\mathcal{P}$, the ground instantiation $Ground(\mathcal{P})$ is the set $GroundRules(\mathcal{P}) \cup GroundWC(\mathcal{P})$, where

$$GroundRules(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r)$$

and

$$GroundWC(\mathcal{P}) = \bigcup_{w \in WC(\mathcal{P})} Ground(w).$$

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

**Answer Sets** For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in three steps:

First we define the answer sets of positive regular disjunctive logic programs, then we give a reduction of disjunctive logic programs containing negation to positive ones and use it to define answer sets of arbitrary disjunctive logic programs. Finally, we specify the way how weak constraints affect the semantics, defining the semantics of general DLP programs.

An interpretation $I$ is a set of ground atoms, i.e. $I \subseteq B_{\mathcal{P}}$ w.r.t. a program $\mathcal{P}$. An interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive logic program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation $X \subseteq B_{\mathcal{P}}$ is an *answer set* for a positive disjunctive logic program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.

**Example 1.1** The positive program $\mathcal{P}_1 = \{a \vee b \vee c.\}$ has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \vee b \vee c. \ ; \ :\!- a.\}$ has the answer sets $\{b\}$ and $\{c\}$. Finally, the positive program $\mathcal{P}_3 = \{a \vee b \vee c. \ ; \ :\!- a. \ ; \ b :\!- c. \ ; \ c :\!- b.\}$ has the single answer set $\{b, c\}$.

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

- deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;
- 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 $Ground(\mathcal{P})^X$.

**Example 1.2** Given the general program $\mathcal{P}_4 = \{ \ a \vee b :\!- c. \ ; \ \ b :\!- \text{not } a, \text{not } c. \ ; \ a \vee c :\!- \text{not } b.\}$ and $I = \{b\}$, the reduct $\mathcal{P}_4^I$ is $\{ \ a \vee b :\!- c. \ ; \ \ b. \ \}$. 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 $\mathcal{P}_4$.

Now consider $J = \{a\}$. The reduct $\mathcal{P}_4^J$ is $\{a \vee b :\!- c. \ ; \ a \vee c.\}$ 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 $\mathcal{P}_4$.

If, on the other hand, we take $K = \{c\}$, 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 $r : a \vee b \text{:-} c$, $B(r) \subseteq K$ holds, but $H(r) \cap K \neq \emptyset$ does not. Indeed, it can be verified that $I$ and $J$ are the only answer sets of $\mathcal{P}_4$.

Given a ground program $\mathcal{P}$ with weak constraints $WC(\mathcal{P})$, we are interested in the answer sets of $Rules(\mathcal{P})$ which minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level,[1] and among them those which minimize the sum of weights of the violated weak constraints in the next lower level, etc. Formally, this is expressed by an objective function $H^{\mathcal{P}}(A)$ for $\mathcal{P}$ and an answer set $A$ as follows, using an auxiliary function $f_{\mathcal{P}}$ which maps leveled weights to weights without levels:

$$\begin{aligned} f_{\mathcal{P}}(1) &= 1, \\ f_{\mathcal{P}}(n) &= f_{\mathcal{P}}(n-1) \cdot |WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1, \quad n > 1, \\ H^{\mathcal{P}}(A) &= \textstyle\sum_{i=1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(i) \cdot \sum_{w \in N_i^{\mathcal{P}}(A)} weight(w)), \end{aligned}$$

where $w^{\mathcal{P}}_{max}$ and $l^{\mathcal{P}}_{max}$ denote the maximum weight and maximum level over the weak constraints in $\mathcal{P}$, respectively; $N_i^{\mathcal{P}}(A)$ denotes the set of the weak constraints in level $i$ that are violated by $A$, and $weight(w)$ denotes the weight of the weak constraint $w$. Note that $|WC(\mathcal{P})| \cdot w^{\mathcal{P}}_{max} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than the sum of weights of any single level.

Intuitively, the function $f_{\mathcal{P}}$ handles priority levels. It guarantees that the violation of a single constraint of priority level $i$ is more "expensive" then the violation of *all* weak constraints of the lower levels (i.e., all levels $< i$).

For a DLP program $\mathcal{P}$ (possibly with weak constraints), a set $A$ is an *(optimal) answer set* of $\mathcal{P}$ if and only if (1) $A$ is an answer set of $Rules(\mathcal{P})$ and (2) $H^{\mathcal{P}}(A)$ is minimal over all the answer sets of $Rules(\mathcal{P})$.

**Example 1.3** Consider the following program $\mathcal{P}_{wc}$, which has three weak constraints:

$$\begin{aligned} &a \vee b. \\ &b \vee c. \\ &d \vee e \text{ :- } a, c. \end{aligned}$$

[1] Higher values for weights and priority levels mark weak constraints of higher importance. E.g., the most important constraints are those having the highest weight among those with the highest priority level.

$$
\begin{aligned}
&:\!- d, e.\\
&:\!\sim b.\ \ [1:2]\\
&:\!\sim a, e.\ \ [4:1]\\
&:\!\sim c, d.\ \ [3:1]
\end{aligned}
$$

$Rules(\mathcal{P}_{wc})$ admits three answer sets: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, e\}$, and $A_3 = \{b\}$. We have: $H^{\mathcal{P}_{wc}}(A_1) = 3$, $H^{\mathcal{P}_{wc}}(A_2) = 4$, $H^{\mathcal{P}_{wc}}(A_3) = 13$. Thus, the unique (optimal) answer set is $\{a, c, d\}$ with weight 3 in level 1 and weight 0 in level 2.

# Chapter 2

# Computational Complexity

In this chapter, we analyze the computational complexity of Disjunctive Logic Programming. We first provide some preliminaries on complexity theory. Then, we define a couple of relevant syntactic properties of disjunctive logic programs, which allow us to single out computationally simpler subclasses of the language. Finally, we define the main computational problems under consideration and illustrate their precise complexity.

## 2.1 Preliminaries

We assume here that the reader is familiar with the concepts of NP-completeness and complexity theory and provide only a very short reminder of the complexity classes of the Polynomial Hierarchy which are relevant to this chapter. For further details, the reader is referred to [Papadimitriou, 1994].

The classes $\Sigma_k^P$, $\Pi_k^P$, and $\Delta_k^P$ of the *Polynomial Hierarchy* (PH, cf. [Johnson, 1990]) are defined as follows:

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathrm{P}$$

$$\text{and for all } k \geq 1,\ \Delta_k^P = \mathrm{P}^{\Sigma_{k-1}^P}, \Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}, \Pi_k^P = \text{co-}\Sigma_k^P,$$

where $\mathrm{NP}^C$ denotes the class of decision problems that are solvable in polynomial time on a nondeterministic Turing machine with an oracle for any decision problem $\pi$ in the class $C$. In particular, NP $= \Sigma_1^P$, co-NP $= \Pi_1^P$, and $\Delta_2^P = \mathrm{P}^{\mathrm{NP}}$.

The oracle replies to a query in unit time, and thus, roughly speaking, models a call to a subroutine for $\pi$ that is evaluated in unit time.

Observe that for all $k \geq 1$,

$$\Sigma_k^P \ \subseteq\ \Delta_{k+1}^P \ \subseteq\ \Sigma_{k+1}^P \ \subseteq\ \mathrm{PSPACE}$$

where each inclusion is widely conjectured to be strict. By the rightmost inclusion above, all these classes contain only problems that are solvable in polynomial space. They allow, however, a finer grained distinction among NP-hard problems that are in PSPACE.

## 2.2 Relevant Classes of Programs

In this section, we introduce syntactic classes of disjunctive logic programs with interesting properties. First we need the following:

**Definition 2.1** Functions $||\ ||\ : B_{\mathcal{P}} \rightarrow \{0, 1, \ldots\}$ from the Herbrand Base $B_{\mathcal{P}}$ to finite ordinals are called *level mappings* of $\mathcal{P}$.

Level mappings give us a useful technique for describing various classes of programs.

**Definition 2.2** A disjunctive logic program $\mathcal{P}$ is called *(locally) stratified* [Apt *et al.*, 1988; Przymusinski, 1988], if there is a level mapping $||\ ||_s$ of $\mathcal{P}$ such that, for every rule $r$ of $Ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_s \leq ||l'||_s$;
2. For any $l \in B^-(r)$, and for any $l' \in H(r)$, $||l||_s < ||l'||_s$.
3. For any $l, l' \in H(r)$, $||l||_s = ||l'||_s$.

**Example 2.3** Consider the following two programs.

$$\begin{array}{lll} \mathcal{P}_1: & p(a) \vee p(c) \text{ :- not } q(a). & \\ & p(b) \text{ :- not } q(b). & \end{array} \qquad \begin{array}{lll} \mathcal{P}_2: & p(a) \vee p(c) \text{ :- not } q(b). & \\ & q(b) \text{ :- not } p(a). & \end{array}$$

It is easy to see that program $\mathcal{P}_1$ is stratified, while program $\mathcal{P}_2$ is not. A suitable level mapping for $\mathcal{P}_1$ is the following:

$$\begin{array}{lll} ||p(a)||_s = 2 & ||p(b)||_s = 2 & ||p(c)||_s = 2 \\ ||q(a)||_s = 1 & ||q(b)||_s = 1 & ||q(c)||_s = 1 \end{array}$$

As for $\mathcal{P}_2$, an admissible level mapping would need to satisfy $||p(a)||_s < ||q(b)||_s$ and $||q(b)||_s < ||p(a)||_s$, which is impossible.

Another interesting class of problems consists of head-cycle free programs.

**Definition 2.4** A program $\mathcal{P}$ is called *head-cycle free (HCF)* [Ben-Eliyahu and Dechter, 1994], if there is a level mapping $||\ ||_h$ of $\mathcal{P}$ such that, for every rule $r$ of $Ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_h \leq ||l'||_h$;
2. For any pair $l, l' \in H(r)$ $||l||_h \neq ||l'||_h$.

**Example 2.5** Consider the following program $\mathcal{P}_3$.

$$\begin{array}{ll} \mathcal{P}_3: & a \vee b. \\ & a \text{ :- } b. \end{array}$$

It is easy to see that $\mathcal{P}_3$ is head-cycle free; an admissible level mapping for $\mathcal{P}_3$ is given by $||a||_h = 2$ and $||b||_h = 1$. Consider now the program

$$\mathcal{P}_4 = \mathcal{P}_3 \cup \{b \text{ :- } a.\}$$

$\mathcal{P}_4$ is not head-cycle free, since $a$ and $b$ should belong to the same level by Condition (1) of Definition 2.4, while they cannot by Condition (2) of that definition. Note, however, that $\mathcal{P}_4$ is stratified.

## 2.3 Main Problems Considered

Three important decision problems, corresponding to three different reasoning tasks, arise in the context of Disjunctive Logic Programming:

> **Brave Reasoning**. Given a program $\mathcal{P}$, and a ground atom $A$, decide whether $A$ is true in some answer set of $\mathcal{P}$ (denoted $\mathcal{P} \models_b A$).
>
> **Cautious Reasoning**. Given a program $\mathcal{P}$, and a ground atom $A$, decide whether $A$ is true in all answer sets of $\mathcal{P}$ (denoted $\mathcal{P} \models_c A$).
>
> **Answer Set Checking.** Given a program $\mathcal{P}$, and a set $M$ of ground literals as input, decide whether $M$ is an answer set of $\mathcal{P}$.

We study the complexity of these decision problems for ground (i.e., propositional) DLP programs; we shall address the case of non-ground programs at the end of this chapter.

An interesting issue is the impact of syntactic restrictions on the logic program $\mathcal{P}$. Starting from normal positive programs (without negation and disjunction), we consider the effect of allowing the (combined) use of the following constructs:

- stratified negation ($\mathrm{not}_s$),
- arbitrary negation ($\mathrm{not}$),
- head-cycle free disjunction ( $\mathtt{v_h}$ ),
- arbitrary disjunction ( $\mathtt{v}$ ),
- weak constraints ($w$).[1]

Given a set $X$ of the above syntactic elements (with at most one negation and at most one disjunction symbol in $X$), we denote by DLP[$X$] the fragment of DLP where the elements in $X$ are allowed. For instance, DLP[$\mathtt{v_h}, \mathrm{not}_s$] denotes the fragment allowing head-cycle free disjunction and stratified negation, but no weak constraints.

## 2.4 Complexity Results and Discussion

We report here, with the help of some tables, results proved in [Eiter *et al.*, 1997c; Gottlob, 1994; Buccafurri *et al.*, 2000; Eiter *et al.*, 1998; Eiter and Gottlob, 1995; Perri, 2004; Leone *et al.*, 2005].

It is worth that we consider the ground case, i.e., we assume that programs and, unless stated otherwise, also atoms, literals etc. are ground. Furthermore, for the sake of the presentations, we disregard integrity constraints in programs. However, this is not significant since the results in presence of these constructs are the same (see, e.g., [Buccafurri *et al.*, 2000]). Some remarks on the complexity and expressiveness of non-ground programs are then provided.

The complexity of Brave Reasoning and Cautious Reasoning from ground DLP programs are summarized in Table 2.1 and Table 2.2, respectively. In Table 2.3, we report the results on the complexity of Answer Set Checking.

The rows of the tables specify the form of disjunction allowed; in particular, $\{\}$ = no disjunction, $\{\mathtt{v_h}\}$ = head-cycle free disjunction, and $\{\mathtt{v}\}$ = unrestricted (possibly not head-cycle free) disjunction. The columns specify the support for negation and weak constraints. For instance, $\{w, \mathrm{not}_s\}$ denotes weak constraints

[1] Following [Buccafurri *et al.*, 2000], possible restrictions on the support of negation affect $Rules(\mathcal{P})$, that is, the rules (including the integrity constraints) of the program, while weak constraints, if allowed, can freely contain both positive and negative literals in any fragment of DLP we consider.

| | $\{\}$ | $\{\mathbf{w}\}$ | $\{\mathrm{not_s}\}$ | $\{\mathrm{not_s}, \mathbf{w}\}$ | $\{\mathrm{not}\}$ | $\{\mathrm{not}, \mathbf{w}\}$ |
|---|---|---|---|---|---|---|
| $\{\}$ | P | P | P | P | NP | $\Delta_2^P$ |
| $\{\mathtt{v_h}\}$ | NP | $\Delta_2^P$ | NP | $\Delta_2^P$ | NP | $\Delta_2^P$ |
| $\{\mathtt{v}\}$ | $\Sigma_2^P$ | $\Delta_3^P$ | $\Sigma_2^P$ | $\Delta_3^P$ | $\Sigma_2^P$ | $\Delta_3^P$ |

Table 2.1: The Complexity of Brave Reasoning in fragments of DLP

| | $\{\}$ | $\{\mathbf{w}\}$ | $\{\mathrm{not_s}\}$ | $\{\mathrm{not_s}, \mathbf{w}\}$ | $\{\mathrm{not}\}$ | $\{\mathrm{not}, \mathbf{w}\}$ |
|---|---|---|---|---|---|---|
| $\{\}$ | P | P | P | P | co-NP | $\Delta_2^P$ |
| $\{\mathtt{v_h}\}$ | co-NP | $\Delta_2^P$ | co-NP | $\Delta_2^P$ | co-NP | $\Delta_2^P$ |
| $\{\mathtt{v}\}$ | co-NP | $\Delta_3^P$ | $\Pi_2^P$ | $\Delta_3^P$ | $\Pi_2^P$ | $\Delta_3^P$ |

Table 2.2: The Complexity of Cautious Reasoning in fragments of DLP

and stratified negation. Each entry of the table provides the complexity of the corresponding fragment of the language, in terms of a completeness result. For instance, $(\{\mathtt{v_h}\}, \{\mathrm{not}_s\})$ is the fragment allowing head-cycle free disjunction and stratified negation, but no weak constraints. The corresponding entry in Table 2.1, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under polynomial time (and in fact LOGSPACE) reductions. All results have either been proved in [Perri, 2004] or emerge from [Eiter *et al.*, 1997c; Gottlob, 1994; Eiter *et al.*, 1998; Eiter and Gottlob, 1995; Buccafurri *et al.*, 2000]. Note that the presence of weights besides priority levels in weak constraints does not increase the complexity of the language, and thus the complexity results reported in [Buccafurri *et al.*, 2000] remain valid also for our more general language. Furthermore, not all complexity results in the quoted papers were explicitly stated for LOGSPACE reductions, but can be easily seen to hold from (suitably adapted) proofs.

Looking at Table 2.1, we see that limiting the form of disjunction and negation reduces the respective complexity. For disjunction-free programs, brave reasoning is polynomial on stratified negation, while it becomes NP-complete if we allow unrestricted (nonmonotonic) negation. Brave reasoning is NP-complete on head-cycle free programs even if no form of negation is allowed. The complexity jumps one level higher in the Polynomial Hierarchy, up to $\Sigma_2^P$-complexity, if

| | $\{\}$ | $\{\mathbf{w}\}$ | $\{\mathrm{not_s}\}$ | $\{\mathrm{not_s}, \mathbf{w}\}$ | $\{\mathrm{not}\}$ | $\{\mathrm{not}, \mathbf{w}\}$ |
|---|---|---|---|---|---|---|
| $\{\}$ | P | P | P | P | P | co-NP |
| $\{\mathtt{v_h}\}$ | P | co-NP | P | co-NP | P | co-NP |
| $\{\mathtt{v}\}$ | co-NP | $\Pi_2^P$ | co-NP | $\Pi_2^P$ | co-NP | $\Pi_2^P$ |

Table 2.3: The Complexity of Answer Set Checking in fragments of DLP

full disjunction is allowed. Thus, disjunction seems to be harder than negation, since the full complexity is reached already on positive programs, even without any kind of negation. Weak constraints are irrelevant, from the complexity viewpoint, if the program has at most one answer set (if there is no disjunction and negation is stratified). On programs with multiple answer sets, weak constraints increase the complexity of reasoning moderately, from NP and $\Sigma_2^P$ to $\Delta_2^P$ and $\Delta_3^P$, respectively.

Table 2.2 contains results for cautious reasoning. One would expect its complexity to be symmetric to the complexity of brave reasoning, that is, whenever the complexity of a fragment is $C$ under brave reasoning, one expects its complexity to be co-$C$ under cautious reasoning (recall that co-$\mathrm{P} = \mathrm{P}$, co-$\Delta_2^P = \Delta_2^P$, co-$\Sigma_2^P = \Pi_2^P$, and co-$\Delta_3^P = \Delta_3^P$).

Surprisingly, there is one exception: while full disjunction raises the complexity of brave reasoning from NP to $\Sigma_2^P$, full disjunction alone is not sufficient to raise the complexity of cautious reasoning from co-NP to $\Pi_2^P$. Cautious reasoning remains in co-NP if default negation is disallowed. Intuitively, to disprove that an atom $A$ is a cautious consequence of a program $\mathcal{P}$, it is sufficient to find *any model* $M$ of $\mathcal{P}$ (which need not be an answer set or a minimal model) which does not contain $A$. For $\mathrm{not}$-free programs, the existence of such a model guarantees the existence of a subset of $M$ which is an answer set of $\mathcal{P}$ (and does not contain $A$).

The complexity results for Answer Set Checking, reported in Table 2.3, help us to understand the complexity of reasoning. Whenever Answer Set Checking for weak constraint-free programs is co-NP-complete for a fragment $F$, the complexity of brave reasoning jumps up to the second level of the Polynomial Hierarchy ($\Sigma_2^P$). In contrast, co-NP-completeness for Answer Set Checking involving weak constraints causes only a modest increase for brave reasoning, which stays within the same level ($\Delta_2^P$). Indeed, brave reasoning on full DLP programs suffers from

three sources of complexity:

($s_1$) the exponential number of answer set "candidates",

($s_2$) the difficulty of checking whether a candidate $M$ is an answer set (the minimality of $M$ can be disproved by an exponential number of subsets of $M$), and

($s_3$) the difficulty of determining the optimality of the answer set w.r.t. the violation of the weak constraints.

Now, disjunction (unrestricted or even head-cycle free) or unrestricted negation preserve the existence of source ($s_1$), while source ($s_2$) exists only if full disjunction is allowed (see Table 2.3). Source ($s_3$) depends on the presence of weak constraints, but it is effective only in case of multiple answer sets (i.e., only if source ($s_1$) is present), otherwise it is irrelevant. As a consequence, e.g., the complexity of brave reasoning is the highest ($\Delta_3^P$) on the fragments preserving all three sources of complexity (where both full disjunction and weak constraints are allowed). Eliminating weak constraints (source ($s_3$)) from the full language, decreases the complexity to $\Sigma_2^P$. The complexity goes down to the first level of PH if source ($s_2$) is eliminated, and is in the class $\Delta_2^P$ or NP depending on the presence or absence of weak constraints (source ($s_3$)). Finally, avoiding source ($s_1$) the complexity falls down to P, as ($s_2$) is automatically eliminated, and ($s_3$) becomes irrelevant.

We close this section with briefly addressing the complexity and expressiveness of non-ground programs. A non-ground program $\mathcal{P}$ can be reduced, by naive instantiation, to a ground instance of the problem. The complexity of this ground instantiation is as described above. In the general case, where $\mathcal{P}$ is given in the input, the size of the grounding $Ground(\mathcal{P})$ is single exponential in the size of $\mathcal{P}$. Informally, the complexity of Brave Reasoning and Cautious Reasoning increases accordingly by one exponential, from P to EXPTIME, NP to NEXPTIME, $\Delta_2^P$ to EXPTIME$^{\text{NP}}$, $\Sigma_2^P$ to NEXPTIME$^{\text{NP}}$, etc. For disjunctive programs and certain fragments of DLP, complexity results in the non-ground case have been derived e.g. in [Eiter *et al.*, 1997c; 1998]. For the other fragments, the results can be derived using complexity upgrading techniques [Eiter *et al.*, 1997c; Gottlob *et al.*, 1999]. Answer Set Checking, however, increases exponentially up to co-NEXPTIME$^{\text{NP}}$ only in the presence of weak constraints, while it stays in PH if no weak constraints occur. The reason is that in the latter case, the conditions

of an answer set can be checked using small guesses, and no alternative (perhaps exponentially larger) answer set candidates need to be considered.

# Chapter 3

# Knowledge Representation

In this chapter, we illustrate the usage of Disjunctive Logic Programming for knowledge representation and reasoning. We first present a new programming methodology, which allows us to encode search problems in a simple and highly declarative fashion; even optimization problems of complexity up to $\Delta_3^P$ can be declaratively encoded using this methodology. Then, we illustrate this methodology on a number of computationally hard problems.

## 3.1 The GCO Declarative Programming Methodology

Disjunctive Logic Programming can be used to encode problems in a highly declarative fashion, following a Guess/Check/Optimize (**GCO**) paradigm, which is an extension and refinement of the "Guess&Check" methodology in [Eiter *et al.*, 2000]. In this section, we will first describe the **GCO** technique and we will then illustrate how to apply it on a number of examples. Many problems, also problems of comparatively high computational complexity ($\Sigma_2^P$-complete and $\Delta_3^P$-complete problems), can be solved in a natural manner by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so in a uniform way over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance $I$ of some problem $\mathbf{P}$, a **GCO** program $\mathcal{P}$ for $\mathbf{P}$ consists of the following three main parts:

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

**Checking Part** The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance $I$.

**Optimization Part** The (optional) optimization part $\mathcal{O} \subseteq \mathcal{P}$ of the program allows to express a quantitative cost evaluation of solutions by using weak constraints. It implicitly defines an objective function $f : \mathcal{AS}(\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I) \rightarrow \mathbb{N}$ mapping the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ to natural numbers. The semantics of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ optimizes $f$ by filtering those answer sets having the minimum value; this way, the optimal (least cost) solutions are computed.

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., checking is completely integrated into the guessing part such that solution candidates are always solutions. Also, 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 further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

In general, both $\mathcal{G}$ and $\mathcal{C}$ may be arbitrary collections of rules (and, for the optimization part, weak constraints), and it depends on the complexity of the problem at hand which kinds of rules are needed to realize these parts (in particular, the checking part).

## Problems in NP and $\Delta_2^P$

For problems with complexity in NP or, in case of optimization problems, $\Delta_2^P$, often a natural **GCO** program can be designed with the three parts clearly separated into the following simple layered structure:

- The guessing part $\mathcal{G}$ consists of disjunctive rules that "guess" a solution candidate $S$.
- The checking part $\mathcal{C}$ consists of integrity constraints that check the admissibility of $S$.
- The optimization part $\mathcal{O}$ consists of weak constraints.

Each layer may have further auxiliary predicates, defined by normal stratified rules (see Section 2.2 for a definition of stratification), for local computations.

The disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches. The weak constraints in $\mathcal{O}$ induce a modular ordering on the answer sets, allowing the user to specify the best solutions according to an optimization function $f$.

## Problems beyond $\Delta_2^P$

For problems which are beyond $\Delta_2^P$, and in particular for $\Sigma_2^P$-complete problems, the layered program schema above no longer applies. If $\mathcal{G}$ has complexity in NP, which is the case if disjunction is just used for making the guess and the layer is head-cycle free [Ben-Eliyahu and Dechter, 1994], then an answer set $A$ of $\mathcal{G} \cup \mathcal{F}_I$ can be guessed in polynomial time, i.e., nondeterministically created with a polynomial number of steps. Hence, it can be shown easily that computing an answer set of the whole program, $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$, is feasible in polynomial time with an NP oracle. Thus, applicability of the same schema to $\Sigma_2^P$-hard problems would imply $\Sigma_2^P \subseteq \Delta_2^P$, which is widely believed to be false.

Until now we tacitly assumed an intuitive layering of the program parts, such that the checking part $\mathcal{C}$ has no "influence" or "feedback" 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). This can be formalized in terms of a "potentially uses" relation [Eiter *et al.*, 1997c] or a "splitting set" condition [Lifschitz and Turner, 1994]. Complexity-wise, this can be relaxed to the property that the union of the program parts is head-cycle free.

In summary, if a program solves a $\Sigma_2^P$-complete problem and has guessing and checking parts $\mathcal{G}$ and $\mathcal{C}$, respectively, with complexities below $\Sigma_2^P$, then $\mathcal{C}$ must either contain disjunctive rules or interfere with $\mathcal{G}$ (and in particular head-cycles must be present in $\mathcal{G} \cup \mathcal{C}$).

We close this section with remarking that the **GCO** programming methodology has also positive implications from the Software Engineering viewpoint. Indeed, the modular program structure in **GCO** allows for developing programs incrementally, which is helpful to simplify testing and debugging. One can start by writing the guessing part $\mathcal{G}$ and testing that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one adds the checking part and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ encode the admissible solutions. Finally, one tests that $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ generates the optimal solutions of the problem at hand.

## 3.2 Applications of the GCO Programming Technique

In this section, we illustrate the declarative programming methodology described in Section 3.1 by showing its application on a number of concrete examples.

### 3.2.1 Exams Scheduling

Let us start by a simple scheduling problem. Here we have to schedule the exams for several university courses in three time slots $t_1$, $t_2$, and $t_3$ at the end of the semester. In other words, each course should be assigned exactly one of these three time slots. Specific instances $I$ of this problem are provided by sets $\mathcal{F}_I$ of facts specifying the exams to be scheduled. The predicate $exam$ has four arguments representing, respectively, the *identifier* of the exam, the *professor* who is responsible for the exam, the *curriculum* to which the exam belongs, and the *year* in which the exam has to be given in the curriculum.

Several exams can be assigned to the same time slot (the number of available rooms is sufficiently high), but the scheduling has to respect the following specifications:

$S1$ Two exams given by the same professor cannot run in parallel, i.e., in the same time slot.

$S2$ Exams of the same curriculum should be assigned to different time slots, if possible. If $S2$ is unsatisfiable for all exams of a curriculum $C$, one should:

($S2_1$) first of all, minimize the overlap between exams of the same year of $C$,

($S2_2$) then, minimize the overlap between exams of different years of $C$.

This problem can be encoded in DLP by the following **GCO** program $\mathcal{P}_{sch}$:

$$assign(Id, t_1) \texttt{ v } assign(Id, t_2) \texttt{ v } assign(Id, t_3) \texttt{ :- } exam(Id, P, C, Y). \quad \Big\} \textbf{ Guess}$$

$$\texttt{:- } assign(Id, T), assign(Id', T), Id <> Id', exam(Id, P, C, Y), exam(Id', P, C', Y'). \quad \Big\} \textbf{ Check}$$

$$\texttt{:}\sim assign(Id, T), assign(Id', T) \; exam(Id, P, C, Y), exam(Id', P', C, Y), Id <> Id'. \; [: 2]$$

$$\texttt{:}\sim assign(Id, T), assign(Id', T) \; exam(Id, P, C, Y), exam(Id', P', C, Y'), Y <> Y'. \; [: 1] \quad \Big\} \textbf{ Optimize}$$

The guessing part $\mathcal{G}$ has a single disjunctive rule defining the search space. It is evident that the answer sets of $\mathcal{G} \cup \mathcal{F}_I$ are the possible assignments of exams to time slots.

The checking part $\mathcal{C}$ consists of one integrity constraint, discarding the assignments of the same time slot to two exams of the same professor. The answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ correspond precisely to the admissible solutions, that is, to all assignments which satisfy the constraint $S1$.

Finally, the optimizing part $\mathcal{O}$ consists of two weak constraints with different priorities. Both weak constraints state that exams of the same curriculum should *possibly not* be assigned to the same time slot. However, the first one, which has higher priority (level 2), states this desire for the exams of the curriculum of the same year, while the second one, which has lower priority (level 1) states it for the exams of the curriculum of different years. The semantics of weak constraints, as given in Section 1.2, implies that $\mathcal{O}$ captures precisely the constraints $S2$ of the scheduling problem specification. Thus, the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ correspond precisely to the desired schedules.

### 3.2.2 Hamiltonian Path

Let us now consider a classical NP-complete problem in graph theory, namely Hamiltonian Path.

**Definition 3.1 (HAMPATH)** Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in $G$ starting at $a$ and passing through each node in $V$ exactly once?

Suppose that the graph $G$ is specified by using facts over predicates $node$ (unary) and $arc$ (binary), and the starting node $a$ is specified by the predicate $start$ (unary). Then, the following **GCO** program $\mathcal{P}_{hp}$ solves the problem HAMPATH (no optimization part is needed here):

$$
\left.
\begin{array}{ll}
inPath(X,Y) \texttt{ v } outPath(X,Y) \texttt{ :- } start(X), arc(X,Y). & \\
inPath(X,Y) \texttt{ v } outPath(X,Y) \texttt{ :- } reached(X), arc(X,Y). & \\
reached(X) \texttt{ :- } inPath(Y,X). & \textbf{(aux.)}
\end{array}
\right\} \textbf{Guess}
$$

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

The two disjunctive rules guess a subset $S$ of the arcs to be in the path, while the rest of the program checks whether $S$ constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is used, which is associated with the guessed predicate $inPath$ using the last rule. Note that $reached$ is completely determined by the guess for $inPath$, and no further guessing is needed.

In turn, through the second rule, the predicate $reached$ influences the guess of $inPath$, which is made somehow inductively: Initially, a guess on an arc leaving the starting node is made by the first rule, followed by repeated guesses of arcs leaving from reached nodes by the second rule, until all reached nodes have been handled.

In the checking part, the first two constraints ensure that the set of arcs $S$ selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by $S$. A less sophisticated encoding can be obtained by replacing the guessing part with the single rule

$$inPath(X, Y) \texttt{ v } outPath(X, Y) \texttt{ :- } arc(X, Y).$$

that guesses for each arc whether it is in the path and by defining the predicate $reached$ in the checking part by rules

$$reached(X) \texttt{ :- } start(X).$$
$$reached(X) \texttt{ :- } reached(Y), inPath(Y, X).$$

However, this encoding is less preferable from a computational point of view, because it leads to a larger search space.

It is easy to see that any set of arcs $S$ which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in $G$ that starts at node $v_0 = a$, 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 in fact a Hamiltonian Cycle (from which a Hamiltonian path can be immediately computed, by dropping the last arc).

Thus, given a set of facts $\mathcal{F}$ for $node$, $arc$, and $start$, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the corresponding graph has a Hamiltonian Path. The above program correctly encodes the decision problem of deciding whether a given graph admits a Hamiltonian Path or not.

This encoding is very flexible, and can be easily adapted to solve the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result has to be a

tour, i.e., a closed path). If we want to be sure that the computed result is an *open* path (i.e., it is not a cycle), we can easily impose openness by adding a further constraint $:\!- \, start(Y), inPath(\_, Y).$ to the program (like in Prolog, the symbol '_' stands for an anonymous variable whose value is of no interest). Then, the set $S$ of selected arcs in any answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at $a$.

## Hamiltonian Cycle

Let us show now another classical NP-complete problem which is very similar to the one of Section 3.2.2.

**Definition 3.2 (HAMCYCLE)** Given an undirected graph $G = (V, E)$, where $V$ is the set of vertices of $G$, and $E$ is the set of edges, and a node $a \in V$ of this graph, does there exist a cycle of $G$ containing $a$ and passing through each node in $V$ exactly once?

Assuming that the graph $G$ is specified by means of predicates $vertex$ (unary) and $arc$ (binary). Please note that predicate $arc$ is symmetric, since undirected edges are bidirectional directed arcs. The starting node is specified by the predicate $start$ (unary). The following program $\mathcal{P}_{HC}$ solves the problem HAMCYCLE:

$$
\left.\begin{array}{l} inCycle(X,Y) \texttt{ v } outCycle(X,Y) \texttt{ :- } start(X), arc(X,Y). \\ inCycle(X,Y) \texttt{ v } outCycle(X,Y) \texttt{ :- } onCycle(X), arc(X,Y). \end{array}\right\} \textbf{Guess}
$$

$$
\left.\begin{array}{l} onCycle(Y) \texttt{ :- } inCycle(\_, Y). \\ \% \ At\ most\ one\ ingoing/outgoing\ arc! \\ \texttt{:- } inCycle(X,Y), inCycle(X,Y1), Y <> Y1. \\ \texttt{:- } inCycle(X,Y), inCycle(X1,Y), X <> X1. \\ \% \ Each\ node\ has\ to\ be\ on\ the\ cycle. \\ \texttt{:- } vertex(X), \text{not } onCycle(X). \end{array}\right\} \textbf{Check}
$$

The guessing part (first two rules) guess a subset of all given arcs, while the rest of the program checks whether it is a Hamiltonian Cycle. The first two constraints in checking part ensure that in the set of arcs $S$ selected by `inCycle` there are not two arcs that start at the same node or end in the same node. The third constraint enforces that all nodes are reached from the starting node in the subgraph induced by $S$, and ensures that this subgraph is connected (since it is a cycle) through the auxiliary predicate `onCycle` defined by the third rule. Thus, given a

set of facts $F$ for $vertex$, $arc$, and $start$ which specify the problem input, the program $\mathcal{P}_{HC} \cup F$ has a stable model if and only if the input graph has a Hamiltonian Cycle.

It is worth noting that another solution to this problem could be easily obtained just stripping off the literal $\text{not } start(X)$ from the last constraint of the program presented in Section 3.2.2.

### 3.2.3 Ramsey Numbers

In the previous examples, we have seen how a search problem can be encoded in a DLP program whose answer sets correspond to the problem solutions. We next see another use of the **GCO** programming technique. We build a DLP program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the DLP program has no answer set. Such a programming scheme is useful to prove the validity of co-NP or $\Pi_2^P$ properties. We next apply the above programming scheme to a well-known problem of number and graph theory.

**Definition 3.3 (RAMSEY)** The Ramsey number $R(k, m)$ is the least integer $n$ such that, no matter how we color the arcs of the complete undirected graph (clique) with $n$ nodes using two colors, say red and blue, there is a red clique with $k$ nodes (a red $k$-clique) or a blue clique with $m$ nodes (a blue $m$-clique).

Ramsey numbers exist for all pairs of positive integers $k$ and $m$ [Radziszowski, 1994]. We next show a program $\mathcal{P}_{ramsey}$ that allows us to decide whether a given integer $n$ is <u>not</u> the Ramsey Number $R(3, 4)$. By varying the input number $n$, we can determine $R(3, 4)$, as described below. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with $n$ nodes. $\mathcal{P}_{ramsey}$ is the following **GCO** program:

$$\left.\begin{array}{l} blue(X,Y) \texttt{ v } red(X,Y) \text{ :- } arc(X,Y). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:- } red(X,Y),\ red(X,Z),\ red(Y,Z). \\ \text{:- } blue(X,Y),\ blue(X,Z),\ blue(Y,Z), \\ \quad blue(X,W),\ blue(Y,W),\ blue(Z,W). \end{array}\right\} \textbf{Check}$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint eliminates the colorings containing a blue clique

with 4 nodes. The program $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on $n$ nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular $n$, then $n$ is not $R(3,4)$, that is, $n < R(3,4)$. On the other hand, if $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has no answer set, then $n \geq R(3,4)$. Thus, the smallest $n$ such that no answer set is found is the Ramsey number $R(3,4)$.

The problems considered so far are at the first level of the Polynomial Hierarchy. We next show that also problems located at the second level of the Polynomial Hierarchy can be encoded by the **GCO** technique.

### 3.2.4 Strategic Companies

A problem located at the second level of the Polynomial Hierarchy is the following, which is known under the name *Strategic Companies* [Cadoli *et al.*, 1997].

**Definition 3.4 (STRATCOMP)** Suppose that we have $C = \{c_1, \ldots, c_m\}$, a collection of companies $c_i$ owned by a holding, a set $G = \{g_1, \ldots, g_n\}$ of goods, and for each $c_i$ we have a set $G_i \subseteq G$ of goods produced by $c_i$ and a set $O_i \subseteq C$ of companies controlling (owning) $c_i$. $O_i$ is referred to as the *controlling set* of $c_i$. 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. Note that, in general, a company might have more than one controlling set. Let the holding produce all goods in $G$, i.e. $G = \bigcup_{c_i \in C} G_i$.

A subset of the companies $C' \subseteq C$ 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 $C$.

This notion is relevant when companies should be sold. Indeed, intuitively, selling any non-strategic company does not reduce the economic power of the holding. Computing strategic companies is $\Sigma_2^P$-hard in general [Cadoli *et al.*, 1997]; reformulated as a decision problem ("Given a particular company $c$ in the input, is $c$ strategic?"), it is $\Sigma_2^P$-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 [Cadoli *et al.*, 1997] 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 $\Sigma_2^P$-hard). Assume that for a given instance of STRATCOMP, $\mathcal{F}$ contains the following facts:

- $company(c)$ for each $c \in C$,
- $prod\_by(g, c_j, c_k)$, if $\{c_i \mid g \in G_i\} = \{c_j, c_k\}$, where $c_j$ and $c_k$ may possibly coincide,
- $contr\_by(c_i, c_k, c_m, c_n)$, if $c_i \in C$ and $O_i = \{c_k, c_m, c_n\}$, where $c_k$, $c_m$, and $c_n$ are not necessarily distinct.

We next present a program $\mathcal{P}_{strat}$, which solves this hard problem elegantly by only two rules:

$$r_{s1}: \quad strat(Y) \mathtt{v}\ strat(Z) \text{:-} prod\_by(X, Y, Z). \qquad \} \textbf{ Guess}$$

$$r_{s2}: \quad \begin{array}{c} strat(W) \text{:-} contr\_by(W, X, Y, Z), strat(X), \\ strat(Y), strat(Z). \end{array} \qquad \left.\right\} \textbf{ Check}$$

Here $strat(X)$ means that company $X$ 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 $\mathcal{P}_{strat}$ is surprisingly succinct, given that STRATCOMP is a hard ($\Sigma_2^P$-hard) problem. To overcome the difficulty of the encoding, coming from the intrinsic high complexity of the STRATCOMP problem, we next explain this encoding more in-depth, compared with the previous **GCO** encodings.

The program $\mathcal{P}_{strat}$ 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 $c_1$ and $c_2$ that produce some item $g$, which is described by $prod\_by(g, c_1, c_2)$. If there were no company control information, minimality of answer sets would 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 control information is available, the rule $r_{s2}$ 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.

The answer sets of $\mathcal{P}_{strat} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$; a company $c$ is thus strategic iff $strat(c)$ is in some answer set of $\mathcal{P}_{strat} \cup \mathcal{F}$.

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 $r_{s1}$. For example, suppose the guessing part gives rise to a ground rule $r_{sg1}$

$$strat(c1) \ \mathtt{v} \ strat(c2) \text{:-} prod\_by(g, c1, c2).$$

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

By the complexity considerations made in Subsection 3.1, such interference is needed to solve STRATCOMP in the above way (without disjunctive rules in the Check part), since deciding whether a particular company is strategic is $\Sigma_2^P$-complete. If $\mathcal{P}_{strat}$ is rewritten to eliminate such interference and layer the parts hierarchically, then further disjunctive rules must be added. An encoding which expresses the strategic sets in the generic **GCO**-paradigm with clearly separated guessing and checking parts is given in [Eiter *et al.*, 2000].

Note that, the program above cannot be replaced by a simple normal (non-disjunctive) program. Intuitively, this is due to the fact that disjunction in the head of rules is not exclusive, while at the same time answer sets are subset-minimal. Using techniques like the ones in [Eiter *et al.*, 2003a], $\mathcal{P}_{strat}$ can be extended to support an arbitrary number of producers per product and controlling companies per company, respectively.

### Preferred Strategic Companies

Let us consider an extension of Strategic Companies which also deals with preferences. Suppose that the president of the holding desires, in case of options given by multiple strategic sets, to discard those where certain companies are sold or kept, respectively, by expressing preferences among possible solutions. For example, the president might give highest preference to discard solutions where

company $a$ is sold; next important to him is to avoid selling company $b$ while keeping $c$, and of equal importance to avoid selling company $d$, and so on.

In presence of such preferences, the STRATCOMP problem becomes slightly harder, as its complexity increases from $\Sigma_2^P$ to $\Delta_3^P$. Let us assume that the president's preferences are represented by a single predicate $avoid(c_{sell}, c_{keep}, pr)$, which intuitively states that selling $c_{sell}$ while keeping $c_{keep}$ should be avoided with priority $pr$; in the above example, the preferences would be $avoid(a, c_\top, top)$, $avoid(b, c, top{-}1)$, $avoid(d, c_\top, top{-}1)$, ..., where $c_\top$ is a dummy company belonging to every strategic set, and $top$ is the highest priority number. Then, we can easily represent this more complicated problem, by adding the following weak constraint to the original encoding for STRATCOMP:

$$:\sim avoid(Sell, Keep, Priority), \text{not } strat(Sell), strat(Keep). \; [: Priority]$$

The (optimal) answer sets of the resulting program then correspond to the solutions of the above problem.

# Part II

# Optimizing the Evaluation of Disjunctive Logic Programming

In this part we explain the computational process commonly performed by DLP systems, with a focus on search space pruning, which is crucial for the efficiency of such systems.

We present two suitable operators for pruning (*Fitting*'s and *Well-founded*), discuss their peculiarities and differences with respect to efficiency and effectiveness. We design an intelligent strategy for combining the two operators, exploiting the advantages of both. We implement our approach in the DLV system and perform some experiments. These experiments show interesting results, and evidence how the choice of the pruning operator affects the performance of DLP systems.

The part is structured as follows.

- In Chapter 4 we introduce the DLV system and give an overview of its architecture and implementation. The theoretical foundations of the implementation of DLV are also briefly discussed. The main procedure for the computation of the answer set semantics is then described.

- In Chapter 5 we present two pruning operators for DLP: *Fitting*'s ($\Phi_{\mathcal{P}}$) operator, and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator, and then we analyze several interesting properties of these pruning operators on some syntactically restricted classes of DLP programs.

- In Chapter 6, we design our new method for the intelligent combination of the pruning operators for DLP, and discuss some key issues for its implementation in DLV. We then report the results of our experimentation activity on a number of benchmark problems.

- Appendix A, at the end of the thesis, provides further details on the experiments and problem encodings.

# Chapter 4

# The DLV System

In this chapter, we describe the general architecture of DLV, and give an overview of the main techniques employed in the implementation.

## 4.1 The Architecture of DLV: an Overview

The system architecture of DLV is shown in Figure 4.1. The internal system language is the one described in Chapter 1, i.e. Disjunctive Logic Programming extended by weak constraints. The DLV Core (the shaded part of the figure) is an efficient engine for computing answer sets (one, some, or all) of its input. The DLV core has three layers (see Figure 4.1), each of which is a powerful subsystem per se: The *Intelligent Grounder* (IG, also *Instantiator*) has the power of a deductive database system; the *Model Generator* (MG) is as powerful as a Satisfiability Checker; and the *Model Checker* (MC) is capable of solving co-NP-complete problems. In addition to its kernel language, DLV provides a number of application front-ends that show the suitability of our formalism for solving various problems from the areas of Artificial Intelligence, Knowledge Representation and (Deductive) Databases. Currently, the DLV system has front-ends for inheritance reasoning [Buccafurri *et al.*, 2002], model-based diagnosis [Eiter *et al.*, 1997b], planning [Eiter *et al.*, 2003b], and SQL3 query processing. Each front-end maps its problem specific input into a DLV program, invokes the DLV kernel, and then post-processes any answer set returned, extracting from it the desired solution; furthermore, there is a Graphical User Interface (GUI) that provides convenient access to some of these front-ends as well as the system itself.

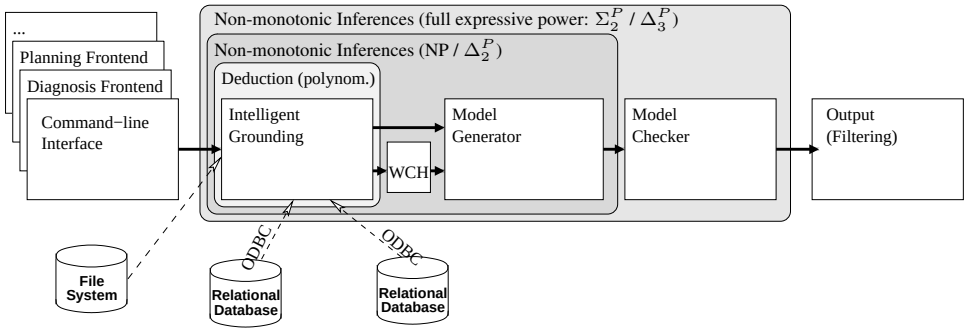


Figure 4.1: The System Architecture of DLV

## 4.2 Theoretical Foundations

The implementation of the DLV system is based on very solid theoretical foundations, and exploits the results on the computational complexity discussed in Chapter 2. Ideally, the performance of a system should reflect the complexity of the problem at hand, such that "easy" problems (say, those of polynomial complexity) are solved fast, while only harder problems involve methods of higher run-time cost. Indeed, the DLV system is designed according to this idea, and thrives to exploit the complexity results reported in Section 2.4.

For example, stratified normal programs (which have polynomial complexity, as reported in Table 2.1[1]) are evaluated solely using techniques from the field of deductive databases, without employing the more complex techniques which are needed to evaluate full DLV programs; in fact, such normal stratified programs are evaluated without generating the program instantiation at all.

The architecture of the DLV Core closely reflects complexity results for various subsets of our language. As mentioned before, the Intelligent Grounding (IG) module is able to completely solve some problems which are known to be of polynomial time complexity (like normal stratified programs); the Model Generator (together with the Grounding) is capable of solving NP-complete problems. Adding the Model Checker is needed to solve $\Sigma_2^P$-complete problems. The WCH (Weak Constraints Handler) comes into play only in presence of weak constraints. More precisely, referring to the notation of Section 2.2, we have the following five disjoint language classes $L_1$ – $L_5$ for evaluation:

- $L_1$ contains the programs included in the class $\langle\{\}, \{w, \mathrm{not}_s\}\rangle$, which all

[1]Note that the complexity of propositional DLV programs reported in Tables 2.1–2.3 coincides with the data complexity of non-ground DLV programs.

have polynomial complexity. They are completely evaluated by the IG module, which runs in polynomial time (referring to propositional complexity).

- $L_2$ contains the programs which are in the subclass corresponding to $\langle\{\ \mathtt{v_h}\ \}, \{\mathrm{not}\}\rangle$, but not in $L_1$. The complexity of this fragment is NP, and the programs are evaluated by the MG module (besides the IG) with only a call to the linear-time part of the MC module. Note that the MG implements a flat backtracking algorithm and is suitable for solving NP-complete problems.

- $L_3$ contains the DLV programs from $\langle\{\ \mathtt{v_h}\ \}, \{\mathrm{not}, w\}\rangle$ minus $L_1 \cup L_2$. The complexity of this fragment is $\Delta_2^P$. Here, also the WCH module is employed, which iteratively invokes the MG. Again, only the linear-time part of the MC is invoked.

- $L_4$ contains the programs from the subclass corresponding to $\langle\{\ \mathtt{v}\ \}, \{\mathrm{not}\}\rangle$ minus $L_1 \cup L_2 \cup L_3$. The complexity of this fragment is $\Sigma_2^P$, and the programs are evaluated by the MG module (besides the IG) with calls to the *full* MC module. Note that a flat backtracking algorithm is not sufficient to evaluate $\Sigma_2^P$-complete problems, and such a nested evaluation scheme, with calls to MC, is needed.

- Finally, $L_5$ contains all other programs, i.e., those in the full class (corresponding to $\langle\{\ \mathtt{v}\ \}, \{\mathrm{not}, w\}\rangle$) which are not contained in $L_1 \cup L_2 \cup L_3 \cup L_4$, where we have the full language complexity of $\Delta_3^P$. The evaluation proceeds as for $L_4$, but also the WCH module comes into play for handling the weak constraints.

The three DLV modules, MG, MC, and WCH, thus deal with the three sources of complexity denoted by $(s_1)$, $(s_2)$, and $(s_3)$ in Section 2.4; each of them is fully activated *only if* the respective source of complexity is present in the program at hand.

## 4.3 General Evaluation Strategy

We present next the evaluation flow of the DLV computation in some more detail. It is worth noting that we describe the computational engine of the DLV system [Faber *et al.*, 1999; 2001], but also other systems (like Smodels [Niemelä and Simons, 1996; Simons, 2000], for instance) employ very similar techniques.

Upon startup, the DLV Core 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 is done efficiently (requiring only linear memory and time). The input is usually read from text files, but DLV also provides a bridge to relational databases through an ODBC interface, which allows for retrieving facts stored in relational tables.

Using differential and other advanced database techniques (see [Faber *et al.*, 1999; Leone *et al.*, 2001]) and suitable data structures, the *Intelligent Grounding* (IG) module then efficiently generates a ground instantiation $Ground(\mathcal{P})$ of the input that has the same answer sets as the full program instantiation, but is much smaller in general. For example, in case of a stratified program, the IG module already computes the single answer set, and does not produce any instantiation.

The heart of the computation is then performed by the Model Generator and the Model Checker. Roughly, the former produces some "candidate" answer sets, the stability of which is subsequently verified by the latter. In presence of weak constraints, further processing is needed, which is performed under the control of the WCH module. Since the handling of weak constraints is somehow orthogonal to the rest of the computation, we first focus on the evaluation of standard disjunctive logic programs, describing the processing of weak constraints later on.

**Function** ModelGenerator(var I: Interpretation): Boolean;
**var** inconsistency: Boolean;
**begin**
    DetCons(I,inconsistency);
    **if** inconsistency **then return false**;
    **if** "no atom is undefined in I" **then return** IsStableModel(I);
    Select an undefined ground atom $A$ according to a heuristic;
    **if** ModelGenerator($I \cup \{A\}$) **then return true**;
    **else return** ModelGenerator($I \cup \{\text{not } A\}$);
**end**;

Figure 4.2: Computation of Answer Sets

The hard part of the computation, on the ground program, is performed by the *Model Generator*, which is sketched in Figure 4.2. For brevity, here $\mathcal{P}$ refers to the (simplified) ground program.

Roughly, as already said, the Model Generator produces some "candidate" answer sets. Each candidate $I$ is then verified by the function *IsStableModel(I)*,

which checks whether $I$ is a minimal model of the program $\mathcal{P}^I$ obtained by applying the GL-transformation w.r.t. $I$. More details about the model checking will follow below.

Initially, the *ModelGenerator* function is invoked with $I$ set to the empty interpretation (all atoms are undefined at this stage). If the program $\mathcal{P}$ has an answer set, then the function returns true and sets $I$ to the computed answer set; otherwise it returns false. The Model Generator is similar to the Davis-Putnam procedure in SAT solvers. It first calls a function DetCons, which extends $I$ with those literals that can be deterministically inferred. This is similar to unit propagation as employed by SAT solvers, but exploits the peculiarities of DLP for making further inferences (e.g., it uses the knowledge that every stable model is a minimal model).

If DetCons does not detect any inconsistency, an atom $A$ is selected according to a heuristic criterion and ModelGenerator is recursively called on both $I \cup \{A\}$ and $I \cup \{\text{not } A\}$. The atom $A$ corresponds to a branching variable of a SAT solver; it is said to be *possibly-true* [Leone *et al.*, 1997]).

The computation proceeds by alternately selecting a possibly-true and recalling ModelGenerator, until either a total model of $Ground(\mathcal{P})$ is reached or two contradictory literals are derived. If a model is found (there are no more undefined literals), the Model Checker is called; otherwise, the function returns *false* and backtracking is performed.

The *Model Checker* (MC) verifies whether the model $M$ at hand is an answer set for the input program $\mathcal{P}$. In particular, the MC disregards weak constraints, and verifies whether $M$ is an answer set for $Rules(\mathcal{P})$; the optimality of the models w.r.t. the violation of weak constraints is handled by the WCH module. The task performed by MC is very hard in general, because checking the stability of a model is well-known to be co-NP-complete (cf. [Eiter *et al.*, 1997c]). However, for some relevant and frequently used classes of programs answer-set checking can be efficiently performed (see Table 2.3 in Section 2.4).

The MC implements novel techniques for answer-set checking [Koch and Leone, 1999], which extend and complement previous results [Ben-Eliyahu and Dechter, 1994; Ben-Eliyahu and Palopoli, 1994; Leone *et al.*, 1997]. The MC fully complies with the complexity bounds specified in Chapter 2. Indeed, (a) it terminates in polynomial time on every program where answer-set checking is tractable according to Table 2.3 (including, e.g., HCF programs); 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

subprograms that are not HCF. Note that it may well happen that only a very small part of the program is not HCF [Koch and Leone, 1999].

Finally, once an answer set has been found, the control is returned to the front-end in use, which performs post-processing and possibly invokes the MG to look for further models.

In presence of weak constraints, after the instantiation of the program, the computation is governed by the WCH and consists of two phases: (i) the first phase determines the cost of an optimal answer set[2], together with one "witnessing" optimal answer set and, (ii) the second phase computes all answer sets having that optimal cost. It is worthwhile noting that both the IG and the MG also have built-in support for weak constraints, which is activated (and therefore incurs higher computational cost) only if weak constraints are present in the input. The MC, instead, does not need to provide any support for weak constraints, since these do not affect answer-set checking at all.

Having a look at the process of model generation, it is clear that both the choice of "good" possibly-trues at each step (i.e., a sequence of possibly-trues that quickly leads to an answer set) as well as the implementation of DetCons is very important. In particular, DetCons is crucial in two ways: it has to perform its task as quickly as possible, while pruning the search space as much as possible.

[2]By *cost* of an answer set we mean the sum of the weights of the weak constraints violated by the answer set, weighted according to their priority level – see Section 1.2.

# Chapter 5

# Pruning the Search Space

In this chapter we present two pruning operators for DLP (*Fitting*'s ($\Phi_{\mathcal{P}}$) operator, and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator), analyzing several interesting properties on some syntactically restricted classes of DLP programs.

## 5.1 Pruning Operators

In this section we review two operators that are useful to implement DetCons (see the previous chapter). As already mentioned, DetCons has to expand a given interpretation as much as possible to reduce the search space, while ensuring that such an expansion never causes any answer set[1] to be missed. In other words, if an interpretation $I$ is contained in an answer set $M$, that answer set will also contain the expansion of $I$ computed by DetCons. We can state this "safety" property formally.

**Definition 5.1** A generic operator $\Gamma_{\mathcal{P}}$ is said to be "safe" if, for each interpretation $I$, and for each answer set M of a given program $\mathcal{P}$, we have $I \subseteq M$ iff $\Gamma_{\mathcal{P}}(I) \subseteq M$.

The two operators in question are the *Fitting*'s ($\Phi_{\mathcal{P}}$) operator and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator. Both have the property described above, and extend the two corresponding operators defined for disjunction-free programs [Fitting, 1985; Van Gelder *et al.*, 1991] to the class of disjunctive logic programs. Both $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ consist of two parts: The part drawing positive inferences (which is an

[1] It is worth remembering that under the answer set semantics *Stable Models* and *Answer Sets* are synonims, and we can refer to a *closed interpretation* as to a *model*. The reader can find full background in Chapter 1.

extension of the immediate consequence operator $T_{\mathcal{P}}$, defined for three-valued interpretations of normal logic programs [Van Gelder *et al.*, 1991], to disjunctive programs) is the same for both operators; they only differ in the way they perform negative inferences.

**Definition 5.2** Let $\mathcal{P}$ be a program, and $I$ be an interpretation.
$\mathcal{T}_{\mathcal{P}}(I) = \{a \mid \exists r \in \mathcal{P}\ s.t.\ a \in H(r) :\ H(r) - \{a\} \subseteq \text{not}.I \wedge B(r) \subseteq I\}$ .

Thus, $\mathcal{T}_{\mathcal{P}}(I)$ derives an atom $a$ from a rule $r$, if the body of $r$ is true w.r.t. $I$ and, apart from $a$, all other atoms in the head of $r$ are false w.r.t. $I$. Note that, in order to be *closed* (i.e., a model), any interpretation extending $I$ must necessarily contain $a$, otherwise rule $r$ is violated.

**Example 5.3** Consider the following program $\mathcal{P}_1$:

$$\{a \vee b.\ ;\ c \text{:-}\ \text{not}\ a.\ ;\ d \text{:-}\ e.\ ;\ e \text{:-}\ d.\ ;\ k \text{:-}\ \text{not}\ e.\}$$

Suppose $I$ = $\{\text{not}\ a\}$: Then $b$ is derived via the first rule, and $c$ via the second (whose body is contained in $I$), so $\mathcal{T}_{\mathcal{P}}(I)$ = $\{b, c\}$.

Intuitively, given an interpretation $I$, $\mathcal{T}_{\mathcal{P}}$ derives a set of atoms that are strictly needed to extend $I$ to a model. Note that $\mathcal{T}_{\mathcal{P}}$ is deterministic, that is, its result is a single set of literals.

### 5.1.1 *Fitting*'s ($\Phi_{\mathcal{P}}$) Operator

We extend *Fitting*'s operator, which was originally defined in [Fitting, 1985], to the disjunctive case. The way this operator makes negative inferences is described and then combined with the $\mathcal{T}_{\mathcal{P}}$ operator.

**Definition 5.4** Let $\mathcal{P}$ be a program, and $I$ an interpretation.

$$\gamma_{\mathcal{P}}(I) \ = \ \{a \in B_{\mathcal{P}} \mid \ \forall r \in ground(\mathcal{P})\ s.t.\ a \in H(r) :\ H(r) - \{a\}\ is\ true\ w.r.t.\ I, or\ B(r)\ is\ false\ w.r.t.\ I\}.$$

Thus, $\gamma_{\mathcal{P}}(I)$ derives an atom $a$, if each rule with $a$ in the head already has a false body or a true head (the head being true by an atom different from $a$). Note that all such a rules with $a$ in the head are satisfied in $I$, and they remain satisfied in all extensions of $I$ even if $a$ is set to false.

**Example 5.5** Consider the program $\mathcal{P}_1$ of Example 5.3, and the interpretation $I$ = {a}. Here $\gamma_{\mathcal{P}}(I) = \{b, c\}$.

Intuitively, given an interpretation $I$, $\gamma_{\mathcal{P}}$ computes those atoms that will not appear in any minimal model extending $I$ since there is no rule left that could be used to derive them (and "support" its introduction in the model).

We can now define a single step of Fitting's operator $\Phi_{\mathcal{P}}$ and its least fixpoint as follows:

**Definition 5.6** Let $\mathcal{P}$ be a program, and $I$ an interpretation.

$$\Phi_{\mathcal{P}}(I) = \mathcal{T}_{\mathcal{P}}(I) \cup \text{not}.\gamma_{\mathcal{P}}(I).$$

Starting from $I$ we define the following sequence $F_k$:

$$F_0 = I$$
$$F_k = F_{k-1} \cup \Phi_{\mathcal{P}}(F_{k-1}), \quad k > 0.$$

We now have a growing sequence whose *n-th* term is the *n-fold* application of $\Phi_{\mathcal{P}}$ to $I$, and define the least fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ of $\Phi_{\mathcal{P}}$ containing $I$, as the limit to which $\{F_n\}_{n \in \mathcal{N}}$ converges.

**Example 5.7** Consider the program $\mathcal{P}_1$ of Example 5.3, and the interpretation $I$ = {a}. It is easy to see that $\Phi_{\mathcal{P}}(I) = \emptyset \cup \text{not}.\{b, c\} = \{\text{not } b, \text{not } c\}$.

We thus obtain:

$$F_0 = I = \{a\}.$$
$$F_1 = F_0 \cup \{\text{not } b, \text{not } c\} = \{a, \text{not } b, \text{not } c\}.$$
$$F_2 = F_1 \cup \emptyset = \{a, \text{not } b, \text{not } c\} = F_1 = \Phi^{\infty}_{\mathcal{P}}(I)$$

Next we assert the already discussed "safety" property for $\Phi_{\mathcal{P}}$.

**Theorem 5.8** For every answer set $M$ of a given program $\mathcal{P}$, *if* an interpretation $I$ is contained in $M$, *then* $\Phi^{\infty}_{\mathcal{P}}(I) \subseteq M$.

**Proof.** It is enough to show that $\Phi_{\mathcal{P}}(I) \subseteq M$ for any $I \subseteq M$; indeed, if this is the case, we can take $I \cup \Phi_{\mathcal{P}}(I)$ as another interpretation contained in $M$, and then iteratively go on until the fixpoint is reached. We first show that the positive part of $\Phi_{\mathcal{P}}(I)$ is contained in $M$; then, we show that also its negative part is contained in $M$.

By definition, the positive part of $\Phi_{\mathcal{P}}(I)$ is $\mathcal{T}_{\mathcal{P}}(I)$. Let's take a *positive* atom $a \in \Phi_{\mathcal{P}}(I)$. Thus, there must exist a rule $r \in \mathcal{P}$ such that $a \in H(r)$ with $H(r) - \{a\} \subseteq \mathrm{not}.I$ and $B(r) \subseteq I$, i.e., the rest of the head of $r$ is false while its body is true w.r.t. $I$. But since $I \subseteq M$, this means that the body of $r$ is true w.r.t. $M$; since $M$ is an answer set, we must have $a \in M$, otherwise $r$ would be violated w.r.t. $M$, and $M$ would not be an answer set.

The negative part of $\Phi_{\mathcal{P}}(I)$, again by definition, is $\gamma_{\mathcal{P}}(I)$. Let $b$ be an atom in $\gamma_{\mathcal{P}}(I)$. Suppose, by contradiction, that $\mathrm{not}\ b \notin M$; then $b \in M$, as $M$ is a total interpretation. Since $b \in \gamma_{\mathcal{P}}(I)$, for each rule having $b$ in the head, we have that either $H(r) - \{b\}$ is true w.r.t. $I$, or $B(r)$ is false w.r.t. $I$; since $I \subseteq M$, this means that all such rules are already satisfied (either by a true head or a false body) also w.r.t. $M$, and cannot give "support" to $b$. Consequently, $M$ contains atom $b$ which is not supported, contradicting the well-known "supportedness" property of answer sets (see, e.g., [Gelfond and Leone, 2002]). □

Importantly, we have the following.

**Proposition 5.9** Given a propositional program $\mathcal{P}$ and an interpretation $I$ for it, $\Phi^{\infty}_{\mathcal{P}}(I)$ is *linear-time computable*.

**Proof.** $\Phi^{\infty}_{\mathcal{P}}(I)$ is well-known to be linear-time computable for a non-disjunctive $\mathcal{P}$. This result was stated in [Berman *et al.*, 1995], where it is attributed to "folklore".

It is easy to see that this result carries over to the disjunctive case by using a suitable data structure for identifying atoms, which are derived by $\gamma_{\mathcal{P}}(I)$, in constant time. One way of achieving this is to keep a counter of potentially supporting rules for each atom $b$ (i.e., rules having $b$ in the head such that the body is not false nor the head is made true by an atom different from $b$) – whenever such a counter becomes zero, atom $b$ is derived false by $\gamma_{\mathcal{P}}(I)$. □

Thus, the $\Phi_{\mathcal{P}}$ operator seems to be a good choice as a pruning operator: it is "*safe*" (Theorem 5.8), has the capability to perform negative inferences (Definition 5.6), and its fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ is efficiently computable (Proposition 5.9).

Unfortunately, $\Phi_{\mathcal{P}}$ fails to derive all possible negative consequences. For instance, in Example 5.7 it fails to derive $d$ and $e$ as false w.r.t. $I$ while the only rules having these atoms in the head will never have a true body. The *Well-founded* operator presented in the following section is "stronger" in this respect.

### 5.1.2 *Well-founded* ($\mathcal{W}_\mathcal{P}$) Operator

The $\mathcal{W}_\mathcal{P}$ operator defined in [Leone *et al.*, 1997] extends the operator defined in [Van Gelder *et al.*, 1991] (whose least fixpoint is the Well-founded model) to the disjunctive case. It is defined by an extension of the notion of *unfounded sets* to disjunctive logic programs.

**Definition 5.10** Let $I$ be an interpretation for a program $\mathcal{P}$. A set $X \subseteq B_\mathcal{P}$ of ground atoms is an *unfounded set* for $\mathcal{P}$ w.r.t. $I$ if, for each $a \in X$ and for each rule $r \in ground(\mathcal{P})$ such that $a \in H(r)$, at least one of the following conditions holds:

1. $B(r) \cap \text{not}.I \neq \emptyset$, that is, the body of $r$ is false w.r.t. $I$.
2. $B^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to $X$.
3. $(H(r) - X) \cap I \neq \emptyset$, that is, an atom in the head of $r$, distinct from $a$ and other elements in $X$, is true w.r.t. $I$.

**Example 5.11** Considering the program $\mathcal{P}_1$ of Example 5.3 and the interpretation $I$ = {a}, we get $GUS_\mathcal{P}(I)$ = $\{b, c, d, e\}$. $b$ is added because of the third condition, and $c$ because of the first in Definition 5.10. Then $d$ and $e$ appear in the head of only a single rule each and for both the second condition of Definition 5.10 holds. We obtain $\mathcal{W}_\mathcal{P}(I) = \{\text{not } b, \text{not } c, \text{not } d, \text{not } e\}$ and $W_0 = \{a\}, W_1 = \{a, \text{not } b, \text{not } c, \text{not } d, \text{not } e\}, W_2 = \{a, \text{not } b, \text{not } c, \text{not } d, \text{not } e, k\}, W_3 = W_2 = \mathcal{W}_\mathcal{P}^\infty(I)$.

While for non-disjunctive programs the union of unfounded sets is again an unfounded set for all interpretations, this does not hold, in general, for disjunctive programs.

**Example 5.12** Given $\mathcal{P} = \{a \vee b\}$ and $I = \{a, b\}$, both $\{a\}$ and $\{b\}$ are unfounded sets w.r.t. $I$; but their union $\{a, b\}$ is not.

We thus denote by $\mathbf{I}_\mathcal{P}$ the set of all interpretations of $\mathcal{P}$ for which the union of all unfounded sets for $\mathcal{P}$ w.r.t. $I$ is an unfounded set for $\mathcal{P}$ w.r.t. $I$ as well. In analogy with traditional logic programming, given $I \in \mathbf{I}_\mathcal{P}$, we call the union of all unfounded sets for $\mathcal{P}$ w.r.t. $I$ the *greatest unfounded set* of $\mathcal{P}$ w.r.t. $I$, and denote it by $GUS_\mathcal{P}(I)$.

Because the existence of the *greatest unfounded set* is not guaranteed in general, the question of how to decide whether an interpretation $I$ is in $\mathbf{I}_\mathcal{P}$ naturally comes up (also from the viewpoint of complexity, which we will deal with later on). There is a class of interpretations, called *unfounded-free interpretations*, which always have the greatest unfounded set.

**Definition 5.13** Let $I$ be an interpretation for a program $\mathcal{P}$. $I$ is *unfounded-free* if $I \cap X = \emptyset$ for each unfounded set $X$ for $\mathcal{P}$ w.r.t. $I$.

Unfounded-free interpretations have a nice semantic property, that, as we will see in the next sections, has also a practical impact on the computation.

**Proposition 5.14** *[Leone* et al.*, 1997]* Let $I$ be an unfounded-free interpretation for a program $\mathcal{P}$. Then

$\mathcal{P}$ has the greatest unfounded set $\mathcal{GUS}_\mathcal{P}(I)$ (i.e., $I \in \mathbf{I}_\mathcal{P}$).

Thus, an unfounded-free interpretation always admits the greatest unfounded set, and this set is efficiently computable. We can now introduce the *Well-founded* operator.

**Definition 5.15** Let $\mathcal{P}$ be a program, and $I \in \mathbf{I}_\mathcal{P}$ be an *unfounded-free* interpretation. We define the $\mathcal{W}_\mathcal{P}$ operator as follows:

$$\mathcal{W}_\mathcal{P}(I) \;=\; \mathcal{T}_\mathcal{P}(I) \cup \mathrm{not}.\mathcal{GUS}_\mathcal{P}(I).$$

This definition extends the $\mathcal{W}_\mathcal{P}$ operator defined in [Van Gelder *et al.*, 1991] (whose least fixpoint is the Well-founded model) to the disjunctive case. Note that, since $\mathcal{W}_\mathcal{P}$ is defined on the domain $\mathbf{I}_\mathcal{P}$, each fixpoint of $\mathcal{W}_\mathcal{P}$ by definition admits the greatest unfounded set (since each fixpoint of $\mathcal{W}_\mathcal{P}$ must belong to the domain $\mathbf{I}_\mathcal{P}$ of $\mathcal{W}_\mathcal{P}$).

**Example 5.16** Consider the program $\mathcal{P}_1$ of Example 5.3, and the interpretation $I$ = {a}. Then, $\mathcal{W}_\mathcal{P}(I) = \emptyset \cup \mathrm{not}.\{b, c, d, e\} = \{\mathrm{not}\ b, \mathrm{not}\ c, \mathrm{not}\ d, \mathrm{not}\ e\}$.

Observe that, as Example 5.16, clearly shows, thanks to $\mathcal{GUS}_\mathcal{P}$, $\mathcal{W}_\mathcal{P}$ derives more negative information than $\Phi_\mathcal{P}$, and therefore ensures a better pruning of the search space.

Let us now define the least fixpoint $\mathcal{W}_\mathcal{P}^\infty$ of the $\mathcal{W}_\mathcal{P}$ operator.

**Definition 5.17** Given a program $\mathcal{P}$ and an interpretation $I$, we define the following sequence $W_k$:

$$W_0 = I$$
$$W_k = W_{k-1} \cup \mathcal{W}_\mathcal{P}(W_{k-1}), \quad k > 0$$

We have a growing sequence whose *n-th* term is the *n-fold* application of $\mathcal{W}_\mathcal{P}$ to $I$, and define the least fixpoint $\mathcal{W}_\mathcal{P}^\infty(I)$ of $\mathcal{W}_\mathcal{P}$ containing $I$, as the limit to which $\{W_n\}_{n\in\mathcal{N}}$ converges.

**Example 5.18** Again considering the program $\mathcal{P}_1$ from Example 5.3 and starting from the interpretation $I$ = {a}, we have:

$$W_0 = I = \{a\}.$$
$$W_1 = W_0 \cup \{\text{not } b, \text{not } c, \text{not } d, \text{not } e\} = \{a, \text{not } b, \text{not } c, \text{not } d, \text{not } e\}.$$
$$W_2 = W_1 \cup \{k\} = \{a, \text{not } b, \text{not } c, \text{not } d, \text{not } e, k\}.$$
$$W_3 = W_2 \cup \emptyset = W_2 = \mathcal{W}_\mathcal{P}^\infty(I).$$

Next we state that the $\mathcal{W}_\mathcal{P}$ operator has the "*safety*" property previously discussed.

**Proposition 5.19** [Leone *et al.*, 1997] Let $I$ be an interpretation for a program $\mathcal{P}$, and let $M$ be an answer set for $\mathcal{P}$. If $I \subseteq M$, then

(a) $I$ belongs to the domain $\mathbf{I}_\mathcal{P}$ of $\mathcal{W}_\mathcal{P}$, and

(b) $\mathcal{W}_\mathcal{P}(I) \subseteq M$.

The $\mathcal{W}_\mathcal{P}$ operator appears to be a good pruning operator: it is "safe", and performs more negative inferences than $\Phi_\mathcal{P}$. A negative point of $\mathcal{W}_\mathcal{P}$ is that it is applicable only on unfounded-free interpretations. According to the following proposition, we cannot efficiently test whether $I$ is indeed unfounded-free (unless P = NP).

**Proposition 5.20** [Leone *et al.*, 1997] Let $\mathcal{P}$ be a propositional program and $I$ be an interpretation for $\mathcal{P}$. Deciding whether $I$ is unfounded-free is co-NP-complete.

## 5.2 Pruning Operators on Syntactically Restricted Classes of Programs

In this section, we explore several interesting properties of the pruning operators on some syntactically restricted classes of programs. To this end, we introduce *dependency graphs* which represent the dependencies of head predicates on the positive body predicates of rules.

**Definition 5.21** With every program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (\mathcal{N}, E)$, called the *dependency graph* of $\mathcal{P}$, where $(i)$ each predicate of $\mathcal{P}$ is a node in $\mathcal{N}$, and $(ii)$ there is an arc in $E$ directed from node $a$ to node $b$ if there is a rule $r$ in $\mathcal{P}$ such that two predicates $a$ and $b$ appear in $B^+(r)$ and $H(r)$, respectively.

The dependency graph allows us to single out the recursive parts of the program, and split the program into subprograms having different properties.

**Definition 5.22** A *component* $C$ of $DG_{\mathcal{P}}$ is a maximal strongly connected subset of nodes of $DG_{\mathcal{P}}$. The *subprogram* of $C$ is the set of rules in $\mathcal{P}$ having a head predicate in $C$, denoted by $\mathcal{P}_C$. The set of all components of $DG_{\mathcal{P}}$ is denoted by $Comp(\mathcal{P})$.

Since there is a one-to-one correspondence between nodes in $DG_{\mathcal{P}}$ and predicates in $\mathcal{P}$, we will often refer to *components of* $\mathcal{P}$ (meaning components of $DG_{\mathcal{P}}$), and identify $Comp(\mathcal{P})$ as the *components of* $\mathcal{P}$.

It is worthwhile noting that the same disjunctive rule may occur in the subprograms of two different components. For instance, the program consisting of the single rule $a \vee b.$, has two components: $C_1 = \{a\}$ and $C_2 = \{b\}$. Rule $a \vee b.$ belongs to both subprograms $\mathcal{P}_{C_1}$ and $\mathcal{P}_{C_2}$.

We are now in the position to define the concepts of (a)cyclicity and head-cycle freeness, which play a very important role in our computational strategy.

**Definition 5.23** Given a program $\mathcal{P}$ and its dependency graph $DG_{\mathcal{P}}$, we say that:

- a component $C$ is *cyclic* if the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains at least one recursive rule (i.e., a rule $r$ such that a head predicate and a positive body predicate of $r$ are in $C$); $C$ is *acyclic* if it is not cyclic.

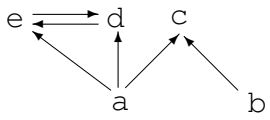


Figure 5.1: *The dependency graph $DG_{\mathcal{P}_1}$ of program $\mathcal{P}_1$*

- a component $C$ is *head-cycle-free* ($HCF$) iff the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains no rule $r$ such that two predicates occurring in the head of $r$ belong to $C$.

$DG_{\mathcal{P}}$ and $\mathcal{P}$ are *cyclic* if there is *at least one cyclic* component, otherwise they are acyclic. They are $HCF$ if *all* components are $HCF$.

Observe that acyclicity obviously implies head-cycle freeness; while an HCF component might be cyclic (e.g., if it is not disjunctive) or acyclic.

**Example 5.24** Consider the following program $\mathcal{P}_1$:

$$\{\ a \vee b.\ ;\ \ c \text{ :- } a.\ \ ;\ \ c \text{ :- } b.\ \ ;\ \ d \vee e \text{ :- } a.\ \ ;\ \ d \text{ :- } e.\ ;\ \ e \text{ :- } d, \text{not } b.\ \}$$

The dependency graph $DG_{\mathcal{P}_1}$ of $\mathcal{P}_1$ is depicted in Figure 5.1. There are four components: $\{a\}$, $\{b\}$, $\{c\}$, $\{d, e\}$. All of them are acyclic except for the last which is also the only non-$HCF$ component, as the head of $d \vee e$ :- $a.$ contains two predicates belonging to the same cycle. The whole graph, and thus the program, is cyclic but not $HCF$.

We next present a well-known theorem in DLP community about a class of programs for which the operators $\Phi^{\infty}_{\mathcal{P}}(I)$ and $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ are equivalent. The importance of this theorem will become clear in the next section.

**Theorem 5.25** Given a program $\mathcal{P}$ and an unfounded-free interpretation $I$, if $\mathcal{P}$ is *acyclic* then $\mathcal{W}_{\mathcal{P}}(I) = \Phi_{\mathcal{P}}(I)$.

**Proof.** Let $I$ be an unfounded-free interpretation of an acyclic program $\mathcal{P}$. The positive parts of $\mathcal{W}_{\mathcal{P}}(I)$ and $\Phi_{\mathcal{P}}(I)$ coincide by definition, since both of them are obtained by $\mathcal{T}_{\mathcal{P}}(I)$ (see Definition 5.6 and Definition 5.15).

We have to show that also $\mathit{GUS}_{\mathcal{P}}(I) = \gamma_{\mathcal{P}}(I)$ holds for acyclic programs. Let $A = \gamma_{\mathcal{P}}(I)$ and $B = \mathit{GUS}_{\mathcal{P}}(I)$. For each atom $a \in A$, the set $\{a\}$ is an unfounded

set for $\mathcal{P}$ w.r.t. $I$, as, by Definition of $\gamma_\mathcal{P}$, all ground rules with $a$ in the head satisfy Condition 1 or Condition 3 of Definition 5.10. Therefore, for each $a \in A$, we have that $\{a\}$ is contained in the Greatest Unfounded Set $B$, that is, $A \subseteq B$. On the other hand, we know that also $B$ is an unfounded set for $\mathcal{P}$ w.r.t. $I$. Since $\mathcal{P}$ is acyclic, we obtain that Condition 2 is superfluous for the unfoundedness of $B$, the ground rules having an element from $B$ in the head satisfy either Condition 1 or Condition 3 of Definition 5.10. Consequently, all elements from $B$ belong to $\gamma_\mathcal{P}(I)$, that is, $B \subseteq A$.

Hence, we have that $A = B$, that is, $\mathit{GUS}_\mathcal{P}(I) = \gamma_\mathcal{P}(I)$ holds. □

Thus, on the class of acyclic programs, one can conveniently use Fitting's pruning operator, which is efficiently computable and equivalent to $\mathcal{W}_\mathcal{P}$ on these programs. The Well-founded operator, however, has a stronger inference power than Fitting's in the general case (see, e.g., Example 5.16, and the subsequent observation). To be able to exploit the $\mathcal{W}_\mathcal{P}$ operator in practice, we have to: (1) be able to efficiently detect whether it is applicable or not (i.e., if the interpretation at hand is unfounded-free or not – in general a co-NP-complete task, cf. Proposition 5.20), and (2) provide a concrete method for computing $\mathit{GUS}_\mathcal{P}(I)$ efficiently. The $\mathcal{R}_{\mathcal{P},I}$ operator, defined next, will serve this purpose.

**Definition 5.26** Let $\mathcal{P}$ be a program and $I$ an interpretation. Then we define an operator $\mathcal{R}_{\mathcal{P},I}$ as follows:

$$\begin{array}{rccl} \mathcal{R}_{\mathcal{P},I}: & 2^{B_\mathcal{P}} & \rightarrow & 2^{B_\mathcal{P}} \\ & X & \mapsto & \{a \in X \mid \forall r \in \mathit{ground}(\mathcal{P})\ \mathit{with}\ a \in H(r), \\ & & & \quad B(r) \cap (\mathrm{not}.I \cup X) \neq \emptyset \text{ or} \\ & & & \quad (H(r) - \{a\}) \cap I \neq \emptyset\} \end{array}$$

Given a set $X \subseteq B_\mathcal{P}$, the sequence $R_0 = X$, $R_n = \mathcal{R}_{\mathcal{P},I}(R_{n-1})$ decreases monotonically and converges finitely to a limit that we denote by $\mathcal{R}^\omega_{\mathcal{P},I}(X)$.

We next prove a lemma, which was not known so far, and is fundamental for the concrete exploitation of the $\mathcal{W}_\mathcal{P}$ operator in DLP systems.

**Lemma 5.27** Let $\mathcal{P}$ be a HCF program and $I$ an interpretation. $\mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$ is equal to the union of all unfounded sets w.r.t. $\mathcal{P}$ and $I$.

**Proof.** Consider an arbitrary unfounded set $X$ w.r.t. $\mathcal{P}$ and $I$. It is easy to see that $X \subseteq \mathcal{R}_{\mathcal{P},I}(X)$ and also $X \subseteq \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$, as also for any $Y \supseteq X$ we have $Y \subseteq \mathcal{R}_{\mathcal{P},I}(Y)$, so $X \subseteq \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$.

On the other hand, we can show that any $a \in \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$ is contained in some unfounded set. Observe that for each rule $r \in \mathcal{P}$ such that $a \in H(r)$, $B(r) \cap (\neg.I \cup \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})) \neq \emptyset$ or $(H(r) \setminus \{a\}) \cap I \neq \emptyset$ holds by definition of $\mathcal{R}_{\mathcal{P},I}$. So in particular $B(r) \cap \neg.I \neq \emptyset$ or $B(r) \cap \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}}) \neq \emptyset$ hold. The only reason why $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$ itself might not be an unfounded set is if $B(r) \cap (\neg.I \cup \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})) = \emptyset$ and $(H(r) \setminus \{a\}) \cap I \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$ holds for some rule, because then $(H(r) \setminus \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})) \cap I = \emptyset$ and consequently none of the three conditions of Definition 5.10 hold for $r$.

If this is the case, we can construct an unfounded set, starting from $X \setminus \{b \mid b \in H(r) \setminus \{a\}, a \in H(r)\}$. This however, may invalidate condition 2 of Definition 5.10 for some rule $r_1$ with $c \in H(r_1)$ and $c \in X_1$. Note that then $c \neq a$, as $\mathcal{P}$ is HCF. Eliminating all such $c$s where also conditions 1 and 3 do not hold, may again entail that condition 2 of Definition 5.10 becomes invalidated. However, this process can be iterated. In this process, $a$ is never eliminated (as the program is HCF, so in the worst case we arrive at $\{a\}$).

More formally, we create a sequence as follows: Start at

$$\begin{aligned} X_0 &= X \setminus Y_0 \text{where} \\ Y_0 &= \{b \mid b \in H(r) \setminus \{a\}, a \in H(r)\} \end{aligned}$$

Subsequently, for $i > 0$:

$$\begin{aligned} X_i &= X_{i-1} \setminus Y_i \text{where} \\ Y_i &= \{c \mid c \in H(r) \cap X_{i-1}, B^+(r) \cap X \subseteq Y_{i-1}, B(r) \cap \text{not}.I = \emptyset, \\ &\quad (H(r) \setminus X_{i-1}) \cap I = \emptyset\} \end{aligned}$$

Obviously this sequence converges, and the set which is the limit, is an unfounded set w.r.t. $I$ and $\mathcal{P}$. □

The properties shown in the following theorem guarantee that the $\mathcal{W}_{\mathcal{P}}$ operator can be efficiently used on head cycle free programs.

**Theorem 5.28** Let $\mathcal{P}$ be a $HCF$ ground program and $I$ be an interpretation, then

1. detecting whether $I$ is unfounded free is feasible in *linear time*,
2. if $I$ is unfounded-free, $GUS_{\mathcal{P}}(I)$ can be computed in *linear time*,
3. $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ (if it is defined) is computable in *quadratic time*,

all in the size of $\mathcal{P}$.

**Proof.** From Lemma 5.27 it follows that $I$ is unfounded-free iff $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}}) \cap I = \emptyset$. We refer to Section 6.1.2, in which a linear time implementation of $\mathcal{R}^{\omega}_{\mathcal{P},I}(X)$ is described, so item 1 follows.

If $I$ is unfounded-free, then $\mathit{GUS}_{\mathcal{P}}(I)$ exists (cf. Proposition 5.14), and in fact from Lemma 5.27 we get that $\mathit{GUS}_{\mathcal{P}}(I) = \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$, obtaining item 2.

Item 3 follows from item 2; a linear number of iterations is sufficient, each of which consumes at most linear time, so in total at most quadratic time is spent. □

These theorems suggest us a direction to follow in order to improve the capability of pruning the search space in DLP systems.

# Chapter 6

# Optimizing the Pruning

In this chapter, we design our new method for the intelligent combination of the pruning operators, and discuss some key issues for its implementation into the DLV system. We then report the results of our experimentation activity on a number of benchmark problems.

## 6.1 Efficient Combination Of Pruning Operators

We now show how to combine the $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ operators, resulting in an efficient implementation of DetCons.

### 6.1.1 A Pondered Choice

From the previous sections, given a program $\mathcal{P}$ and an interpretation $I$, we know that:

- the computation of $\Phi_{\mathcal{P}}^{\infty}(I)$ is always very efficient (*linear time computable*);
- $\mathcal{W}_{\mathcal{P}}$ is "stronger" than $\Phi_{\mathcal{P}}$ (i.e., $\Phi_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(I)$ for any interpretation $I$);
- the computation of $\mathcal{W}_{\mathcal{P}}$ is intractable in the general case (since deciding whether an interpretation belongs to its domain is co-NP-hard);
- the computation of $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$ is tractable (*quadratic*) when $\mathcal{P}$ belongs to the restricted class of *head-cycle-free* programs;
- $\Phi_{\mathcal{P}}$ is equivalent to $\mathcal{W}_{\mathcal{P}}$ when $\mathcal{P}$ belongs to the restricted class of *acyclic* programs.

Based on these observations, we have designed an approach which exploits the positive aspects of both operators, including the efficiency of *Fitting*'s operator wherever we are sure that it is equivalent to the *Well-founded* operator or that the computation of the latter is intractable. On the other hand, our approach takes advantage of the (potentially) stronger pruning of the *Well-founded* operator where feasible. In particular, we treat each program component differently, and apply to each component the most appropriate pruning operator.

Our implementation of DetCons is sketched in Figure 6.1. Functions *ComputeFittingFixpoint(I, inconsistency)* and *ComputeWellFoundedFixpoint(I, inconsistency)* compute $\Phi_{\mathcal{P}}^{\infty}(I)$ and $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$, respectively. They set the boolean variable *inconsistency* to true if they detect a contradiction (e.g., a branching variable previously assumed true is proven to be false). Depending on the syntactical structure of each component, we choose the more suitable of the two operators (Fitting's and Well-founded). In particular, we apply $\mathcal{W}_{\mathcal{P}}$ on *cyclic and HCF* components, where it is stronger than $\Phi_{\mathcal{P}}$ but efficiently computable. On the other hand, we apply $\Phi_{\mathcal{P}}$ on *acyclic* components, where it is equivalent to $\mathcal{W}_{\mathcal{P}}$ and more efficiently computable, and on *cyclic and not-HCF* components, where $\mathcal{W}_{\mathcal{P}}$ is intractable.

Thus, if the input program is acyclic, we always apply the linear operator $\Phi_{\mathcal{P}}$ without any loss in pruning strength. If the program is cyclic, we limit the application of $\mathcal{W}_{\mathcal{P}}$ to those components (*cyclic and HCF*) where it has potential for pruning the search space and is efficiently computable.

```
Procedure DetCons(var I: Interpretation, var inconsistency: Boolean);
begin
    inconsistency := false;
    for each component C ∈ Comp(P) do
    begin
        switch classOf(C)
            case acyclic: ComputeFittingFixpoint(I, inconsistency);
            case cyclic-notHCF: ComputeFittingFixpoint(I, inconsistency);
            case cyclic-HCF: ComputeWellFoundedFixpoint(I, inconsistency);
        end;
        if inconsistency then break;
    end;
end;
```

Figure 6.1: *The DetCons Procedure*

### 6.1.2 Implementation Issues

We conclude this section with some relevant implementation remarks. For better understanding, DetCons has been presented in a simplified manner; for details we refer to [Faber, 2002].

At the beginning of the DLP computation, we classify the components of the program w.r.t. acyclicity and head-cycle freeness, since the *DetCons* procedure needs this information. This classification is performed efficiently: We first build the dependency graph $DG_{\mathcal{P}}$ of $\mathcal{P}$ (in linear time); then, we compute the strongly connected components of $DG_{\mathcal{P}}$ applying the linear-time Tarjan algorithm [Tarjan, 1972], and we finally scan the components, checking whether they are acyclic or HCF, also in linear time.

To implement the *Well-founded* operator, we have designed an algorithm computing its negative part, i.e., $GUS_{\mathcal{P},C}(I)$ for an interpretation $I$ and a cyclic HCF component $C$ of a program $\mathcal{P}$ ($GUS_{\mathcal{P},C}(I)$ denotes the union of all unfounded sets of $\mathcal{P}$ w.r.t. $I$ which are contained in component $C$). The efficient implementation of the positive part of the Well-founded operator is straightforward and has already been implemented within DetCons, cf. [Faber, 2002]. The sketch of the algorithm is depicted in Figure 6.2 (all implementation details can be found in [Calimeri, 2001]).

Recall Definition 5.10 where three conditions account for cases in which a set of atoms cannot be derived. Conditions (i) and (iii) basically correspond to rule satisfaction (w.r.t. $I$ and $I - X$, respectively), while condition (ii) is used to detect positive cycles without foundation. The basic idea, given a component $C$, is to compute $C - GUS_{\mathcal{P},C}(I)$ by incrementally deriving atoms in $C$ which are "founded", i.e., which do not belong to $GUS_{\mathcal{P},C}(I)$. This means that we want to build a finite sequence $Y_0, \ldots, Y_n$, where $Y_0 = \emptyset$ and $Y_n = C - GUS_{\mathcal{P},C}(I)$. To this end, we look for rules which do not satisfy any of the three conditions of Definition 5.10 (the conditions are checked w.r.t. $X$ set to $Y_i$ and the interpretation $I$). Once one such a rule $r$ is found, we derive that $H(r) \cap C$ (which is a single atom, since $C$ is HCF) is "founded" (more accurately, "not unfounded"), that is, it does not belong to $GUS_{\mathcal{P},C}(I)$ and can thus be added to $Y_{i+1}$. The "foundedness" of an atom may imply the foundedness of further atoms; we proceed until a fixpoint is reached.

There is yet more room for optimization. Observing Definition 5.10, one can see that condition (1) does not involve the unfounded set $X$: it is therefore "static" with respect to the narrowing process, and can be checked once before computing

the $Y_i$s. In addition, for condition (3) a similar, less straightforward, argument holds: if we take for each $r$ the set $\{r \mid (H(r) - C) \cap I = \emptyset\}$, then for each $i$ we have that $(H(r) - (C - Y_i)) \cap I \neq \emptyset$ only if some $a \in I$ and $a \in Y_i$, but then there is some other rule $r_1$ satisfying (2) or (3), otherwise $a \in Y_i \subseteq C - \mathit{GUS}_{\mathcal{P},C}(I)$ would not hold (as the component $C$ is HCF). So for creating the sequence $Y_0, \ldots, Y_n$ it is sufficient to consider only rules for which (1) and (3) (with $X = C$) do not hold (let us call these rules "active"). Thus, for $i \geq 0$, we compute $Y_{i+1} = Y_i \cup \{a \mid a \in H(r), (B^+(r) \cap C) \subseteq Y_i\}$ where $r$ is an "active" rule. We have implemented this computation by a linear-time algorithm using a propagation queue and counters which store $|B^+(r) \cap Y_i|$ for each "active" rule $r$.

At the end of the computation, all atoms in $C - Y_n$ are known to be unfounded, and we set them to false in $I$. This can result in inconsistency if $I \cap (C - Y_n) \neq \emptyset$, i.e., if an unfounded atom was set to true in $I$.

It is worthwhile noting that procedure *computeGUS* can be seen as a (linear time) implementation of the computation of the fixpoint of the $\mathcal{R}$ operator for component $C$ (i.e., $\mathcal{R}^{\omega}_{\mathcal{P}_C,I}(C)$). At each step, instead of explicitly computing $\mathcal{R}_{\mathcal{P}_C,I}(X)$, the procedure computes its complement $C - \mathcal{R}_{\mathcal{P}_C,I}(X)$. The i-th element $Y_i$ of the above sequence corresponds to the element $X_i = C - Y_i$. Thus, in terms of the $\mathcal{R}_{\mathcal{P}_C,I}$ operator, the procedure computes the sequence $X_0 = C$, $X_1 = \mathcal{R}_{\mathcal{P}_C,I}(X_0)$ ($X_1$ is the set of atoms of $C$ which are not in *FoundedAtoms* after the initialization phase of the procedure in Figure 6.2),..., $X_n = \mathit{GUS}_{\mathcal{P},C}(I)$.

We have designed a further optimization to the above algorithm, that we have also incorporated in our actual implementation of DetCons in the DLV system.[1] Frequently, all atoms in $\mathit{GUS}_{\mathcal{P},C}(I)$ happen to be already false w.r.t. $I$, and its computation is completely useless. We would like to identify cases where this condition can be recognized without actually computing $\mathit{GUS}_{\mathcal{P},C}(I)$. To this end, at each step of DetCons (Figure 6.1), we propagate the deterministic consequences over all components by means of *Fitting*'s operator, and subsequently invoke the *Well-founded* operator only on *some selected* cyclic and HCF components instead of all, as described below.

At the very beginning of the computation, the GUS-computation is applied on each cyclic and HCF component. Later, we invoke the GUS-computation only on components where some atom may have become unfounded by the most recent propagation step. In order to do that, we store some further information during the *Fitting* propagation.

[1] A similar technique, for the (smaller) class of disjunction-free programs, is implemented also in Smodels [Simons, 2000].

```
Procedure computeGUS (var C:Component, var I: Interpretation,
                      var inconsistency: Boolean)
var a,b : Atom;
var FoundedAtoms : Interpretation;  % Stores the set of atoms of C which are
                                    % proven to be "founded" (not unfounded).
var GUSqueue : Queue;       % Stores the atoms whose "foundedness" is to be
                            % propagated; controls the fixpoint computation.
var r.counter : Integer; (∀r)  % Stores the number of atoms of C in B+(r) which are
                            % not proved to be founded. If r.counter becomes zero,
                            % then the head of r gets founded.
  inconsistency := false;
  % Initialize the rules counters and the queue.
  For each atom a ∈ C do
      For each rule r such that (r is active and a ∈ H(r)) do
          r.counter := |{b : b ∈ B+(r) ∩ C}|;
          If r.counter = 0 then
              FoundedAtoms.Add(a);
              GUSqueue.Push(a);
          EndIf;
      EndFor;
  EndFor;
  % Fixpoint Computation.
  While not GUSqueue.empty() do
      a := GUSqueue.Pop();
      For each rule r such that (r is active and a ∈ B+(r)) do
          r.counter := r.counter - 1;
          If r.counter = 0 then
              Let b be the atom of C in H(r);
              FoundedAtoms.Add(b);
              GUSqueue.Push(b);
          EndIf;
      EndFor;
  EndWhile;
  % Set to false all atoms of C which are not in FoundedAtoms.
  For each atom a ∈ C do
      If a ∉ FoundedAtoms then
          If a ∈ I then
              inconsistency := true;
              return;
          else
              I := I ∪ {not a};
          EndIf;
      EndIf;
  EndFor;
EndProcedure;
```

Figure 6.2: *The computeGUS procedure*

Basically, an atom $A$ can become unfounded if it has lost a "potential support", as some rule $r$ containing $A$ in the head has become satisfied during the last propagation, i.e., either the body of $r$ has become false or a head atom of $r$, different from $A$, has become true (recall that a (disjunctive) rule can support only one atom in its head). If no atom of a component $C$ has lost any potential support, then we know that $\mathit{GUS}_{\mathcal{P},C}(I)$ is unaltered, and its computation is superfluous. To automatically recognize such superfluous computations, when a rule becomes satisfied, we push the component of each atom that loses a potentially supporting rule into a queue. We eventually launch the GUS-computation only for the components stored in this queue, i.e. only for those cyclic and HCF components in which at least one head atom lost a potentially supporting rule. We thus avoid a lot of useless GUS computations.

It is worthwhile noting that the algorithms for computing both the *Greatest Unfounded Set* (as described above) and *Fitting*'s operator are *linear-time* algorithms; they use propagation queues and suitable counters à la Dowling and Gallier [Dowling and Gallier, 1984; Minoux, 1988].

## 6.2 Comparisons and Benchmarks

In order to evaluate our intuitions, we have implemented two new pruning operators, based on the conclusions drawn in the previous section in the DLP system DLV [Leone *et al.*, 2005] and experimentally compared the new pruning operators against the original pruning operator employed by DLV. Next, we describe the compared methods, the benchmark problems and instances, and then discuss the results of the experiments.

### 6.2.1 Overview of the Compared Methods

To evaluate our proposed operators, we have implemented the following three approaches on top of DLV and compared them by means of various benchmarks:

**Old.** The method originally employed by DLV. It always uses the generalization of Fitting's operator $\Phi_{\mathcal{P}}$ introduced in Section 5.1.1, which is efficiently computable (a fixpoint is reached in linear time), but does not prune the search space as much as $\mathcal{W}_{\mathcal{P}}$.

**ifPoss.** Based on the generalized Well-founded operator $\mathcal{W}_\mathcal{P}$ introduced in Section 5.1.2, and exploiting observations from Section 6.1, this method avoids the use of $\mathcal{W}_\mathcal{P}$ on those components where its computation is very expensive (since deciding its applicability is intractable). It employs $\mathcal{W}_\mathcal{P}$ on all head-cycle free components, while resorting to the generalization of Fitting's operator $\Phi_\mathcal{P}$ on the remaining (i.e., non-HCF) components.

**ifNeed.** This is the method described in Figure 6.1. It fully implements the theoretical results from Section 6.1, using both $\Phi_\mathcal{P}$ and $\mathcal{W}_\mathcal{P}$ where appropriate, and is a refinement of method **ifPoss**. $\mathcal{W}_\mathcal{P}$ is only used for cyclic head-cycle free components, whereas $\Phi_\mathcal{P}$ is applied on all acyclic components.

### 6.2.2 Benchmark Problems

To evaluate the pruning techniques described in the previous sections, we chose three benchmark problems, namely Hamiltonian Path, Blocksworld Planning, and Sokoban.

For the sake of readability, the full encodings used for the benchmarks are reported in Appendix A.1.

**Hamiltonian Path (HAMPATH)** is a classical NP-complete problem (already introduced in Chapter 3.1) from graph theory : Given an undirected graph $G = (V, E)$, where $V$ is the set of vertices of $G$ and $E$ is the set of edges, 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?

This is almost the same problem as described in Chapter 3.2, but the path does not have to be cyclic. The full encoding, almost the same as the one presented in Chapter 3.1, is reported in Appendix A.1.1.

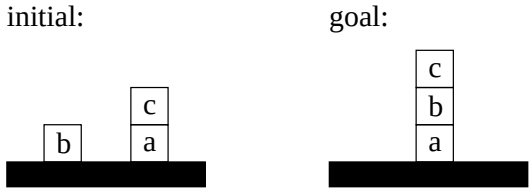


Figure 6.3: A Blocksword Instance

**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. The encoding is reported in Appendix A.1.2.

Figure 6.3 shows a simple instance 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.

**Sokoban (SOKO)** is a game puzzle developed by the Japanese company *Thinking Rabbit, Inc.* in 1982. *Sokoban* means "warehouse-keeper" in Japanese. Each puzzle consists of a room layout (a number of square fields representing walls or parts of the floor, some of which are marked as storage space) and a starting situation (one sokoban and a number of boxes, all of which must reside on some floor location, where one box occupies precisely one location and each location can hold at most one box). The goal is to move all boxes onto storage locations. To this end, the sokoban can walk on floor locations (unless occupied by some box), and push single boxes onto unoccupied floor locations. Figure 6.4 shows a typical configuration involving two boxes, where grey fields are storage fields and black fields are walls.

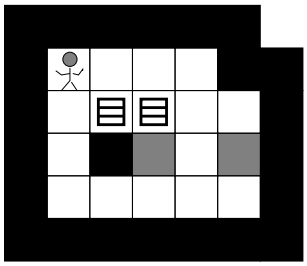

Figure 6.4: A Sokoban Instance

We have written a DLV program, reported in Appendix A.1.3, which finds solutions with a given number of push actions (where one push action can move a box over a number of fields, but always in the same direction) for a given puzzle together with a script which iteratively runs that DLV program with increasing numbers of push actions (starting at one) until some solution is found. This finds solutions with a minimal number of push actions.

The puzzle in Figure 6.4 is solvable with 6 push actions, so the script uses DLV to prove that no solutions with 1 to 5 push actions exist, and then to compute

a solution with 6 push actions.

### 6.2.3 Benchmark Data

We created random graph instances for HAMPATH using a tool[2] by Patrik Simons which has been used to compare Smodels against SAT solvers [Simons, 2000]. For each problem size $n$ we generated *ten* instances, always assuming node 0 as the starting node, and for each instance we stopped after the first solution had been found.

The blocksworld problems P1 to P4 have been employed in [Erdem, 1999] to compare DLP systems, and can be solved in 4, 6, 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 generated 8 random permutations of the full input (including the encoding). In addition, we also tried to solve each of these problems with one step less than required, which fails to produce any plan but shows the minimality of the regular solutions. These instances are labeled P1-1, P2-1 and so forth in Figure 6.6.

A vast amount of Sokoban puzzles is available on the Internet in a simple ANSI text format. The examples we used for benchmarks are results of efforts to automatically generate hard puzzles. One set has been created by Yoshio Murase[3], the other set is due to Jacques Duthen[4]. The puzzle in Figure 6.4 is number 2 of Duthen's instances.

### 6.2.4 Experimental Results

All reported experiments have been carried out on a Pentium III/1GHz machine with 512MB of main memory, running SuSE GNU/Linux (kernel 2.4.21). The different DLV executables have been built with the GCC compiler (version 3.2.2).

For each invocation of DLV we allowed a maximum run-time of 600 seconds. For SOKO there may be several invocations per problem instance, so the total reported time may be more than 600 seconds.

Results for HAMPATH are shown in Figure 6.5. It is easy to see that both **ifPoss** and **ifNeed** perform similarly to **Old** for very small problem instances, but scale tremendously better and are able to efficiently deal with graphs of 120 nodes, whereas **Old** is not able to solve almost all problems with more than 60 nodes

[2] `http://www.tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`
[3] `http://www.ne.jp/asahi/ai/yoshio/sokoban/auto52/`
[4] `http://hem.passagen.se/awl/ksokoban/sokogen-990602.skm`

| | | Average | | | Maxima | |
|---|---|---|---|---|---|---|
| | **Old** | **ifPoss** | **ifNeed** | **Old** | **ifPoss** | **ifNeed** |
| 10 | 0.02 | 0.02 | 0.04 | 0.02 | 0.03 | 0.10 |
| 20 | 0.05 | 0.06 | 0.05 | 0.07 | 0.08 | 0.05 |
| 30 | 0.08 | 0.09 | 0.09 | 0.15 | 0.12 | 0.09 |
| 40 | 0.12 | 0.14 | 0.13 | 0.13 | 0.16 | 0.14 |
| 50 | 55.59 | 0.20 | 0.19 | 443.58 | 0.23 | 0.20 |
| 60 | 11.52 | 0.29 | 0.27 | 86.54 | 0.32 | 0.29 |
| 70 | - | 0.38 | 0.35 | - | 0.42 | 0.37 |
| 80 | - | 0.50 | 0.47 | - | 0.54 | 0.50 |
| 90 | - | 0.66 | 0.60 | - | 0.80 | 0.65 |
| 100 | - | 0.83 | 0.85 | - | 1.09 | 1.11 |
| 110 | - | 1.02 | 1.01 | - | 1.25 | 1.22 |
| 120 | - | 17.77 | 16.86 | - | 116.62 | 110.32 |

Figure 6.5: Hamiltonian Path Running Times

within the allowed time. **ifPoss** and **ifNeed** scale much better, their behavior is similar here, with **ifNeed** being slightly faster than **ifPoss**.

The programs for HAMPATH have highly cyclic HCF dependency graphs. Thus, **ifPoss** and **ifNeed** can exploit the pruning power of the Well-founded operator, significantly outperforming **Old** which employs only Fitting's operator. On the other hand, the dependency graphs of these programs usually have one big component containing nearly all atoms. Therefore, there are only few differences (but still noticeable) between **ifPoss** and **ifNeed**, as the latter cannot avoid many calls to the Well-founded operator.

For BW, **Old**, **ifPoss** and **ifNeed** are nearly equivalent, and all three approaches seem to scale similarly. We explain this as follows: These programs have only few cyclic HCF components while most components are acyclic. Moreover, these few cyclic components are also very small, and the Well-founded operator does not bring a relevant gain in terms of pruning compared to Fitting's operator, so the three implementations show essentially the same behavior.

SOKO, finally, shows that both **ifPoss** and **ifNeed** perform significantly better than **Old**, which fails to solve more than 60% of all problems instances and usually takes one or two orders of magnitude longer to solve the remaining ones (see Figure 6.7).

As can be verified on the data reported in Appendix A.2, both for the Yoshio Murase and the Jacques Duthen sets, **ifNeed** yields an average speedup of about

| | **Old** | **ifPoss** | **ifNeed** |
|---|---|---|---|
| P1 -1 | 0.03 | 0.03 | 0.03 |
| P2 -1 | 0.05 | 0.05 | 0.06 |
| P3 -1 | 2.35 | 2.37 | 2.40 |
| P4 -1 | 1.28 | 1.31 | 1.32 |
| P5 -1 | 15.86 | 15.90 | 15.94 |
| P6 -1 | 262.64 | 263.45 | 263.65 |
| P1 | 0.04 | 0.04 | 0.04 |
| P2 | 0.07 | 0.08 | 0.08 |
| P3 | 5.97 | 5.99 | 5.99 |
| P4 | 14.53 | 14.68 | 14.67 |
| P5 | 259.65 | 261.40 | 261.96 |
| P6 | 234.62 | 235.00 | 235.45 |

Figure 6.6: Blocksworld - Average Running Times

| | Yoshio Murase | | Jacques Duthen | |
|---|---|---|---|---|
| | solved | unsolved | solved | unsolved |
| **Old** | 9 | 43 | 36 | 42 |
| **ifPoss** | 40 | 12 | 68 | 10 |
| **ifNeed** | 40 | 12 | 68 | 10 |

Figure 6.7: Soko - Instances Solved

6% over **ifPoss**, the maximum speedup being about 10% (instances 14 and 33 for Yoshio Murase and Jacques Duthen, respectively). For this class of benchmarks, the potential gain brought about by avoiding useless calls to $\mathcal{W}_\mathcal{P}$ is evident.

In summary, the experiments show that both **ifPoss** and **ifNeed** are strictly preferable to **Old**; and that of these two, **ifNeed** shows a measurable speedup on a wide range of examples. Therefore, recent DLV releases employ **ifNeed** by default, even if a command-line option easily allows the user to switch it off, thus making DLV use only the original implementation of *Fitting*'s operator.

# Part III

# Extending Disjunctive Logic Programming

In this part we present two extensions of DLP, namely *Template Predicates* and *External Predicates*. Thus, the part is actually divided into two main pieces. In the first, we introduce the *template* paradigm into DLP, providing syntax and giving a clear operational semantics, and discussing theoretical properties of the extension and its main advantages; in the second, we introduce a formal framework for accommodating *external predicates* in the context of Disjunctive Logic Programming, showing how it enhances the applicability of DLP to a variety of problems.

More in details, the part is structured as follows.

- In Chapter 7 we introduce template predicates and provide the syntax of the language DLP$^T$. Then we show the features of DLP$^T$, with the help of some examples. The semantics of DLP$^T$is formally introduced, and some theoretical properties are discussed. We present also an implementation of the DLP$^T$ language on top of the DLV system.

- Chapter 8 introduces external predicates, beside some example motivations. Then formally present the framework, named DLP-EX, for accommodating *external predicates* in the context Disjunctive Logic Programming. We present the semantics and discuss some theoretical properties of the DLP-EX language; since it includes the explicit possibility of inventing new values from external sources, and this setting could lead to non-termination of any conceivable evaluation algorithm, we tailor specific cases where decidability is preserved. The chapter presents then our implementation of the framework into the DLV system, and some experiments. An overview on the related works in the literature is also given.

# Chapter 7

# Templates

Although current DLP systems have been extended in many directions, they still miss features which may be helpful towards industrial applications, like the capability of quickly introducing new predefined constructs or of dealing with modules. Indeed, in spite of the fact that a wide literature about modular logic programming is known, code reusability has never been considered as a critical point in Disjunctive Logic Programming. We extend the Disjunctive Logic Programming, under the answer set semantics, with the notion of 'template' predicates, which are introduced in this chapter.

## 7.1 Introducing Templates

It is very likely that this new generation of DLP applications require the introduction of repetitive pieces of standard code. Indeed, a major need of complex and huge DLP applications such as [Nogueira *et al.*, 2001] is dealing efficiently with large pieces of such a code and with complex data structures, more sophisticated than the simple, native DLP data types.

Indeed, the non-monotonic reasoning community has continuously produced, in the past, several extensions of nonmonotonic logic languages, aimed at improving readability and easy programming through the introduction of new constructs, employed in order to specify classes of constraints, search spaces, data structures, new forms of reasoning, new special predicates [Cadoli *et al.*, 1999; Eiter *et al.*, 1997a; Kuper, 1990], such as aggregate predicates [Calimeri *et al.*, 2005].

Nonetheless, code reusability has never been considered as a priority in the Answer Set Programming/DLP field, despite the fact that modular logic pro-

gramming has been widely studied in the general case [Bugliesi *et al.*, 1994; Eiter *et al.*, 1997d].

The language DLP$^T$ we propose here has basically two purposes. First, DLP$^T$ moves the DLP field towards industrial applications, where code reusability is a crucial issue. Second, DLP$^T$ aims at minimizing developing times in DLP system prototyping. DLP systems developers wishing to introduce new constructs are enabled to fast prototype their languages, make their language features quickly available to the scientific community, and successively concentrate on efficient (and long lasting) implementations. To this end, it is necessary a sound specification language for new DLP constructs. DLP itself proves to fit very well for this purpose.

The proposed framework introduces the concept of ''template'' predicate, whose definition can be exploited whenever needed through binding to usual predicates.

Template predicates can be seen as a way to define intensional predicates by means of a subprogram, where the subprogram is generic and reusable. This eases coding and improves readability and compactness of DLP programs:

**Example 7.1** The following template definition

```
#template max[p(1)](1)
{
exceeded(X) :- p(X),p(Y), Y > X.
max(X) :- p(X), not exceeded(X).
}
```

introduces a generic template program, defining the predicate `max`, intended to compute the maximum value over the domain of a generic unary predicate `p`. A template definition may be instantiated as many times as necessary, through *template atoms*, as in the following sample program.

```
:- max[weight(*)](M), M > 100.          (a)
:- max[student(Sex,$,*)](M), M >25.     (b)
```

Template definitions may be unified with a template atom in many ways. The above program contains two invocations: a *plain* invocation (`a`), and a *compound*

invocation (`b`). The latter allows to employ the definition of the template predicate `max` on a ternary predicate, discarding the second attribute of `student`, and grouping by values of the first attribute.

The operational semantics of the language is defined through a suitable algorithm (actually, a *pseudo*-algorithm, as we will see in Chapter 7.4) which is able to produce, from a set of nonrecursive template definitions and a DLP$^T$ program, an equivalent DLP program. There are some important theoretical questions to be addressed, such as the termination of the algorithm, and the expressiveness of the DLP$^T$ language. Indeed, we prove that it is guaranteed that DLP$^T$ program encodings are as efficient as plain DLP encodings, since unfolded programs are just polynomially larger with respect to the originating program.

The DLP$^T$ language has been successfully implemented and tested on top of the DLV system [Faber *et al.*, 2001]. Anyway, the proposed paradigm does not rely at all on DLV special features, and is easily generalizable. In sum, benefits of the DLP$^T$ language are: improved declarativity and succinctness of the code; code reusability and possibility to collect templates within libraries; capability to quickly introduce new, predefined constructs; fast language prototyping.

## 7.2 Syntax of the DLP$^T$ language

A DLP$^T$ program is a DLP program where (possibly negated) *template atoms* may appear in rules and constraints. Definition of template atoms is next provided.

**Definition 7.2** A *template definition* $D$ consists of:

- a template header,

$$\#\mathbf{template}\ n_D[f_1(b_1)\ ,\ \ldots\ ,\ f_n(b_n)](b_{n+1})$$

  where $b_1, \ldots, b_{n+1}$ are (nonnegative) integer values, and $f_1, \ldots, f_n$ are predicate names (*formal predicates*, from now on). $n_D$ is called *template name*;

- an associated DLP$^T$ subprogram enclosed in curly braces; $n_D$ may be used within the subprogram as predicate of arity $b_{n+1}$, whereas the predicates $f_i, \ldots, f_n$ are intended to be of arity $b_i, \ldots, b_n$, respectively. At least a rule having $n_D$ within its head must appear in the subprogram.

**Example 7.3** Beside the one introduced in Example 7.1, another valid template definition is the following:

```
#template subset[p(1)](1)
{
subset(X) v  non_subset(X) :- p(X).
}
```

Intuitively, this defines a subset of the predicate 'p'; such a subset is non-deterministically chosen by means of disjunction.

**Definition 7.4** A *template atom* $t$ is of the form:

$$n_t[p_1(\mathbf{X}_1)\ ,\ \ldots\ ,\ p_n(\mathbf{X}_n)](\mathbf{A})$$

where $p_1, \ldots, p_n$ are predicate names (namely, *actual* predicates), and $n_t$ is a template name. $\mathbf{X}_i, \ldots, \mathbf{X}_n$ are lists of *special* terms (referred in the following as *special* lists of terms), where $\mathbf{A}$ is a list of standard terms.

A special term is either a standard term, or a dollar ('$') symbol (from now on, *projection term*) or a star ('*') symbol (from now on, *parameter term*).

$p_1(\mathbf{X}_i), \ldots, p_n(\mathbf{X}_n)$ are called *special* atoms. $\mathbf{A}$ is called *output list*.

Given a template atom $t$, let $D(t)$ be the corresponding template definition having the same template name. It is assumed there is a unique definition for each template name.

**Example 7.5** Some template atoms are

```
max[company($,State,*)](Income). subset[node(*)](X).
```

Template atoms may "instantiate" template definitions as many times as necessary.

**Example 7.6** The following short piece of program contains multiple instantiation of the 'max' template, whose definition has been introduced in Example 7.1:

```
:- max[weight(*)](M), M > 100.
:- max[student(Sex,$,*)](M), M > 25.
```

Looking at Example 7.5 and Example 7.6, we can get some intuitions on ('$' and '*' symbols). Basically, projection terms ('$' symbols) are intended to indicate which attributes, among those belonging to an actual predicate, have to be ignored. A standard term (a constant or a variable) within an actual atom indicates a 'group-by' attribute, whereas parameter terms ('*' symbols) indicate which attributes have to be considered as parameters.

Thus, the intuitive meaning of the first template atom of example 7.5 is to compute the companies with the maximum value of the 'income' attribute (the third attribute of the *company* predicate), grouped by the 'state' attribute (the second one), ignoring the first attribute. The computed values of $Income$ are returned through the output list.

**Example 7.7** Given a database by means of facts like

```
emp_companyA("Jones",30000,35,"Accounting").
 [...]
emp_companyB("Miller",34000,29,"Marketing").
```

the following single-rule program

```
emp_companyAB(Name) :- intersection[emp_companyA](*,$,$,$),
  emp_companyB(*,$,$,$)](Name).
```

computes the employees working for both company A and company B. It exploits the template 'intersection', defined in Section 7.3, and again shows how '$' and '*' symbols can be used. The last three attributes (name, salary, department) are thus ignored, by meaning of '$' symbols, while the first (name) is intended as parameter, by meaning of '*' symbol.

## 7.3 Knowledge Representation by DLP$^T$

In this section we show by examples the main advantages of template programming. Examples point out the provision of a succinct, elegant and easy-to-use way for quickly introducing new constructs through the DLP$^T$ language.

**Aggregates.** Aggregate predicates [Ross and Sagiv, 1997], allow to represent properties over sets of elements. Aggregates or similar special predicates have been already studied and implemented in several DLP solvers [Dell'Armi *et al.*, 2003; Simons, 2000]: the next example shows how to fast prototype aggregate semantics without taking into account of the efficiency of a built-in implementation. Here we take advantage of the template predicate `max`, defined in Example 7.1. The next template predicate defines a general program to count distinct values of a predicate `p`, given an order relation `succ` defined on the domain of `p`. We assume the domain of integers is bounded to some finite value.

```
#template count[p(1),succ(2)](1)
{
partialCount(0,0).
partialCount(I,V) :- not p(Y), I=Y+1,
  partialCount(Y,V).
partialCount(I,V2) :- p(Y), I=Y+1,
  partialCount(Y,V), succ(V,V2).
partialCount(I,V2) :- p(Y),I=Y+1,
  partialCount(Y,V), max[succ(*,$)](V2).
count(M) :- max[partialCount($,*)](M).
}
```

The above template definition is conceived in order to count, in a iterative-like way, values of the $p$ predicate through the $partialCount$ predicate. A ground atom $partialCount(i, a)$ means that at the stage $i$, the constant $a$ has been counted up. The predicate $count$ takes the value which has been counted at the highest (i.e. the last) stage value. The above program is somehow involved and shows how difficult could be to simulate aggregate constructs in Disjunctive Logic Programming. Anyway, the use of templates allows to write it once, and reuse it as many times as necessary.

It is worth noting how template `max` is employed over the binary predicate `partialCount`, instead of an unary one. Indeed, the `'$'` and `'*'` symbols are employed to project out the first argument of `partialCount`. The last rule is equivalent to the piece of code:

```
partialCount'(X) :- partialCount(_,X).
count(M) :- max[partialCount'(*)](M).
```

**Definition of ad hoc search spaces.** Template definitions can be employed to introduce and reuse constructs defining the most common search spaces. This improves declarativity of DLP programs to a larger extent. The next two examples show how to define a predicate `subset` and a predicate `permutation`, ranging, respectively, over subsets and permutations of the domain of a given predicate `p`. Such kind of constructs enriching plain Datalog languages have been proposed, for instance, in [Laenens *et al.*, 1990; Cadoli and Schaerf, 2001].

```
#template subset[p(1)](1)
{
subset(X) v non_subset(X) :- p(X).
}

#template permutation[p(1)](2).
{
permutation(X,N) v npermutation(X,N) :- p(X),
  #int(N), count[p(*),>(*,*)](N1), N<=N1.
:- permutation(X,A),permutation(Z,A), Z <> X.
:- permutation(X,A),permutation(X,B), A <> B.
covered(X) :- permutation(X,A).
:- p(X), not covered(X).
}
```

The explanation of the `subset` template predicate (already appeared in Example 7.3 is quite straightforward. As for the `permutation` definition, a ground atom $permutation(x, i)$ tells that the element $x$ (taken from the domain of $p$), is in position $i$ within the currently guessed permutation. The rest of the template subprogram forces permutations properties to be met.

Next we show how `count` and `subset` can be exploited to succinctly encode the *k-clique* problem [Garey and Johnson, 1979], i.e., given a graph $G$ (represented by predicates `node` and `edge`), find if there exists a complete subgraph containing at least $k$ nodes (we consider here the 5-clique problem):

```
in_clique(X) :- subset[node(*)](X).
:- count[in_clique(*),>(*,*)](K), K < 5.
:- in_clique(X),in_clique(Y), X <> Y,
  not edge(X,Y).
```

The first rule of this example guesses a clique from a subset of nodes. The first constraint forces a candidate clique to be at least of `5` nodes, while the last forces a candidate clique to be strongly connected. The `permutation` template can be employed, for instance, to encode the Hamiltonian Path problem (see Chapter 3.2.2): given a graph $G$, find a path visiting each node of $G$ exactly once.

```
path(X,N) :- permutation[node(*)](X,N).
:- path(X,M), path(Y,N), not edge(X,Y),
  M = N+1.
```

The following template may be employed in order to (non-deterministically) select exactly one value from the domain of a predicate $p$. It is built on top of the `subset` predicate.

```
#template any[p(1)](1)
{
any (X) :- subset[p(*)](X).
:- any(X), any(Y), X <> Y.
:- p(X), not any(X).
}
```

**Handling of complex data structures.** DLP$^T$ can be fruitfully employed to introduce operations over complex data structures, such as sets, dates, trees, etc.

*Sets*: Extending Datalog with Set programming is another matter of interest for the DLP field. This topic has been already discussed (e.g. in [Leone and Rullo, 1993; Kuper, 1990]), proposing some formalisms aiming at introducing a suitable semantics with sets. It is fairly quick to introduce set primitives using DLP$^T$; a set $S$ is modeled through the domain of a given unary predicate $s$. Intuitive constructs like `intersection`, `union`, or `symmetricdifference`, can be modeled as follows.

```
#template intersection[a(1),b(1)](1).
{
intersection (X) :- a(X),b(X).
}
```

```
#template union[a(1),b(1)](1).
{
union(X) :- a(X).
union(X) :- b(X).
}

#template symmetricdifference[a(1),b(1)](1)
{
symmetricdifference(X) :- union[a(*),b(*)](X),
  not intersection[a(*),b(*)](X).
}
```

*Dates*: managing time and date data types is another important issue in engineering applications of DLP. For instance, in [Ianni *et al.*, 2003], it is very important to reason on compound records containing date values. The following template shows how to compare dates represented through a ternary relation ⟨day, month, year⟩.

```
#template before[date1(3),date2(3)](6)
{
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D1,M1,Y1), Y<Y1.
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D1,M1,Y1), Y==Y1, M<M1.
before(D,M,Y,D1,M1,Y1) :- date1(D,M,Y),
  date2(D,M1,Y1), Y==Y1,M==M1,D<D1.
}
```

## 7.4 Semantics of the DLP$^T$ language

The semantics of the DLP$^T$ language is given through a suitable ''explosion'' algorithm.

**Remark 7.8** It is worth noting that, as already briefly mentioned, and more deeply discussed in Section 7.5, the ''explosion'' algorithm is actually a *pseudo*-algorithm, since it might not terminate in some cases. Nevertheless, we do prefer to keep the term *algorithm* also in the following.

It is given a DLP$^T$ program $P$. The aim of the *Explode* algorithm, introduced next, is to remove template atoms from $P$. Each template atom $t$ is replaced with a standard atom, referring to a fresh intensional predicate $p_t$. The subprogram $d_t$,

defining the predicate $p_t$, is computed taking into account of the template definition $Def(t)$ associated to $t$. Actually, many template atoms may be grouped and associated to the same subprogram. The concept of atom signature, introduced next, helps in finding groups of equivalent template atoms. The final output of the algorithm is a DLP program $P'$. Answer sets of the originating program $P$ are constructed, *by definition*, from answer sets of $P'$. Throughout this section, we will refer to Example 7.1 as running example. By little abuse of notation, $a \in P$ (resp. $a \in r$) means that the atom $a$ appears in the program $P$ (the rule $r$, respectively).

**Definition 7.9** Given a template atom $t$, the corresponding *template signature* $s(t)$ is obtained from $t$ by replacing each standard term with a conventional (mute variable) '_' symbol. Let $Def(s(t))$ be the template definition associated to the signature $s(t)$; Given a DLP$^T$ program $P$, let $At(P)$ be the set of template atoms occurring in $P$. Let $Sig(At(P))$ be the set of signatures $\{s(t) : t \in At(P)\}$.

For instance, `max[p(*,S,$)](M)` and `max[p(*,a, $)](H)` have the same signature, namely `max[p(*, _,$)](_)`.

### 7.4.1 The Explode algorithm

The **Explode** algorithm ($\mathcal{E}$ in the following) is sketched in Figure 7.4.1. It is given a DLP$^T$ program $P$ and a set of template definitions $T$. The output of $\mathcal{E}$ is a DLP program $P'$. $\mathcal{E}$ takes advantage of a stack of signatures $S$, which contains the set of signatures to be processed; $S$ is initially filled up with each template signature occurring within $P$.

The purpose of the main loop of $\mathcal{E}$ is to iteratively apply the $\mathcal{U}$ (Unfold) operation to $P$, until $S$ is empty. Given a signature $s$, the $\mathcal{U}$ operation generates from the template definition $Def(s)$ a DLP$^T$ program $P^s$ which defines a fresh predicate $t^s$, where $t$ is the template name of $s$. Then, $P^s$ is appended to $P$; furthermore, each template atom $a \in P$, such that $a$ has signature $s$, is replaced with a suitable atom $a^s(\mathbf{X}')$. It is important pointing out that, if $P^s$ contains template atoms, the unfolding operation updates $S$ with new template signatures.

We show next how $P^s$ is constructed and template atoms are removed.

Let the header of $Def(s)$ be

$$\#\mathbf{template}\ t[f_1(b_1)\,,\ \ldots\ ,\ f_n(b_n)](b_{n+1})$$

```
Explode(Input: aDLP^T program P,a set of
                template definitions T.
        Outputs: an updated version of P'
                of P in DLP form.
        Data Structures: a queue S )
begin
        put each s ∈ Sig(At(P)) in S;
        P' = P;
        while ( S is not empty ) do begin
                extract a template signature s from S;

                //Start of the U (Unfold) operation;
                  construct P^s (see Subsection 7.4.2),
                        then set P = P ∪ P^s;
                  put all the s' ∈ Sig(At(P^s)) in S;
                for each template atom a ∈ P
                    if a has signature s
                      construct the standard atom
                            a^s(X') (see Subsection 7.4.3);
                      replace a with a^s(X') in P;
                //End of the U operation;

        end;
end.
```

Figure 7.1: The Explode ($\mathcal{E}$) Algorithm

Let $s$ be of the form

$$t[p_1(\mathbf{X}_1)\ ,\ \ldots\ ,\ p_n(\mathbf{X}_n)](\mathbf{X}_{n+1})$$

Given a special list $\mathbf{X}$ of terms, let $\mathbf{X}[j]$ denote the $j^{th}$ term of $\mathbf{X}$; let $fr(\mathbf{X})$ be a list of $|\mathbf{X}|$ fresh variables $F_{\mathbf{X},1}, \ldots, F_{\mathbf{X},|\mathbf{X}|}$; let $st(\mathbf{X}), pr(\mathbf{X})$ and $pa(\mathbf{X})$ be the sublist of (respectively) standard, projection and parameter terms within $\mathbf{X}$. Given two lists $\mathbf{A}$ and $\mathbf{B}$, let $\mathbf{A}\&\mathbf{B}$ be the list obtained appending $\mathbf{B}$ to $\mathbf{A}$.

### 7.4.2 How $P^s$ is constructed.

The program $P^s$ is built in two steps. On the first step, $P^s$ is enriched with a set of rules, intended in order to deal with projection variables.

For each $p_i \in s$, we introduce a predicate $p_i^s$ and we enrich $P^s$ with the auxiliary rule $p_i^s(\mathbf{X}_i') \text{:-} p_i(\mathbf{X}_i'')$, where:

- $\mathbf{X}_i''$ is built from $\mathbf{X}_i$ substituting $pr(\mathbf{X}_i)$ with $fr(pr(\mathbf{X}_i))$, substituting $pa(\mathbf{X}_i)$ with $fr(pa(\mathbf{X}_i))$, and substituting $st(\mathbf{X}_i)$ with $fr(st(\mathbf{X}_i))$;
- $\mathbf{X}_i'$ is set to $fr(st(\mathbf{X}_i))\&fr(pa(\mathbf{X}_i))$.

For instance, given the signature

$s_2 = max[student(\_,\$,*)](\_)$

and the example template definition given in Example 7.1, let $\mathbf{L}$ be the list $\langle \_, \$, * \rangle$; it is introduced the rule:

$$student^{s_2}(F_{st(\mathbf{L}),1}, F_{pa(\mathbf{L}),1}) : -student(F_{st(\mathbf{L}),1}, F_{pr(\mathbf{L}),1}, F_{pa(\mathbf{L}),1}).$$

Note that projection variables are filtered out from $student^s$. In the second step, for each rule $r$ belonging to $D(s)$, we create an updated version $r'$ to be put in $P^s$, where each atom $a \in r$ is modified this way:

- if $a$ is $f_i(\mathbf{Y})$ where $f_i$ is a formal predicate, it is substituted with the atom $p_i^s(\mathbf{Y}')$. $\mathbf{Y}'$ is set to $fr(st(\mathbf{X}_i))\&\mathbf{Y}$;
- if $a$ is a either a standard (included atoms having $t$ as predicate name) or a special atom (in this latter case $a$ occurs within a template atom) $p(\mathbf{Y})$, it is substituted with an atom $p^s(\mathbf{Y}')$, where

$$Yvect' = fr(st(\mathbf{X}_1))\& \ldots \&fr(st(\mathbf{X}_n))\&\mathbf{Y}.$$

**Example 7.10** For instance, consider the rule

$$max(X) \text{:-}\ p(X), not\ exceeded(X).$$

from Example 7.1, and the signature

$s_2 =$ `max[student(_,$,*)](_);`

let $\mathbf{L}$ be the special list $\langle \_, \$, * \rangle$; according to the steps introduced above, this rule is translated to

$$max^{s_2}(F_{\mathbf{L},1}, X) \text{:-}\ student^{s_2}(F_{\mathbf{L},1}, X), not\ exceeded^{s_2}(F_{\mathbf{L},1}, X).$$

### 7.4.3 How template atoms are replaced

Consider a template[1] atom in the form

$$t[p_1(\mathbf{X}_1)\,,\,\ldots\,,\,p_n(\mathbf{X}_n)](\mathbf{X}_{n+1}).$$

It is substituted with

$$t^s(\mathbf{X}')$$

where

$$\mathbf{X}' = st(\mathbf{X}_1)\&\ldots\&st(\mathbf{X}_n)\&\mathbf{Y}.$$

**Example 7.11** The complete output of $\mathcal{E}$ on the constraint

$$:\!- max[student(\_, \$, *)](M), M > 25.$$

coupled with the template definition of `max` given in Example 7.1 is:

$$\begin{array}{rcl}
student^{s_2}(S_1, P_1) & :\!- & student(S_1, _, P_1). \\
exceeded^{s_2}(F_{\mathbf{L},1}, X) & :\!- & student^{s_2}(F_{\mathbf{L},1}, X), \\
 & & student^{s_2}(F_{\mathbf{L},1}, Y), \\
 & & Y > X. \\
max^{s_2}(F_{\mathbf{L},1}, X) & :\!- & student^{s_2}(F_{\mathbf{L},1}, X), \\
 & & not\ exceeded^{s_2}(F_{\mathbf{L},1}, X). \\
 & :\!- & max^{s_2}(Sex, M), M > 25.
\end{array}$$

We are now able to give the formal semantics of DLP$^T$. It is important highlighting that answer sets of a DLP$^T$ program are, by definition, constructed in terms of answer sets of an equivalent DLP program.

**Definition 7.12** Given a DLP$^T$ program $P$, and a set of template definitions $T$, let $P'$ the output of the *Explode* algorithm on input $\langle P, T\rangle$. Let $H(P)$ be the Herbrand base of $P'$ restricted to those atoms having predicate name appearing in $P$. Given an answer set $m \in M(P')$, then we define $H(P) \cap m$ as an answer set of $P$.

Note that the Herbrand base of a DLP$^T$ program is defined in terms of the Herbrand base of a DLP program *which is not* the output of $\mathcal{E}$.

[1]Depending on the form of $D(s)$, some template atom might not to be allowed, since some atom with same predicate name but with mismatched arities could be generated.

## 7.5 Theoretical properties of $\text{DLP}^T$

The explosion algorithm replaces template atoms from a $\text{DLP}^T$ program $P$, producing a DLP program $P'$. It is very important to investigate about two theoretical issues:

- Finding whether and when $\mathcal{E}$ terminates; in general, we observe that $\mathcal{E}$ might not terminate, for instance, in case of recursive template definitions. Anyway, we prove that it can be decided in polynomial time whether $\mathcal{E}$ terminates on a given input.

- Establishing whether $\text{DLP}^T$ programs are encoded as efficiently as DLP programs. In particular, we are able to prove that $P'$ is polynomially larger than $P$. Thus $\text{DLP}^T$ keeps the same expressive power as DLP. This way, we are guaranteed that $\text{DLP}^T$ program encodings are as efficient as plain DLP encodings, since unfolded programs are always reasonably larger with respect to the originating program.

**Definition 7.13** It is given a $\text{DLP}^T$ program $P$, and a set of template definitions $T$. The *dependency graph* $G_{T,P} = \langle V, E \rangle$ of $T$ and $P$ is a graph encoding dependencies between template atoms and template definitions, and it's built as follows. Each template definition $t \in T$ will be represented by a corresponding node $v_t$ of $V$. $V$ contains a node $v_P$ associated to $P$ as well. $E$ will contain a direct edge $(v_t, v_{t'})$ if the template $t$ contains a template atom referring to the template $t'$ inside its subprogram (as for the node referred to $P$, we consider the whole program $P$). Let $G_{T,P}(u) \subseteq G_{T,P}$ be the subgraph containing nodes and arcs of $G_{T,P}$ reachable from $u$.

**Lemma 7.14** It is given a $\text{DLP}^T$ program $P$, and a set of template definitions $T$. Let $v_P$ the node of $G_{T,P}$ corresponding to $P$. If $G_{T,P}(v_P)$ is acyclic then $\mathcal{E}$ terminates whenever applied to $P$ and $T$.

**Proof.** We assume $G_{T,P}(v_P) = \langle N, E \rangle$ is acyclic. we can state a partial ordering $\gg$ between its nodes, such that for each $v, v' \in N, v \gg v'$ iff either $(v, v') \in E$ or there is a $v''$ such that $v \gg v''$ and $v'' \gg v$.

We can build a total ordering $\succ$ by extending $\gg$ in a way that, whenever neither $v \gg v'$ nor $v \gg v'$ holds, it is chosen appropriately whether $v \succ v'$ or $v' \succ v$ holds. This can be done, for instance, by performing an in-depth visit of $G_{T,P}(v_P)$ and taking the resulting order of visit.

Let $level(v)$ be defined as follows:

- $level(v) = 0$ if there is no $v'$ such that $v' \succ v$;
- for $i > 0$, $level(v) = i$ if $i$ is the maximum value such that there is a $v'$ such that $level(v') = i - 1$ and $v \succ v'$.

Note that $level(v) > level(v')$ iff $v \succ v'$.

Given a queue $S$ of signatures, let $level(S)$ be $\max_{Def(s)|s \in S} level(v_{Def(s)})$.

We will assume $S$ is managed as a priority queue such that an element $s \in S$ having better value of $level(v_{Def(s)})$ is extracted first[2].

Note that $\mathcal{E}$ has a main loop where at each iteration a signature $s$ is popped from $S$, whereas a new set of signatures $S'$ is put in $S$. A new signature $s' \in S'$ can be put on $S$ iff $Def(s) \succ Def(s')$. This means that $level(S)$ is non-increasing from one iteration to another.

$level(S)$ can stay unchanged from one iteration $i$ to the next iteration $i + 1$ only if there is some $s'$ such that $Def(s) = Def(s')$ still in $S$ at the beginning of iteration $i + 1$. But, in this case, during iteration $i + 1$, the cardinality of the set $\{s' \text{ s.t. } Def(s) = Def(s')\}$ is decreased by 1, since a new signature referring to the same template definition (and having same level) will be extracted from $S$.

Thus, there exists an iteration $j$, such that the difference $j - i$ has a maximum value bounded by $|\{s' \text{ s.t. } Def(s) = Def(s')\}|$, where $s$ is the signature extracted at iteration $i$.

$\mathcal{E}$ will terminate once $level(S)$ is 0 and $S$ is emptied up. □

**Theorem 7.15** It is given a DLP$^T$ program $P$, and a set of template definitions $T$. It can be decided in polynomial time whether $\mathcal{E}$ terminates when $P$ and $T$ are taken as input.

**Proof.** We observe that $G_{T,P}$ can be built in polynomial time. By Lemma 7.14 we can show that $\mathcal{E}$ terminates if $G_{T,P}(u_P)$ is acyclic. Vice versa $\mathcal{E}$ does not terminate if we assume there is a cycle in $G_{T,P}(u_P)$.

In order to show this, assume there is a cycle $C = \{u_{t_0}, u_{t_1}, \ldots, u_{t_k}, u_{t_0}\}$, with $k \geq 0$ in $G_{T,P}(u_P) = \langle N, E \rangle$.

Since any node of $N$ is reachable from $u_P$, we can assume that $\mathcal{E}$ either loops infinitely or does not terminate until some node $u_t$ such that $(u_t, u_{t_0}) \in E$ is reached, i.e. until $\mathcal{E}$ does not enters $C$ or a similar cycle. This means that $\mathcal{E}$ will

[2] Although this assumption can be relaxed, we prefer to introduce it in order to keep the line of reasoning of this proof clearer.

extract, during some iteration $j$, a signature $s$, such that $Def(s) = t$, from $S$, and then at least one $s'$ such that $Def(s') = t_0$ and $s' \in Sig(At(P^s))$ is added to $S$.

We can prove that starting from the iteration $j$ there is no iteration $j' > j$ such that $S = \emptyset$ at its beginning. $j'$ can exist only if at the iteration $j' - 1$ a remaining signature $s_{last}$ is extracted and nothing else is added to $S$. Define $S_C$ as the set of signatures such that $Def(s) \in \{t_0, \ldots, t_k\}$, that is, $S_C$ is the set of signatures corresponding to nodes appearing in $C$. $s_{last}$ cannot be member of $S_C$, because, in this case, an $s_{last}$ will generate new signatures to be added in $S$. However, once $C$ is reached, $S$ will always contain, at the beginning of any iteration $j' > j$, at least one element of $S_C$. Indeed, it cannot be avoided that once an element $s' \in S_C$ is extracted, new elements of $S_C$ are inserted during the iteration $j'$. □

**Definition 7.16** A set of template definitions $T$ is said *nonrecursive* if for any DLP$^T$ program $P$, the subgraph $G_{T,P}(v_P)$ is acyclic.

It is useful to deal with nonrecursive sets of template definitions, since they may be safely employed with any program. Checking whether a set of template definitions is nonrecursive is quite easy.

**Proposition 7.17** A set $T$ of template definitions is nonrecursive iff $G_{T,\emptyset}$ is acyclic.

**Proposition 7.18** Given a DLP$^T$ program $P$ and a nonrecursive set of template definitions $T$, the number of arcs of $G_{T,P}(u_P)$ is bounded by the overall size of $T$ and $P$, i.e., it is $O(|T| + |P|)$.

**Theorem 7.19** Given a DLP$^T$ program $P$ and a nonrecursive set of template definitions $T$, the output $P'$ of $\mathcal{E}$ on input $\langle P, T\rangle$ is polynomially larger than $P$ and $T$.

**Proof.** We first observe that each execution of $\mathcal{U}$ adds to $P$ a number of rules (or constraints) whose overall size is clearly bounded by the size of $T$ (see Figure 7.4.1). According to Lemma 7.14, if $T$ is nonrecursive, the number of $\mathcal{U}$ operations carried out by $\mathcal{E}$ is bounded by the maximum level $l$ (bounded by the number of nodes of $G_{T,P}(u_P)$, and thus by the size of $T$) which can be assigned to a node of $G_{T,P}(u_P)$, times the number of different template atoms that occur in $P$ and $T$. Thus, the size of $P'$ is $O(|T|^2(|T| + |P|))$. □

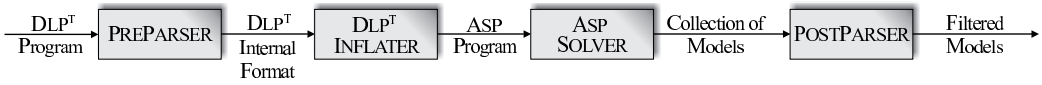


Figure 7.2: Architecture of the DLP$^T$ compiler

As already discussed in Chapters 1 and 2, and proved also in [Dantsin *et al.*, 2001], plain DLP programs (under the brave reasoning semantics) entirely capture the complexity class $\Sigma_2^P$. This bounds the expressive power of DLP$^T$, too. Indeed, as previously shown, DLP$^T$ programs may allow to express more succinct encodings of problems, w.r.t. DLP; but, despite this, the expressive power is not increased, accordingly to the following Corollary.

**Corollary 7.20** DLP$^T$ has the same expressive power as DLP.

**Proof.** The result is straightforward. Theorem 7.19 showed as unfolded DLP programs produced as the output of $\mathcal{E}$ are polynomially larger than the input programs. In addition, DLP$^T$ semantics is defined in terms of the equivalent, unfolded, DLP program. Thus, DLP$^T$ has the same expressiveness properties as DLP. □

## 7.6 System architecture and usage

The DLP$^T$ language has been implemented on top of the DLV system [Faber *et al.*, 1999; 2001; Faber and Pfeifer, since 1996]. The current version of the language is available through the DLP$^T$ Web page [Calimeri *et al.*, since 2003], and the overall architecture of the system is shown in Figure 7.2. The system work-flow is described in the following.

A DLP$^T$ program is sent to a DLP$^T$ pre-parser, which performs syntactic checks (included non-recursivity checks), and builds an internal representation of the DLP$^T$ program. The DLP$^T$ Inflater performs the *Explode* Algorithm and produces an equivalent DLV program $P'$; $P'$ is piped towards the DLV system. The answer sets $M(P')$ of $P'$, computed by DLV, are then converted in a readable format through the Post-parser module; the Post-parser filters out from $M(P')$ informations about internally generated predicates and rules.

The system introduces also some useful features in order to ease programming. For instance, the possibility to define some predicates as 'global', just specifying them in the template definition.

$$\#\textbf{template}\ n_D[f_1(b_1)\ ,\ \ldots\ ,\ f_n(b_n)](b_{n+1})$$
$$\quad\textbf{GLOBAL}\ g_1\ ,\ \ldots\ ,\ g_m$$

where $g_1, ..., g_m$ is a list of predicate symbols defined as global. This introduces the notion of *scope*. The notion is similar to traditional imperative languages, such as C++, where it is possible to mask global variables. Intuitively, the meaning of the local predicates results from the rules defined within the template body, while the meaning of the global predicates results from the rules belonging to the general program. We refer to function scope in the former case, and program scope in the latter.

**Example 7.21** In this template definition, we have `node` as a global predicate, while `coloring` is local, and `arc` is an argument.

```
#template coloring[arc(2)](2) GLOBAL node {
coloring(Country, red) v
  coloring(Country, green) v
  coloring(Country, blue) :- node(Country).
:- arc(Country1, Country2),
  coloring(Country1, CommonColor),
  coloring(Country2, CommonColor).
}
```

# Chapter 8

# External Predicates

We introduce here Disjunctive Logic Programming with External Predicates, a framework aimed at enabling DLP to deal with external sources of computation and called DLP-EX. This feature is realized by the introduction of 'parametric' *external predicates*, whose extension is not specified by means of a logic program but computed through external code. With respect to existing approaches it is explicitly addressed the issue of invention of new information coming from external predicates, in form of new, and possibly infinite, constant symbols. Several decidable restrictions of the language are identified as well as suitable algorithms for evaluating Disjunctive Logic Programs with external predicates. The framework paves the way to Disjunctive Logic Programming in several directions such as pattern manipulation applications, as well as the possibility to exploit function symbols.

We present also our successful implementation of DLP-EX into the DLV system, which is now enabled to make external program calls.

## 8.1 Introducing External Predicates

Despite the good results achieved by the current DLP systems, even state-of-the-art hardly deal with data types such as strings, natural and real numbers. Although simple, this data types bring two kinds of technical problems: first, they range over infinite domains; second, they need to be manipulated with primitive constructs which can be encoded in logic programming at the cost of compromising efficiency and declarativity. Furthermore, interoperability with other software is nowadays important, especially in the context of those Semantic Web applications aimed at managing external knowledge.

Thus, the introduction of external sources of computation in tight synergy with DLP solvers opens a variety of possible applications. We show next an example of these successful experiences.

The discovery of complex pattern repetitions in string databases plays an important role in genomic studies, and in general in the areas of knowledge discovery. Genome databases mainly consist of sets of strings representing DNA or protein sequences (biosequences) and most of these strings still require to be interpreted. In this context, discovering common patterns in sets of biologically related sequences is very important.

It turns out that specifying pattern search strategies by means of Answer Set Programming and its extensions is an appealing idea: constructs like strong and weak constraints, disjunction, aggregates may help an algorithm designer to fast prototype search algorithms for a variety of pattern classes.

Unfortunately, state-of-the-art DLP Solvers lack the possibility to deal in a satisfactory way with infinite domains such as strings or natural numbers. Furthermore, although very simple, such data types need of ad hoc manipulation constructs, which are typically difficult to be encoded and cannot be efficiently evaluated in logic programming.

So, in order to cope with these needs, one may conceive to properly extend answer set programming with the possibility of introducing *external* predicates. The extension of an external predicate can be efficiently computed by means of an intensional definition expressed using a traditional imperative language.

Thus, we might allow a pattern search algorithm designer to take advantage of Answer Set Programming facilities, but extended with special atoms such as e.g. *#inverse(S1,S2)* (true if *S1* is the inverse string of *S2*), *#strcat(S1,S2,S3)* (true if *S3* is equal to the concatenation of *S1* and *S2*), or *#hammingDistance(S1,S2,N)* (true if *S1* has $N$ differences with respect to *S2*). Note that it is desirable that these predicates introduce new values in the domain of a program whenever necessary. For instance, the semantics of *#strcat(a,b,X)* should be such that $X$ matches with the new symbol $ab$.

Provided with a suitable mechanism for defining external predicates, the authors of [Palopoli *et al.*, 2005] have been able to define and implement a framework allowing to specify and resolve genomic pattern search problems; the framework is based on automatically generating logic programs starting from user-defined extraction problems, and exploits disjunctive logic programming properly extended in order to enable the possibility of dealing with a large variety of pattern problems. The external built-in framework implemented into the DLV system is

essential in order to deal with strings and patterns.

## 8.2 Syntax of DLP-EX

Syntax of DLP-EX language is fundamentally the ‘‘usual’’, already presented in Chapter 1.1. We will address here some differences due to the novelties.

Let $\mathcal{U}$, $\mathcal{X}$, $\mathcal{E}$ and $\mathcal{P}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, *external predicate names*, and *ordinary predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{U}$) are denoted with first letter in upper case (resp., lower case); elements from $\mathcal{E}$ are usually prefixed with ‘#’. $\mathcal{U}$ will constitute the default *Herbrand Universe*. We will assume that any constant appearing in a program or generated by external computation is taken from $\mathcal{U}$, which is possibly infinite[1].

Elements from $\mathcal{U} \cup \mathcal{X}$ are called *terms*. An *atom* is a structure $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \mathcal{P} \cup \mathcal{E}$; $n \geq 0$ is the *arity* of the atom. Intuitively, $p$ is the predicate name. The atom is *ordinary*, if $p \in \mathcal{P}$, otherwise we call it *external atom*. A list of terms $t_1, \ldots, t_n$ is succinctly represented by $\bar{t}$. A positive *literal* is an atom, whereas a *negative literal* is $\mathrm{not}\, a$ where $a$ is an atom.

For example, *node(X)*, and *#succ(a,Y)* are atoms; the first is ordinary, whereas the second is an external atom.

A *rule* $r$ is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \text{ :- } \beta_1, \ldots, \beta_n, \mathrm{not}\, \beta_{n+1}, \ldots, \mathrm{not}\, \beta_m, \tag{8.1}$$

where $m \geq 0$, $k \geq 1$, $\alpha_1, \ldots, \alpha_k$, are ordinary atoms, and $\beta_1, \ldots, \beta_m$ are (ordinary or external) atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. $E(r)$ is the set of external atoms of $r$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then $r$ is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then $r$ is a *fact*; $r$ is *ordinary*, if it contains only ordinary atoms. A DLP-EX *program* is a finite set $P$ of rules; it is *ordinary*, if all rules are ordinary. Without loss of generality, we will assume $P$ has no constraints[2] and only ground facts.

[1] Also, we assume that constants are encoded using some finite alphabet $\Sigma$, i.e. they are finite elements of $\Sigma^*$.

[2] A constraint $\text{:-}\, B(r)$ can be easily simulated through the introduction of a corresponding standard rule *fail :- B(r),* $\mathrm{not}$ *fail*, where *fail* is a fresh predicate not occurring elsewhere in the program.

The dependency graph $G(P)$ of $P$ is built in the standard way by inserting a node $n_p$ for each predicate name $p$ appearing in $P$ and a directed edge $(p_1, p_2)$, labelled $r$, for each rule $r$ such that $p_2 \in B(r)$ and $p_1 \in H(r)$.

The following is a short DLP-EXprogram:

$$mustChangePasswd(Usr) \text{ :- } passwd(Usr, Pass), \\ \#strlen(Pass, Len), \#<(Len, 8).$$

## 8.3 Semantics of the DLP-EX language

We formally introduce here the semantics of DLP-EX, and discuss some theoretical properties. As for the syntax, we define the semantics of DLP-EX by generalizing what already introduced (Chapter 1.2), pointing out the differences.

In the sequel, we will assume $P$ is a DLP-EX program. The *Herbrand base* of $P$ with respect to $\mathcal{U}$, denoted $HB_{\mathcal{U}}(P)$, is the set of all possible ground versions of ordinary atoms and external atoms occurring in $P$ obtained by replacing variables with constants from $\mathcal{U}$. The grounding of a rule $r$, $grnd_{\mathcal{U}}(r)$, is defined accordingly, and the grounding of program $P$ by $grnd_{\mathcal{U}}(P) = \bigcup_{r \in P} grnd_{\mathcal{U}}(r)$.

An *interpretation* $I$ for $P$ is a couple $\langle S, F \rangle$ where:

- $S \subseteq HB_{\mathcal{U}}(P)$ contains only ordinary atoms; We say that $I$ (or by small abuse of notation, $S$) is a *model* of ordinary atom $a \in HB_{\mathcal{U}}(P)$, denoted $I \models a$ ($S \models a$), if $a \in S$.
- $F$ is a mapping associating with every external predicate name *#e* $\in \mathcal{E}$, a decidable $n$-ary Boolean function (which we will call *oracle*) $F($*#e*$)$ assigning each tuple $(x_1, \ldots, x_n)$ either 0 or 1, where $n$ is the fixed arity of *#e*, and $x_i \in \mathcal{U}$. $I$ (or by small abuse of notation, $F$) is a *model* of a ground external atom $a =$ *#e*$(x_1, \ldots, x_n)$, denoted $I \models a$ ($F \models a$), if $F($*#e*$)(x_1, \ldots, x_n) = 1$.

A positive literal is modeled if its atom is modeled, whereas a negated literal is modeled if its corresponding atom is not modeled.

**Example 8.1** We give an interpretation $I = \langle S, F \rangle$ such that the external predicate *#strlen* is associated to the oracle $F($*#strlen*$)$, and $F(\#<)$ to $\#<$. Intuitively these oracles are defined such that *#strlen(pat4dat,7)* and $\#<$*(7,8)* are modeled by $I$, whereas *#strlen(mypet,8)* and $\#<$*(10,8)* are not.

The following is a ground version of rule 8.2:

$$mustChangePasswd(frank) \text{ :- } passwd(frank, pat4dat), \\ \#strlen(pat4dat, 7), \#<(7, 8).$$

Let $r$ be a ground rule. We define

i. $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$;

ii. $I \models, B(r)$ iff $I \models, a$ for each atom $a \in B^+(r)$ and $I \not\models, a$ for each atom $a \in B^-(r)$;

iii. $I \models r$ (i.e., $r$ is satisfied) iff $I \models H(r)$ whenever $I \models B(r)$.

We say that $I$ is a *model* of a DLP-EX program $P$ with respect to a universe $\mathcal{U}$, denoted $I \models_{\mathcal{U}} P$, iff $I \models r$ for all $r \in grnd_{\mathcal{U}}(P)$. A model $M$ is minimal if there is no model $N$ such that $N \subset M$.

Given a general ground program $P$, we can give a definition (equivalent to the one already presented in Chapter 1.2) of its *GL reduct* w.r.t. an interpretation $I$ as the positive ground program $P^I$, obtained from $P$ by:

- deleting all rules having a negated literal which is not modeled by $I$;
- deleting all the negated literals from the remaining rules.

$I \subseteq HB_{\mathcal{U}}(P)$ is an answer set for a program $P$ w.r.t. $\mathcal{U}$ iff $I$ is a minimal model for the positive program $grnd_{\mathcal{U}}(P)^I$. Let $ans_{\mathcal{U}}(P)$ be the set of answer sets of $grnd_{\mathcal{U}}(P)$. We call $P$ ***F-satisfiable***, if it has some answer set for a fixed function mapping $F$, i.e. if there is some interpretation $\langle S, F \rangle$ which is an answer set. In the following we will assume the semantics associated to each external predicate is defined a priori, i.e. $F$ is fixed.

## 8.4 Properties of DLP-EX programs

Although simple in its definition, the above semantics does not give any hint on how to actually compute answer sets of a given program $P$. In general, given an infinite domain of constants $\mathcal{U}$, and a program $P$, $HB_{\mathcal{U}}(P)$ is indeed infinite.

**Theorem 8.2** It is given a DLP-EXprogram $P$, a domain of constants $\mathcal{U}$, and a function mapping $F$ where the co-domain of $F$ contains only boolean functions decidable in polynomial time in the size of their arguments. Deciding whether $P$ is $F$-satisfiable in the domain $\mathcal{U}$ is undecidable.

**Proof.** (Sketch) The proof is carried out by showing that the Answer Set Semantics of a ordinary program $\overline{P}$ with function symbols[3] can be reduced to the Answer Set Semantics of a DLP-EX program $P$. We take advantage of a family of external predicates $\{\#function_i\}$. In a given interpretation $\langle S, F\rangle$, $F$ will be such that $\#function_i(C, f, x_1, \ldots, x_i)$ is modeled if $C$ unifies with the compound term $f(x_1, \ldots, x_i)$.

This allows to rewrite a logic program $\overline{P}$ with function symbols by means of external predicates. For instance, the rule

$$p(s(X)) \text{:-} a(X, f(Y, h(Z))).$$

can be rewritten in an equivalent DLP-EX rule:

$$p(S) \text{:-} a(X, F), \#function_1(S, s, X), \#function_2(F, f, Y, H), \\ \#function_1(H, h, Z).$$

□

Tailoring cases where a finite portion of $\mathcal{U}$ is enough to evaluate the semantics of a given program is thus of interest. In the following we reformulate some results regarding splitting sets [Lifschitz and Turner, 1994].

**Definition 8.3** Given a DLP-EX program $P$, a *splitting set* is a set of atoms $A \in HB_{\mathcal{U}}(P)$ such that for each atom $a \in A$, if $a \in H(r)$ for some $r \in grnd_{\mathcal{U}}(P)$, then $B(r) \cup H(r) \subseteq A$. The *bottom* $b_A(P)$ is the set of rules $\{r \mid r \in grnd_{\mathcal{U}}(P)$ and $H(r) \subseteq A\}$. The *residual* $r_{\mathcal{U}}(P, I)$ is a program obtained from $grnd_{\mathcal{U}}(P)$ by deleting all the rules which are not modeled by $I$, and removing from the remaining rules all the $a \in A$ modeled by $I$.

We take advantage here of the formulation of the splitting theorem as given in [Bonatti, 2004].

**Theorem 8.4** *(Splitting theorem [Lifschitz and Turner, 1994; Bonatti, 2004])* Given a program $P$ and a splitting set $A$, $M \in ans_{\mathcal{U}}(P)$ iff $M$ can be split in two disjoint sets $I$ and $J$, such that $I \in ans_{\mathcal{U}}(b_A(P))$ and $J \in ans_{\mathcal{U}}(r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_A(P)), I)$.

**Definition 8.5** Given a rule $r$, a variable $X$ is *safe* in $r$ if it appears in some ordinary atom $a \in B^+(r)$. A rule $r$ is *safe* if each variable $X$ appearing in $r$ is safe. A program $P$ is *safe* if each rule $r \in P$ is safe.

[3] Positive Horn programs with function symbols are undecidable, see e.g. [Dantsin *et al.*, 2001]

**Theorem 8.6** Given a safe DLP-EX program $P$, let $U \subset \mathcal{U}$ be the set of constants appearing in $P$. Then $ans_U(P) = ans_{\mathcal{U}}(P)$.

**Proof.**(Sketch) The line of reasoning of the theorem is proving that, assuming $P$ is safe, $grnd_U(P)$ is a finite splitting set for $P$. Furthermore, $grnd_U(P) = b_U(P)$. For each $M \in ans_U(P)$, we can prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_U(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Viceversa, assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

In case a safe program is given, the above theorem allows to consider as the set of 'relevant' constants only those values explicitly appearing in the program at hand. Intuitively, the semantics of a safe program $P$ can be evaluated by means of the following steps:

- compute $grnd_U(P)$;
- remove from $grnd_U(P)$ all the rules containing at least one external literal $e$ such that $F \not\models e$, and remove from each rule all the remaining external literals.
- compute the remaining ordinary program by means of a standard DLP solver, such as DLV.

It is worth pointing out that, assuming the complexity of computing oracles is polynomial in the size of their arguments, this algorithm as same complexity as computing $grnd_U(P)$[4].

## 8.5 Dealing with values invention

Although important for clarifying the given semantics, it is an actual practice to specify external sources of computation not in terms of boolean oracles. So we aim at introducing the possibility to specify *functional oracles*, keeping anyway the simple reference semantics given previously. In the new setting we are going to introduce, it is also very important that an external atom brings knowledge from external sources of computation, in terms of new symbols added to a given program.

[4]Assuming rules can have unbounded length, grounding a disjunctive logic program is in the worst case exponential in the size of the Herbrand base (see e.g. [Leone *et al.*, 2001]).

For instance, assume $\mathcal{U}$ contains encoded values that can be interpreted as natural numbers and that the external predicate *#sqr* is defined such that the atom *#sqr(X,Y)* is true whenever $Y$ encodes a natural number representing the square of the natural number $X$; we want to extract a series of squared values from this predicate.

**Example 8.7** Consider now the following short program, containing an unsafe rule.

$number(2).$
$square(Y) \text{:-} number(X), \#sqr(X, Y).$

In the presence of unsafe rules as in the above example, Theorem 8.6 ceases to hold: it is indeed unclear whether there is a finite set of constants which the program can be grounded on. In the above example, we can intuitively conclude that the set of meaningful constants is $\{2, 4\}$. It is however undecidable, given a computable boolean oracle $f$ to establish whether a given set $S$ contains all and only all those tuples $\bar{t}$ such that $f(\bar{t}) = 1$.

In order to overcome these limits, we extend our framework with the possibility of explicitly computing missing values on demand. Although restrictive, this setting is not far from a realistic scenario where external predicates are defined by means of generic partial functions instead of boolean ones.

**Definition 8.8** It is given an external predicate name *#p*, having arity $n$ and its oracle function $F(\#p)$. A *pattern* is a list of $b$'s and $u$'s. A $b$ will represent a placeholder for a constant (or a bounded variable), whereas an $u$ will be a placeholder for a variable. Given a list of terms, the corresponding pattern will be given by replacing each constant with a $b$, and each variable with a $u$.

For instance, the pattern related to the list of terms $(X, a, Y)$ is $(u, b, u)$. Let $pat$ be a pattern of length $n$ having $k$ placeholders $b$ (which we will call input positions), and $n - k$ placeholders of $u$ type (which we will call output positions). A *functional oracle* $F(\#p)[pat]$ for the pattern $pat$, associated to the external predicate *#p*, is a partial function taking $k$ constant arguments from $\mathcal{U}$ and returning a tuple of arity $n - k$, and such that $F(\#p)[pat](a_1, ..., a_k) = b_1, ..., b_{n-k}$ iff $F(\#p)(a_1, \ldots, a_k, b_1, \ldots, b_{n-k}) = 1$. Let $pat[j]$ be the $j$-th element of a pattern $pat$. Let $unbound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = u$ for each $X_j \in \overline{X}$, and $bound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = b$ for each $X_j \in \overline{X}$.

An external predicate *#p* might be associated to one or more functional oracles 'consistent' with the originating boolean oracle. For instance, consider the *#sqr* external predicate, defined as mentioned above. We associate to it two functional oracles, $F(\#sqr)[b,u]$ and $F(\#sqr)[u,b]$. The two functional oracles are such that, e.g.

$$F(\#sqr)[b,u](3) = 9 \tag{8.2}$$
$$F(\#sqr)[u,b](16) = 4 \tag{8.3}$$

and this is consistent with the fact that $F(\#sqr)(3,9) = F(\#sqr)(4,16) = 1$, whereas $F(\#sqr)[u,b](5)$ is set as undefined since $F(\#sqr)(X,5) = 0$ for any natural $X$.

In the sequel, given an external predicate *#e*, we will assume it comes equipped with its oracle $F(\#e)$ (called also *base oracle*) and a list of consistent functional oracles $\{F(\#e)[pat_1], \ldots, F(\#e)[pat_m]\}$, having different patterns $pat_1, \ldots, pat_m$[5].

Adopting functional oracles in the context of safe programs is however to big a restriction. We thus aim at enlarging the class of programs that can be evaluated against a finite Herbrand universe. To this end, we introduce a relaxed notion of safety. Intuitively, a variable is weakly safe if its value, although not explicitly appearing in a program, can be computed through a functional oracle.

**Definition 8.9** Given a rule $r$, let $E(r)$ its set of external atoms. A *choice $C$ of functional oracles* is a mapping $C : E(r) \mapsto \mathbf{N}$ associating each external atom of $r$ with the index of one of its functional oracles. Given a choice $C$, let $F_C(\#e)$ a shortcut for the functional oracle $F(\#e)[pat_{C(\#e)}]$.

Given a rule $r$ and a choice $C$, a variable $X$ is *weakly safe in $r$ w.r.t. to $C$* if either

- $X$ is safe; or
- $X$ appears in some external atom $\#e(\overline{X}) \in B^+(r)$, $X \in unbound_{pat_{C(\#e)}}$ and each variable $Y \in bound_{pat_{C(\#e)}}$ is weakly safe.

A rule $r$ is weakly safe if there is a choice $C_r$ such that each variable $X$ appearing in some atom $a \in B(r)$ is weakly safe with respect to $C_r$. A program $P$ is weakly safe if each rule $r \in P$ is weakly safe.

[5]Note that functional oracles prevent, to some extent, to define multivalued functions and/or generic relations. We consider anyway this setting acceptable for a variety of applications.

**Example 8.10** Assume that *#sqr* is associated to the list of functional oracles $\{F(\#sqr)[b,u],\ F(\#sqr)[u,b]\}$ defined above. Given a choice of oracles $C$ such that $C(\#sqr(X,Y)) = 2$, the second rule of Program 8.7 is not weakly safe (intuitively there is no way for computing the value of the variable $Y$ with the oracle $F(\#sqr)[u,b]$. The same rule is weakly safe if we set $C(\#sqr(X,Y)) = 1$.

It turns out that deciding whether a given rule is weakly safe or not depends on a given choice, but also from the set of available functional oracles. It is assumed indeed that an external predicate does not come with all its possible functional oracles.

**Proposition 8.11** Given a set of external predicates $\mathcal{E}$, and a list of functional oracles for each $\#e \in \mathcal{E}$, it can be checked in polynomial time whether a program $P$ is weakly safe.

**Proof.**(Sketch) Simply observe that for each rule $r \in P$ it can be checked in time linear in the number of atoms of $r$ whether a choice making $r$ weakly safe exists. □

Weakly safe rules can be grounded with respect to functional oracles as follows.

**Definition 8.12** Given a weakly safe rule $r$, a choice $C$ for it, and a set of ordinary ground atoms $A$, a ground rule $r'$ is member of $ins(r, A)$ if $r$ can be grounded to $r'$ by the following algorithm:

1. replace positive literals of $r$ with a consistent nondeterministic choice of matching ground atoms from $A$; let $\theta$ the resulting variable substitution;
2. until $\theta$ instantiates all the variables of $r$:
   - pick from $r\theta$ an external atom $\#e(\overline{X})\theta$ such that $\theta$ instantiates all the variables $X \in bound_{pat_{C(\#e)}}(\overline{X})$.
   - If $F_C(\#e)(bound_{pat_{C(\#e)}}(\overline{X}\theta)) = a_1, \ldots, a_k$, then update $\theta$ by assigning $a_1, \ldots, a_k$ to $unbound_{pat_{C(\#e)}}(\overline{X}\theta)$; else fail;
3. return $r' = r\theta$.

**Example 8.13** Let's consider the second rule of Example 8.7; then, $ins(r, \{number(1), number(2)\})$ contains the two rules:

$square(1) \text{:-} number(1), \#sqr(1, 1).$
$square(4) \text{:-} number(2), \#sqr(2, 4).$

Although desirable, weak safety is not sufficient in order to intuitively guarantee finiteness of answer sets and decidability.

**Example 8.14** For instance, the program:

$square(2).$
$square(Y) \text{:-} square(X), \#sqr(X, Y).$

is modeled by the infinite set of atoms $\{square(2), square(4), \dots\}$.

We thus introduce the notion of *semi-safe* program. Intuitively a semi-safe program is such that external atoms cannot create infinite chains of new values to be taken in account.

**Definition 8.15** A weakly safe program $P$ is *semi-safe* if each cycle in $G(P)$ contains only edges corresponding to safe rules.

**Example 8.16** The program

$square(Y) \text{:-} square(X), number(Y), \#sqr(X, Y).$
$square(Y) \text{:-} number(X), \#sqr(X, Y).$

is semi-safe.

We extend next Theorem 8.6 to the case of semi-safe programs.

**Theorem 8.17** It is given a semi-safe program $P$. Then there is a finite set of constants $U$ such that $ans_U(P) = ans_{\mathcal{U}}(P)$.

**Proof.**(Sketch) The set $U$ is defined as all the constant symbols appearing in the set of atoms $T_P^\infty(\emptyset)$ where the operator $T_P$ is defined as follows.

$$T_P(A) = A \cup \{a \in H(r') | r' \in ins(r, A) \text{ for some } r \in P\}$$

It is provable that $T_P^\infty(\emptyset) = T_P^n(\emptyset)$ for some $n$ in case $P$ is semi-safe; $T_P^\infty(\emptyset)$ is a splitting set, and $U$ is finite; as in Theorem 8.6 for each $M \in ans_U(P)$, we can

prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_{T_P^\infty}(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

The above theorem allows to compute semantics of a semi-safe program $P$ by means of a traditional answer set solver, following the steps:

- compute the ground program $T_P^\infty(\emptyset)$. This computation involves a number of evaluation of $ins(r, A)$ that trigger evaluation of functional oracles whenever needed;
- eliminate external literals as in the case of safe programs;
- evaluate the remaining ordinary program by means of a traditional solver;

We observe that, assuming $F$ contains polynomial-time functional oracles, the complexity of the above algorithm is not greater than the complexity of computing grounding for an ordinary program.

## 8.6 Implementation and experiments

The proposed DLP-EX language has been integrated into the DLV system. The prototype is called DLV-EX[6], and it's introduced here, as well as some experiments.

From a practical point of view, the external atoms are dealt with in the following steps (see Figure 8.1):

1. at design time: a developer provides a library of external atoms, each of them associated with a set of functional oracles. Each functional oracle has a corresponding pattern. Although useful in practice, it is not compulsory to provide functional oracles other than the base oracle. However, the absence of specific functional oracles limits *de facto* the possibility to exploit an external atom in weakly safe rules. A testing environment helps checking the correctness of the oracles by means of automatically generated test programs.

[6] `http://www.mat.unical.it/kali/dlv-ex`

2. at run-time in a pre-processing stage: each rule is checked to be weakly safe, and a suitable choice of functional oracles is made. Then the overall program is checked to be semi-safe. It is anyway possible to relax this second condition, provided that termination of grounding algorithms is not guaranteed in this case. It is worth pointing out that an user developing a logic program is not in charge of specifying a choice of oracles, since the system itself will choose the best functional oracles among a variety of possibilities.

3. at run-time during the rule instantiation stage: the optimized grounder of the DLV system has been extended in order to compute $ins(r, A)$ for a given rule $r$ and a set of 'active' atoms $A$. For each external atom in $r$,the chosen functional oracles are repeatedly invoked according to Definition 8.12.

Point 2 and 3 above are integrated in the existing grounding algorithm of the system. We briefly recall the rule instantiation algorithm, part of the Intelligent Grounding module of the DLV system already cited in Chapter 4.3. Given a rule $r$, this algorithm exploits an intelligent backtracking algorithm, where a given atom $a \in B(r)$ is picked at each stage and it is tried to be instantiated with respect to currently allowed values. The picking order is crucial in order to tailor the search space to the smallest extent: in principle, it is preferred to pick first those atoms whose estimated set of possible values is smaller.

The presence of external atoms impacts within such algorithm in a two-fold way: for what point 2 above is concerned, given a rule $r$, among possible choices of functional oracles, our algorithm prefers those patterns whose number of unbounded variables is bigger. This intuitively allows to reduce the space of possible instantiations for a given external atom. For instance, given the atom *#sqr(X,Y)*, the choosing algorithm prefers, whenever possible, to choose the oracle with pattern $(b, u)$ instead of the base oracle (which can be seen has having the pattern $(b, b)$), since this way it is searched only the space of values where $Y$ is equal to the square of $X$. In the second case, an oracle with pattern $(b, b)$ forces in principle to check all the possible couples of values for $X$ and $Y$.

Point 3 impacts on the atom pick-up ordering strategy. For the same reasons above, it is preferred to pick up external atoms, with pattern having many unbounded variables, as earlier as possible. This strategy relies on the assumption, often true in practice, that the computation of a functional oracle is less time consuming than several computations of the corresponding base oracle.

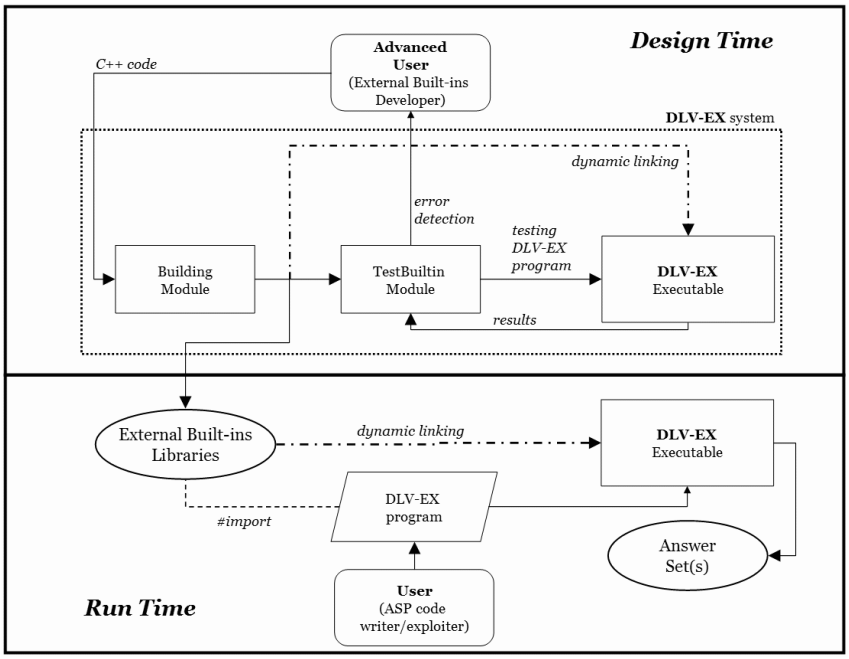


Figure 8.1: *System Architecture*

All the pre-existing built-in atoms available in the DLV system (such as arithmetic and relational operators) have been rewritten using the new general framework. We carried out some experiment in order to appreciate the impact and the possible overhead of the new construct. Results are encouraging: grounding times are in most cases equivalent, and the slowdown reported in few cases is never above 6-7%.

External predicate definitions can be grouped in one or more libraries. Libraries have to be compiled such that they can be dynamically linked to the DLV-EXexecutable; oracles are written in the C++ language. A special directive inside DLV-EXprograms tells the system which libraries have to be linked at runtime. Also, built-in developers are enabled to redefine predefined operators in order to deal with new data types, e.g. real numbers.

Some usage experiments have been carried out as well; few users have been requested to start implementing some customized libraries [Palopoli *et al.*, 2005; Cumbo *et al.*, 2004], and early feedbacks are positive both from the correctness and the ease of use points of view.

## 8.7 Related works

We give now a due overview on the related works in the literature.

For what the possibility of calling external modules in a logic program is concerned, it is worth to mention the foundational work of Eiter *et al.* [1997d]. This paper takes the notion of generalized quantifier, known in formal logics, and adapts it in the context of modular logic programming. A generalized quantifier indeed, can be seen as a way for delegating the truth value of a formula to an external source of computation.

Based on this work, the same authors are addressing the issue of implementing generalized quantifiers under Answer Set Semantics, in order to enable Answer Set Solvers to communicate, in both directions, with external reasoners [Eiter *et al.*, 2004b; 2005]. This approach is different from the one considered in this work since the former is inspired from second order logics and allows bidirectional flow of relational data (to and from an external atom), whereas, in our setting, the information flow is strictly value based. Nonetheless, HEX programs, as defined in [Eiter *et al.*, 2005], do not deal with infinite domains explicitly.

Although this know-how has not been explicitly divulgated yet, other Answer Set Solvers introduced the possibility to deal with externally computed functions [Syrjänen, 2002; Osorio and Corona, 2003].

Furthermore, there are several works aiming at bringing in Answer Set Programming a restricted capability of dealing with infinite domains. Among these, it is worth citing the notion of $\omega$-restricted programs [Syrjänen, 2001]. $\omega$-restricted programs allow to keep decidability of Answer Set Semantics in the presence of functions symbols, and constitute a subclass of finitary programs. It is indeed important to recall the work of Bonatti [2004], aimed at tailoring the class of finitary programs. Although, in general, recognizing this class of programs is undecidable, finitary programs allow function symbols but are decidable under brave/skeptical reasoning with ground queries. As shown in Theorem 8.2, external functions might be exploited in order to simulate function symbols. It is a matter of future search to extend the notion of semi-safe program to a larger class and investigate equivalence conditions with the notion of finitary program.

In the above cited literature, infinite domains are obtained through the introduction of compound functional terms. Thus the studied theoretical insights are often specialized to this notion of term, and take advantage e.g., of the common unification rules of formal logics over infinite domains. Similar in spirit to our approach is the work on open logic programs, and conceptual logic programs

[Heymans *et al.*, 2004]. Such paper addresses the possibility of grounding a logic program, under Answer Set Semantics, over an infinite domain, in a way similar to classical logics and/or description logics. Each constant symbol has no predefined compound structure however. Also similar is the work of Cabibbo [1998], which extend the work of Hull and Yoshikawa [1990]. The latter authors introduce a language (ILOG) with a special construct aimed at introducing new invented values in a logic program, for the purpose of creating new tuple identifiers in relational databases. Based on this work, Cabibbo investigates about decidable fragments of the language. Despite some crucial semantic differences, the presented notion of weak safety is similar to the one herein presented, and describes conditions such that new values do not propagate in infinite chains.

# Chapter 9

# Conclusions

In this thesis we have studied Disjunctive Logic Programming, a very powerful and expressive formalism which has recently became quite popular in AI in the areas of non-monotonic reasoning and logic programming. Our studies addressed some issues; indeed, practical applications in many emerging areas require always higher performances; moreover, there are some kinds of problems that cannot be encoded in a natural way and then the resulting programs are often complex and tricky.

We focused on these issues; thus, we proposed new techniques aiming at improving the efficiency of the DLV system (or similar DLP systems), and new extensions of Disjunctive Logic Programming for enhancing its knowledge modelling abilities.

In summary, the main contributions of the work are the following:

- [$PartI$] We have presented Disjunctive Logic Programming; we have first defined the syntax of this language and its associated semantics, the *Answer Set Semantics*. Then, we have analyzed the computation complexity of this language and we have illustrated the usage of Disjunctive Logic Programming for knowledge representation and reasoning.

- [$PartII$] We have addressed some key issues for the computation of disjunctive logic programs. In particular, we have focused on search space pruning, which is crucial for the efficiency of DLP systems. We have carried out an in-depth analysis of two main pruning operators for DLP, namely *Fitting*'s operator and the *Well-founded* operator. We have proposed a new strategy for the intelligent combination of the two pruning operators, and

we have implemented it in the DLV system. We have carried out experiments, which confirm the strong impact of the pruning operators on the efficiency of DLP systems, and assess the importance of our results. Interestingly, even if the Well-founded operator is computationally more expensive than Fitting's operator (quadratic time versus linear time in the propositional case), its stronger pruning power often pays off and reduces the computation time by an order of magnitude in some cases.

Future work can focus on tuning the actual implementations of the pruning operators, and on singling out new classes of DLP programs where they are efficiently computable.

- [$PartIII$] In the first piece of this part of the thesis we have started addressing some lacks of DLP, namely code reusability and modularity. We have presented the DLP$^T$ language, an extension of DLP allowing to define template predicates. The proposed language is very promising; the future work can have as objectives: introducing a clearer model theoretic semantic and prove its equivalence with the current operational semantics; generalizing template semantics in order to allow safe and meaningful forms of recursion between template definitions; introducing new forms of template atoms in order to improve reusability of the same template definition in different contexts; prove the formal equivalence of DLT sub-programs with semantics for aggregate constructs such as in [Calimeri *et al.*, 2005]; extending the template definition language using standard languages such as C++, such as in [Eiter *et al.*, 2005]; consider program equivalence results [Eiter *et al.*, 2004a] in order to optimize the size of unfolded programs.

  In the second piece of this last part of the thesis, continuing in addressing some lacks of DLP, namely the gaps between Disjunctive Logic Programming and practical applications, we have presented a framework where external atoms with value invention are taken in account. The purpose of this work is in the direction of starting to close those gaps. Also, we believe this works paves the way to an actual implementation of finitary programs with function symbols. The system prototype, examples, manuals and benchmark results are available at `http://www.mat.unical.it/kali/dlv-ex`.

# Bibliography

[Anger *et al.*, 2001] 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.

[Apt and Bol, 1994] K. Apt and N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[Apt *et al.*, 1988] 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.

[Aravindan *et al.*, 1997] 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.

[Babovich, since 2002] Yulia Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[Baral and Gelfond, 1994] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[Baral, 2002] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[Bell *et al.*, 1994] Colin Bell, Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *Journal of the ACM*, 41:1178–1215, 1994.

[Ben-Eliyahu and Dechter, 1994] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[Ben-Eliyahu and Palopoli, 1994] 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.

[Berman *et al.*, 1995] Kenneth A. Berman, John S. Schlipf, and John V. Franco. Computing the well-founded semantics faster. In *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'95)*, volume 928 of *Lecture Notes in AI (LNAI)*, pages 113–125. Springer, 1995.

[Bonatti, 2004] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.

[Brass and Dix, 1995] 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.

[Buccafurri *et al.*, 2000] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[Buccafurri *et al.*, 2002] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive Logic Programs with Inheritance. *Theory and Practice of Logic Programming*, 2(3), May 2002.

[Bugliesi *et al.*, 1994] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[Cabibbo, 1998] Luca Cabibbo. The Expressive Power of Stratified Logic Programs with Value Invention. *Information and Computation*, 147(1):22–56, 1998.

[Cadoli and Schaerf, 2001] Marco Cadoli and Andrea Schaerf. Compiling Problem Specifications into SAT. In *ESOP*, pages 387–401, 2001.

[Cadoli *et al.*, 1997] 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.

[Cadoli *et al.*, 1999] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An Executable Specification Language for Solving all Problems in NP. In Gopal Gupta, editor, *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 16–30. Springer, 1999.

[Calimeri *et al.*, 2005] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and Computational Properties of Logic Programs with Aggregates. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 406–411, August 2005.

[Calimeri *et al.*, since 2003] Francesco Calimeri, Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. The DLP$^T$ homepage, since 2003. `http://dlpt.gibbi.com/`.

[Calimeri, 2001] Francesco Calimeri. Progettazione e Sviluppo di Tecniche Di Ottimizzazione per Sistemi di Basi di Conoscenza. Master's thesis, D.E.I.S., Università degli Studi della Calabria, Rende (CS), Italy, 2001. Supported by Nicola Leone.

[Chen and Warren, 1996] 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.

[Cholewiński *et al.*, 1996] 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.

[Cholewiński *et al.*, 1999] Paweł Cholewiński, Victor W. Marek, Artur Mikitiuk, and Mirosław Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.

[Cumbo *et al.*, 2004] Chiara Cumbo, Salvatore Iiritano, and Pasquale Rullo. Reasoning-based knowledge extraction for text classification. In *Discovery Science*, pages 380–387, 2004.

[Dantsin *et al.*, 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[Dell'Armi *et al.*, 2003] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, August 2003. Morgan Kaufmann Publishers.

[Dix, 1995] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[Dowling and Gallier, 1984] 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.

[East and Truszczyński, 2000] Deborah East and Mirosław Truszczyński. dcs: An Implementation of DATALOG with Constraints. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[East and Truszczyński, 2001a] D. East and M. Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. Springer Verlag, LNAI 2174, 2001.

[East and Truszczyński, 2001b] Deborah East and Mirosław Truszczyński. System Description: aspps – An Implementation of Answer-Set Programming with Propositional Schemata. 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 402–405. Springer Verlag, September 2001.

[Egly *et al.*, 2000] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 417–422. AAAI Press / MIT Press, 2000.

[Eiter and Gottlob, 1995] 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.

[Eiter *et al.*, 1997a] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[Eiter *et al.*, 1997b] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from Logic Programs: Semantics and Complexity. In *Theoretical Computer Science* [1997a], pages 129–177.

[Eiter *et al.*, 1997c] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[Eiter *et al.*, 1997d] Thomas Eiter, Georg Gottlob, and Helmuth Veith. Modular Logic Programming and Generalized Quantifiers. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)*, number 1265 in LNCS, pages 290–309. Springer, 1997.

[Eiter *et al.*, 1997e] Thomas Eiter, Nicola Leone, and Domenico Saccà. On the Partial Semantics for Disjunctive Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(1–2):59–96, April 1997.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, and Domenico Saccá. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theoretical Computer Science*, 206(1–2):181–218, October 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and F. Scarcello. The

DLV System. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[Eiter *et al.*, 2000] 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.

[Eiter *et al.*, 2003a] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. *Theory and Practice of Logic Programming*, 3:463–498, July/September 2003.

[Eiter *et al.*, 2003b] 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.

[Eiter *et al.*, 2004a] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, number 2923 in Lecture Notes in AI (LNAI), pages 87–99, Fort Lauderdale, Florida, USA, January 2004. Springer.

[Eiter *et al.*, 2004b] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Nonmonotonic description logic programs: Implementation and experiments. In *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004*, pages 511–527, 2004.

[Eiter *et al.*, 2005] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, pages 90–96, Edinburgh, UK, August 2005.

[Erdem, 1999] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[Faber and Pfeifer, since 1996] Wolfgang Faber and Gerald Pfeifer. `DLV` homepage, since 1996. `http://www.dlvsystem.com/`.

[Faber *et al.*, 1999] 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.

[Faber *et al.*, 2001] Wolfgang Faber, Nicola Leone, and Gerald 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, August 2001. Morgan Kaufmann Publishers.

[Faber, 2002] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[Fages, 1994] François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[Fitting, 1985] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Franconi *et al.*, 2001] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 561–578. Springer, 2001.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[Gelfond and Leone, 2002] M. Gelfond and N. Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1-2):3–38, 2002.

[Gelfond and Lifschitz, 1988] 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.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a Source of Expression Complexity. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.

[Gottlob, 1994] Georg Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.

[Heymans *et al.*, 2004] Stijn Heymans, Davy Van Nieuwenborgh, and Dirk Vermeir. Semantic web reasoning with conceptual logic programs. In *Rules and Rule Markup Languages for the Semantic Web: Third International Workshop, RuleML 2004, Hiroshima, Japan, November 8, 2004.*, pages 113–127, 2004.

[Hull and Yoshikawa, 1990] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 455–468. Morgan Kaufmann, 1990.

[Ianni *et al.*, 2003] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, and Maria Carmela Santoro. Answer Set Programming with Templates. In Marina de Vos and Alessandro Provetti, editors, *Proceedings ASP03 - Answer Set Programming: Advances in Theory and Implementation*, pages 239–252, Messina, Italy, September 2003. Online at `http://CEUR-WS.org/Vol-78/`.

[Janhunen *et al.*, 2005] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 2005. To appear.

[Johnson, 1990] David S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Pub., 1990.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, 1992.

[Koch and Leone, 1999] 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.

[Kuper, 1990] G. M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, 1990.

[Laenens *et al.*, 1990] Els Laenens, Domenico Saccà, and Dirk Vermeir. Extending Logic Programming. In *SIGMOD Conference*, pages 184–193, 1990.

[Leone and Rullo, 1993] Nicola Leone and Pasquale Rullo. Ordered Logic Programming with Sets. *Journal of Logic and Computation*, 3(6), December 1993.

[Leone *et al.*, 1997] 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.

[Leone *et al.*, 2001] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP Instantiators by Join-Ordering Methods. 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). Springer Verlag, September 2001.

[Leone *et al.*, 2005] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2005. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

[Lierler and Maratea, 2004] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, LNCS, pages 346–350. Springer, January 2004.

[Lierler, 2005] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[Lifschitz and Turner, 1994] 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.

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lifschitz, 1999] 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.

[Lin and Zhao, 2004] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[Lobo *et al.*, 1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability Planning with Causal Theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[Minker, 1982] Jack Minker. On Indefinite Data Bases and the Closed World Assumption. In D.W. Loveland, editor, *Proceedings $6^{th}$ Conference on Automated Deduction (CADE '82)*, number 138 in Lecture Notes in Computer Science, pages 292–308, New York, 1982. Springer.

[Minker, 1994] Jack Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[Minker, 1996] J. Minker. Logic and Databases: a 20 Year Retrospective. In *Proceedings International Workshop on Logic in Databases (LID '96)*, number 1154 in LNCS, pages 3–57. Springer, 1996.

[Minoux, 1988] Michel Minoux. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29:1–12, 1988.

[Niemelä and Simons, 1996] 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.

[Niemelä and Simons, 1997] 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 Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[Nogueira *et al.*, 2001] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog Decision Support System for the Space Shuttle. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*, number 1990 in Lecture Notes in Computer Science, pages 169–183. Springer, 2001.

[Osorio and Corona, 2003] Mauricio Osorio and Enrique Corona. The A-Pol system. In *Answer Set Programming*, 2003.

[Palopoli *et al.*, 2005] L. Palopoli, S. Rombo, and G. Terracina. Flexible Pattern Discovery with (Extended) Disjunctive Logic Programming. In *International*

*Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, number 3448 in Lecture Notes in AI (LNAI), pages 504–513, Saratoga Springs, New York, USA, 2005. Springer-Verlag.

[Papadimitriou, 1994] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Perri, 2004] Simona Perri. *Disjunctive Logic Programming: Efficient Evaluation and Language Extensions*. PhD thesis, University of Calabria, 2004.

[Przymusinski, 1988] Theodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[Przymusinski, 1990] T. Przymusinski. Stationary Semantics for Disjunctive Logic Programs and Deductive Databases. In *Proceedings of North American Conference on Logic Programming*, pages 40–62, 1990.

[Przymusinski, 1991] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[Przymusinski, 1995] T. Przymusinski. Static Semantics for Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.

[Radziszowski, 1994] Stanislaw P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.

[Rao *et al.*, 1997] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[Ross and Sagiv, 1997] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *Journal of Computer and System Sciences*, 54(1):79–97, February 1997.

[Ross, 1990] K.A. Ross. The Well-Founded Semantics for Disjunctive Logic Programs. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 385–402. Elsevier Science Publishers B. V., 1990.

[Sakama, 1989] C. Sakama. Possible Model Semantics for Disjunctive Databases. In *Proceedings First Intl. Conf. on Deductive and Object-Oriented Databases (DOOD-89)*, pages 369–383, Kyoto, Japan, 1989. North-Holland.

[Seipel and Thöne, 1994] 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.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[Simons, 2000] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[Subrahmanian *et al.*, 1995] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.

[Syrjänen, 2001] Tommi Syrjänen. Omega-restricted logic programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.

[Syrjänen, 2002] Tommi Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[Tarjan, 1972] R. Tarjan. Depth-First Search and Linear Graph Algorithm. *SIAM Journal on Computing*, 1(2), June 1972.

[Van Gelder *et al.*, 1991] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Wolfinger, 1994] B. Wolfinger, editor. Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Berlin, August 1994. German Society for

Computer Science (GI), Springer. $13^{th}$ IFIP World Computer Congress, Hamburg, Germany.

# Appendix A

# Further Details on Experiments

## A.1 Encodings of the Problems

Note that some of the encodings reported here employ true negation (denoted using "$-$" in front of atoms), which has not been introduced in the syntax definition in this work. However, DLV reduces such programs to equivalent ones without true negations, substituting each truly negated occurrence of an atom $a$ by a new atom $na$ and adding a constraint $:- a, na.$ for each atom $a$ that occurs both with and without true negation.

### A.1.1 Hamiltonian Path

The actual encoding for HAMPATH used for experiment is almost the same as the one in Chapter 3.2.2. Suppose that the graph $G$ is specified by two predicates $node(X)$ and $arc(X, Y)$, and the starting node is specified by the predicate $start$ which contains only a single tuple. Then, the following program solves the problem HAMPATH:

% Each node has to be reached.
$reached(X) :- start(X).$
$reached(X) :- inPath(Y, X).$
$:- node(X), \text{not } reached(X).$
% Guess whether a given arc is in the path or not.
$inPath(X, Y) \vee outPath(X, Y) :- reached(X), arc(X, Y).$
% At most one incoming/outgoing arc!
$:- inPath(X, Y), inPath(X, Y1), Y <> Y1.$
$:- inPath(X, Y), inPath(X1, Y), X <> X1.$

### A.1.2 Blocksworld

For blocksworld, we used an encoding of the problem domain which is derived from an encoding in an action language. The reader can refer to [Erdem, 1999; Faber *et al.*, 1999] for further details.

```
% specification of the move action
move(B,L,T) v -move(B,L,T) :- block(B), location(L),
    actiontime(T), B <> L.
% the effects of moving a block
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
-on(B,L,T1) :- move(B,_,T), on(B,L,T), #succ(T,T1).
% move preconditions
% a block can be moved only when it's clear
:- move(B,L,T), on(B1,B,T).
% if a block is moved onto another block, the latter must be clear
:- move(B,B1,T), on(B2,B1,T), block(B1).
% concurrent actions are not allowed
:- move(B,_,T), move(B1,_,T), B <> B1.
:- move(_,L,T), move(_,L1,T), L <> L1.
% inertia
on(B,L,T1) :- on(B,L,T), not -on(B,L,T1), #succ(T,T1).
% time at which actions can be initiated
actiontime(T) :- T < #maxint, #int(T).
% location definition (blocks are defined in the problem instances)
location(t).    location(B) :- block(B).
```

### A.1.3 Sokoban

The encoding solving SOKO puzzles follows.

```
% Timesteps etc.
time(T) :- #int(T).    actiontime(T) :- #int(T), T! = #maxint.
% define left and bottom for simplicity
left(L1,L2) :- right(L2,L1).    bot(L1,L2) :- top(L2,L1).
% define the adjacent squares
adj(L1,L2) :- right(L1,L2).    adj(L1,L2) :- left(L1,L2).
adj(L1,L2) :- top(L1,L2).    adj(L1,L2) :- bot(L1,L2).
% all the locations
location(L) :- adj(L,_).
```

% It is possible to push a box if the Sokoban can move to the square
% in front the box and the box can be pushed in the desired direction.
$push(B, right, B1, T)$ v $-push(B, right, B1, T)$ :–
$reachable(L, T), right(L, B), box(B, T), pushable\_right(B, B1, T)$,
$good\_pushlocation(B1), actiontime(T)$.
$push(B, left, B1, T)$ v $-push(B, left, B1, T)$ :–
$reachable(L, T), left(L, B), box(B, T), pushable\_left(B, B1, T)$,
$good\_pushlocation(B1), actiontime(T)$.
$push(B, up, B1, T)$ v $-push(B, up, B1, T)$ :–
$reachable(L, T), top(L, B), box(B, T), pushable\_top(B, B1, T)$,
$good\_pushlocation(B1), actiontime(T)$.
$push(B, down, B1, T)$ v $-push(B, down, B1, T)$ :–
$reachable(L, T), bot(L, B), box(B, T), pushable\_bot(B, B1, T)$,
$good\_pushlocation(B1), actiontime(T)$.

% reachable represents the locations which are reachable at some
% timestep from the location of the Sokoban in that timestep.
$reachable(L, T)$ :– $sokoban(L, T)$.
$reachable(L, T)$ :– $reachable(L1, T), adj(L1, L), notbox(L, T)$.

% The following rules define the possible pushes during some timestep.
$pushable\_right(B, D, T)$ :– $box(B, T), right(B, D), notbox(D, T)$,
$actiontime(T)$.
$pushable\_right(B, D, T)$ :– $pushable\_right(B, D1, T), right(D1, D)$,
$notbox(D, T)$.
$pushable\_left(B, D, T)$ :– $box(B, T), left(B, D), notbox(D, T), actiontime(T)$.
$pushable\_left(B, D, T)$ :– $pushable\_left(B, D1, T), left(D1, D), notbox(D, T)$.
$pushable\_top(B, D, T)$ :– $box(B, T), top(B, D), notbox(D, T), actiontime(T)$.
$pushable\_top(B, D, T)$ :– $pushable\_top(B, D1, T), top(D1, D), notbox(D, T)$.
$pushable\_bot(B, D, T)$ :– $box(B, T), bot(B, D), notbox(D, T), actiontime(T)$.
$pushable\_bot(B, D, T)$ :– $pushable\_bot(B, D1, T), bot(D1, D), notbox(D, T)$.

% Effects of pushing.
$sokoban(L, T1)$ :– $push(\_, right, B1, T), \#succ(T, T1), right(L, B1)$.
$sokoban(L, T1)$ :– $push(\_, left, B1, T), \#succ(T, T1), left(L, B1)$.
$sokoban(L, T1)$ :– $push(\_, up, B1, T), \#succ(T, T1), top(L, B1)$.
$sokoban(L, T1)$ :– $push(\_, down, B1, T), \#succ(T, T1), bot(L, B1)$.
$-sokoban(L, T1)$ :– $push(\_, \_, \_, T), \#succ(T, T1), sokoban(L, T)$.
$box(B, T1)$ :– $push(\_, \_, B, T), \#succ(T, T1)$.
$-box(B, T1)$ :– $push(B, \_, \_, T), \#succ(T, T1)$.

% Inertia. Unless changes are caused, things remain as they were.
$box(LB, T1) \text{:-}\ box(LB, T), \#succ(T, T1), not - box(LB, T1).$
$sokoban(LS, T) \text{:-}\ sokoban(LS, T), \#succ(T, T1), not - sokoban(LS, T1).$
% Unique actions per timestep.
$\text{:-}\ push(B, \_, \_, T), push(B1, \_, \_, T), B! = B1.$
$\text{:-}\ push(B, D, \_, T), push(B, D1, \_, T), D! = D1.$
$\text{:-}\ push(B, D, B1, T), push(B, D, B11, T), B1! = B11.$
% Auxiliary definitions.
$good\_pushlocation(L) \text{:-}\ right(L, \_), left(L, \_).$
$good\_pushlocation(L) \text{:-}\ top(L, \_), bot(L, \_).$
$good\_pushlocation(L) \text{:-}\ solution(L).$

## A.2 Sokoban Detailed Results

In Tables A.1 and A.2 you find detailed results for the Sokoban puzzle benchmarks. The reported numbers are seconds for runtime (user + system time).

| | **Old** | **ifPoss** | **ifNeed** |
|---|---|---|---|
| 1 | 738.04 | 21.39 | 19.70 |
| 2 | - | - | - |
| 3 | - | 7.48 | 6.99 |
| 4 | - | 7.74 | 7.29 |
| 5 | - | 77.59 | 71.42 |
| 6 | - | 173.68 | 162.39 |
| 7 | - | 195.27 | 178.51 |
| 8 | 548.24 | 12.27 | 11.62 |
| 9 | 264.85 | 16.04 | 14.97 |
| 10 | - | 16.01 | 14.77 |
| 11 | - | - | - |
| 12 | - | 54.76 | 50.20 |
| 13 | - | 23.90 | 22.18 |
| 14 | - | 271.24 | 242.87 |
| 15 | - | 19.25 | 17.54 |
| 16 | - | 627.05 | 575.95 |
| 17 | - | 20.13 | 18.58 |
| 18 | - | 45.31 | 41.43 |
| 19 | - | 487.70 | 441.67 |
| 20 | - | 32.13 | 30.09 |
| 21 | - | 99.02 | 92.08 |
| 22 | - | - | - |
| 23 | 255.83 | 21.41 | 20.10 |
| 24 | - | 13.81 | 12.79 |
| 25 | - | 31.69 | 29.24 |
| 26 | 321.81 | 20.69 | 19.26 |

| | **Old** | **ifPoss** | **ifNeed** |
|---|---|---|---|
| 27 | - | 7.54 | 7.13 |
| 28 | - | 199.58 | 183.73 |
| 29 | 421.22 | 4.07 | 3.86 |
| 30 | - | - | - |
| 31 | 124.32 | 14.42 | 13.23 |
| 32 | - | - | - |
| 33 | - | 219.63 | 202.54 |
| 34 | - | 62.37 | 57.57 |
| 35 | - | 199.52 | 182.09 |
| 36 | - | - | - |
| 37 | 549.44 | 17.63 | 16.68 |
| 38 | - | 56.75 | 51.69 |
| 39 | - | - | - |
| 40 | - | 16.62 | 15.07 |
| 41 | - | - | - |
| 42 | - | - | - |
| 43 | - | 27.04 | 25.15 |
| 44 | - | 295.75 | 276.50 |
| 45 | - | - | - |
| 46 | - | - | - |
| 47 | - | 9.44 | 8.76 |
| 48 | - | - | - |
| 49 | 385.71 | 13.89 | 12.90 |
| 50 | - | 25.35 | 23.18 |
| 51 | - | 217.76 | 203.39 |
| 52 | - | 246.47 | 230.39 |

Table A.1: Detailed results for the Yoshio Murase SOKO instances.

| | **Old** | **ifPoss** | **ifNeed** |
|---|---|---|---|
| 1 | 0.98 | 0.83 | 0.82 |
| 2 | 1.90 | 1.67 | 1.60 |
| 3 | 0.65 | 0.68 | 0.67 |
| 4 | 2.99 | 1.60 | 1.56 |
| 5 | 61.65 | 2.22 | 2.16 |
| 6 | 152.54 | 21.19 | 19.28 |
| 7 | 3.86 | 1.67 | 1.60 |
| 8 | 100.86 | 3.87 | 3.73 |
| 9 | 1.95 | 1.43 | 1.37 |
| 10 | 1.16 | 0.90 | 0.90 |
| 11 | 3.78 | 1.59 | 1.55 |
| 12 | 83.54 | 2.27 | 2.20 |
| 13 | 27.30 | 5.05 | 4.75 |
| 14 | 445.60 | 4.86 | 4.65 |
| 15 | 48.18 | 2.92 | 2.74 |
| 16 | 0.59 | 0.60 | 0.60 |
| 17 | 53.47 | 9.74 | 9.10 |
| 18 | 361.54 | 5.98 | 5.58 |
| 19 | 8.21 | 2.14 | 2.06 |
| 20 | - | 8.72 | 8.07 |
| 21 | 1.55 | 1.01 | 0.99 |
| 22 | 565.06 | 7.54 | 7.01 |
| 23 | 44.30 | 1.72 | 1.68 |
| 24 | - | 27.28 | 25.18 |
| 25 | 82.02 | 11.41 | 10.57 |
| 26 | 125.81 | 7.29 | 6.76 |
| 27 | 3.87 | 1.55 | 1.50 |
| 28 | - | 13.63 | 12.52 |
| 29 | - | 23.90 | 22.46 |
| 30 | 13.81 | 2.22 | 2.18 |
| 31 | 37.72 | 1.92 | 1.87 |
| 32 | 3.86 | 1.21 | 1.19 |
| 33 | - | 39.44 | 35.61 |
| 34 | - | 63.17 | 58.07 |
| 35 | 189.21 | 10.37 | 9.44 |
| 36 | - | 302.66 | 281.64 |

| | **Old** | **ifPoss** | **ifNeed** |
|---|---|---|---|
| 37 | - | 589.14 | 533.16 |
| 38 | 789.52 | 81.20 | 75.19 |
| 39 | 132.97 | 2.59 | 2.52 |
| 40 | - | 389.44 | 354.95 |
| 41 | 8.37 | 1.71 | 1.70 |
| 42 | - | 104.60 | 96.22 |
| 43 | 403.12 | 2.51 | 2.42 |
| 44 | 224.12 | 3.61 | 3.40 |
| 45 | - | 31.23 | 28.88 |
| 46 | - | 6.43 | 6.10 |
| 47 | - | 8.67 | 8.20 |
| 48 | - | 221.16 | 201.23 |
| 49 | - | 33.38 | 30.79 |
| 50 | - | - | - |
| 51 | - | - | - |
| 52 | - | 349.29 | 328.23 |
| 53 | - | 10.98 | 10.26 |
| 54 | - | - | - |
| 55 | 37.24 | 2.77 | 2.62 |
| 56 | - | 10.87 | 10.04 |
| 57 | - | 79.42 | 71.75 |
| 58 | - | 64.82 | 58.67 |
| 59 | - | 8.31 | 7.73 |
| 60 | - | 5.62 | 5.29 |
| 61 | - | 260.75 | 241.98 |
| 62 | - | 116.11 | 106.23 |
| 63 | - | 688.73 | 638.46 |
| 64 | 257.59 | 21.11 | 19.60 |
| 65 | - | 375.09 | 341.36 |
| 66 | - | 40.46 | 37.40 |
| 67 | - | 695.20 | 629.97 |
| 68 | - | 589.31 | 533.12 |
| 69 | - | 233.59 | 214.51 |
| 70 | - | 30.53 | 28.05 |
| 71 | - | 14.55 | 13.34 |

Table A.2: Detailed results for the Jacques Duthen SOKO instances.