I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG WISSENSBASIERTE SYSTEME

THE DLV SYSTEM FOR KNOWLEDGE REPRESENTATION AND REASONING

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Christoph Koch, Cristinel Mateis, Simona Perri, Francesco Scarcello

INFSYS RESEARCH REPORT 1843-02-14

OCTOBER 2002

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT

INFSYS RESEARCH REPORT 1843-02-14, OCTOBER 2002

THE DLV SYSTEM FOR KNOWLEDGE REPRESENTATION AND REASONING

Nicola Leone[1], Gerald Pfeifer[2], Wolfgang Faber[3], Thomas Eiter[3], Georg Gottlob[2], Christoph Koch[2], Cristinel Mateis[1], Simona Perri[1], Francesco Scarcello[4]

Abstract. Disjunctive Logic Programming (DLP) is an advanced formalism for knowledge representation and reasoning, which is very expressive in a precise mathematical sense: it allows to express *every* property of finite structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$). Thus, under widely believed assumptions, DLP is strictly more expressive than *normal* (*disjunction-free*) logic programming, whose expressiveness is limited to properties decidable in NP. Importantly, apart from enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion.
This paper presents the DLV system, which is widely considered the state-of-the-art implementation of disjunctive logic programming, and addresses several aspects. As for problem solving, we provide a formal definition of its kernel language, function-free disjunctive logic programs (also known as disjunctive datalog), extended by weak constraints, which are a powerful tool to express optimization problems. We then illustrate the usage of DLV as a tool for knowledge representation and reasoning, describing a new declarative programming methodology which allows one to encode complex problems (up to Δ_3^P-complete problems) in a declarative fashion. On the foundational side, we provide a detailed analysis of the computational complexity of the language of DLV, and by deriving new complexity results we chart a complete picture of the complexity of this language and important fragments thereof.
Furthermore, we illustrate the general architecture of the DLV system which has been influenced by these results. As for applications, we overview application front-ends which have been developed on top of DLV to solve specific knowledge representation tasks, and we briefly describe the main international projects investigating the potential of the system for industrial exploitation. Finally, we report about thorough experimentation and benchmarking, which has been carried out to assess the efficiency of the system. The experimental results confirm the solidity of DLV and highlight its potential for emerging application areas like knowledge management and information integration.

Keywords: answer sets, computational complexity, implementation, knowledge representation, non-monotonic reasoning, stable models

---

[1]Dept. of Mathematics, University of Calabria, 87030 Rende (CS), Italy. E-mail: {leone, mateis, perri}@unical.it.
[2]Institut of Information Systems, Database and AI Group, TU Vienna, Favoritenstraße 9-11, A-1040 Vienna, Austria. E-mail: {pfeifer, gottlob, koch}@dbai.tuwien.ac.at.
[3]Institut of Information Systems, Knowledge Bases Systems Group, TU Vienna, Favoritenstraße 9-11, A-1040 Vienna, Austria. E-mail: {eiter, faber}@kr.tuwien.ac.at.
[4]DEIS Department, University of Calabria, 87030 Rende (CS), Italy. E-mail: scarcello@deis.unical.it

Acknowledgements: This work was supported by the European Commission under projects IST-2002-33570 INFOMIX, FET-2001-37004 WASP and by FWF (Austrian Science Funds) under the projects Z29-INF and P14781.

Copyright © 2002 by the authors

Contents

1 Introduction **1**

2 Core Language **3**
2.1 Syntax . . . 3
2.2 Semantics . . . 4

3 Knowledge Representation in DLV **7**
3.1 Deductive Database Applications . . . 7
3.1.1 Reachability . . . 7
3.1.2 Same Generation . . . 7
3.2 The **GCO** Declarative Programming Methodology . . . 8
3.3 Applications of the **GCO** Programming Technique . . . 9
3.3.1 Exams Scheduling . . . 9
3.3.2 Hamiltonian Path . . . 10
3.3.3 Traveling Salesperson . . . 11
3.3.4 Winners in Combinatorial Auctions . . . 12
3.3.5 Ramsey Numbers . . . 13
3.3.6 Strategic Companies . . . 14
3.3.7 Preferred Strategic Companies . . . 16

4 The Complexity of the DLV **Language** **17**
4.1 A Reminder of the Polynomial Hierarchy . . . 17
4.2 Relevant Fragments of the DLV Language . . . 17
4.3 Main Problems Considered . . . 18
4.4 Derivation of Complexity Results . . . 19
4.5 Summary of Results and Discussion . . . 23

5 DLV **Front-Ends** **26**
5.1 Inheritance Front-End . . . 26
5.2 Diagnosis Front-End . . . 27
5.3 Planning Front-End . . . 28
5.4 SQL3 Front-End . . . 29
5.5 Meta-Interpreter Front-End . . . 30
5.6 External Front-Ends . . . 30

6 The Implementation of the DLV **System: An Overview** **31**

7 Experiments and Benchmarks **34**
7.1 Overview of Compared Systems . . . 34
7.2 Benchmark Problems and Data . . . 36
7.3 Results and Discussion . . . 43

8 Ongoing Projects **51**

9 Conclusion **52**

A Sokoban Encodings **59**

1 Introduction

The need for representing and manipulating complex knowledge arising in Artificial Intelligence and in other emerging areas, like Knowledge Management and Information Integration (see also Section 8), has renewed the interest in advanced logic-based formalisms for Knowledge Representation and Reasoning (KR&R), which started in the early 1980s. Among them, Disjunctive Logic Programming (DLP), which has first been considered by Minker [76] in the deductive database context, is one of the most expressive KR&R formalisms.

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [7, 73, 100, 32, 55, 70, 77, 6]. One of the attractions of disjunctive logic programming (DLP) is its capability of allowing the natural modeling of incomplete knowledge [7, 73]. Much research has been spent on the semantics of disjunctive logic programs, and several alternative semantics have been proposed [10, 47, 55, 76, 85, 88, 86, 92, 93] (see [2, 24, 73, 77, 78] for comprehensive surveys). The most widely accepted semantics is the *answer sets semantics* proposed by Gelfond and Lifschitz [55] as an extension of the stable model semantics of normal logic programs [54]. 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 programs under answer sets semantics are very expressive. It was shown in [45, 59] that, under this semantics, disjunctive logic programs capture the complexity class Σ_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.* [45] 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. The high expressiveness of disjunctive logic programming comes at the price of a higher computational cost in the worst case: The typical reasoning tasks, brave reasoning and cautious reasoning, on disjunctive logic programs are complete problems for the complexity class Σ_2^P and Π_2^P, respectively [45, 43], while non-disjunctive logic programs are complete for NP and co-NP, respectively (cf. [74, 22]).

The hardness of the evaluation of DLP programs has discouraged the implementation of DLP engines for quite some time. Only in the 1990s, more systematic implementation efforts have been launched, including [53, 73, 95, 25], but the systems created did not reach a level of maturity such that they could be used beyond toy examples.

A first solid, efficiency-geared implementation of a DLP system, called DLV, became available only in 1997, after 15 years of theoretical research on DLP. The development of DLV started at the end of 1996, in a research project funded by the Austrian Science Funds (FWF) and led by Nicola Leone at the Vienna University of Technology; at present, DLV is the subject of an international cooperation between the University of Calabria and the Vienna University of Technology, and its extension and application is supported also by the European Commission through IST projects (see Section 8).

After its first release, the DLV system has been significantly improved over and over in the last years, and its language has been enriched in several ways (see, e.g., [14, 13]). Relevant optimization techniques have been incorporated in all modules of the DLV engine, including database techniques for efficient instantiation [50, 68], novel techniques for answer-set checking [65], heuristics and advanced pruning operators for model generation [51, 17].A s a result, at the time being, DLV is generally recognized to be the state-of-the-art implementation of disjunctive logic programming, it 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 the area of Answer Set Programming (ASP).[1]

This paper focuses on the DLV system, one of the most successful and widely used DLP engines, and provides an in-depth description of several important aspects of the system; it is the first such article, since previous papers focused only on particular issues of the system. The main contributions of this paper are the following.

1. We provide a formal definition of the kernel language of DLV: disjunctive datalog (i.e., function-free disjunctive logic programs) under the consistent answer sets semantics, extended by weak constraints. The concept of weak constraints presented here generalizes previous work on weak constraints in [14], as it allows us to specify both weights and priorities over weak constraints.

2. We illustrate the usage of DLV as a tool for knowledge representation and reasoning. We describe a declarative "Guess/Check/Optimize" (**GCO**) programming methodology which allows us to encode complex queries and, more generally, search problems in a simple and highly declarative fashion in the DLV language; even some optimization problems of rather high computational complexity (up to Δ_3^P) can be declaratively encoded by using this methodology. We illustrate the usage of the **GCO** methodology on a large number of computationally hard problems taken from various application domains.

3. We analyze the computational complexity of the language of DLV. The analysis pays attention to the impact of syntactic restrictions on the complexity of the reasoning tasks. We derive new complexity results, which allow for providing a full picture of the complexity of the relevant fragments of the DLV language, and we identify syntactic subclasses which have lower complexity than the whole class. The complexity analysis is in fact the theoretical basis on which the DLV implementation has been founded, and its exploitation is one of the key factors of DLV's efficiency.

4. We overview application front-ends which have been developed on top of DLV to solve specific KR tasks, and we briefly describe the main international projects aimed at investigating the potential of DLV for industrial exploitation in the areas of Information Integration and Knowledge Management.

5. We illustrate the general architecture of DLV, explaining the inspiring principles underlying the DLV engine, and we overview the main techniques employed in DLV's implementation, describing how simpler language-fragments are recognized and evaluated by tailored algorithms, exploiting the information coming from the complexity analysis.

6. We perform a thorough experimentation activity. The main goal of the experiments is to assess the efficiency of DLV among the DLP systems. To this end, we compare DLV with another robust DLP system, named GnT [62]. We conduct some experiments also in the more general context of Answer Set Programming, comparing DLV and GnT against non-disjunctive ASP systems (such experiments are performed on a smaller set of benchmark problems, since non-disjunctive systems are more limited in power, and they not able solve Σ_2^P-hard problems). In particular, we compare the performance of DLV with two state-of-the-art ASP systems, namely, Smodels [97, 80, 79] and ASSAT [72].

[1] The term "Answer Set Programming" was coined by Vladimir Lifschitz in his invited talk at ICLP'99, to denote a declarative programming methodology, similar to SAT-based programming, but using more expressive logic programming languages with disjunction or nonmonotonic negation to encode the problems.

The entire experimentation activity is conducted on an ample set of benchmarks, covering different application areas, and comprising problems of very different complexities. We provide an in-depth discussion on the benchmark results, highlighting the major strengths and weaknesses of the compared systems, and identifying the best-suited systems for different application and and problem (complexity) classes.

The results of the experiments provide evidence for the wide range of applicability of DLV, and confirm that it is the best-suited ASP system for applications which require dealing with large amounts of data. The latter result is very important, also in the light of the emerging applications of ASP systems in the area of Information Integration in Databases (see Section 8).

The remainder of this paper is structured as follows. In the next section, we described the core language of DLV. After that, we consider in Section 3 knowledge representation and problem solving in DLV, where we present the **GCO** methodology and illustrate it on many examples. Section 4 is devoted to the complexity analysis of the DLV language. Section 5 surveys the DLV front-ends, while Section 6 gives an overview of the architecture and implementation of the DLV system. In Section 7 we then report about extensive experiments and benchmarking. Ongoing projects related to DLV are briefly addressed in Section 8. The final Section 9 contains some conclusions and an outlook to future work.

2 Core Language

In this section, we provide a formal definition of the syntax and the semantics of the kernel language of DLV, which is disjunctive datalog under the answer sets semantics [55][2] (which involves two kinds of negation), extended with weak constraints. For further background, see [73, 45, 55].

2.1 Syntax

Following Prolog's convention, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. In addition, DLV also supports positive integer constants and arbitrary string constants, which are embedded in double quotes. 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 *classical literal* l is either an atom p (in this case, it is *positive*), or a negated atom $\neg p$ (in this case, it is *negative*). A *negation as failure (NAF) literal* ℓ is of the form l or $\mathrm{not}\ l$, where l is a classical literal; in the former case ℓ is *positive*, and in the latter case *negative*. Unless stated otherwise, by *literal* we mean a classical literal.

Given a classical literal l, its *complementary* literal $\neg l$ is defined as $\neg p$ if $l = p$ and p if $l = \neg p$. A set L of literals is said to be *consistent* if, for every literal $l \in L$, its complementary literal is not contained in L.

Moreover, DLV also supports built-in predicates like equality, less-than, and greater-than, as well as various arithmetic predicates like addition or multiplication. For details, we refer to [52].

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

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \ \mathrm{not}\ b_{k+1}, ..., \ \mathrm{not}\ b_m$ is the *body* of r. A rule without head

[2] See footnote 4.

literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. 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.

The following notation will be useful for further discussion. If r is a rule of form (1), then $H(r) = \{a_1, \ldots, a_n\}$ is the set of the literals in the head and $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, 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 *disjunctive datalog program* (alternatively, *disjunctive logic program*, *disjunctive deductive database*) $\mathcal{P}$ is a finite set of rules. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*,[3] and a v-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *datalog program* (or *normal logic program*, *deductive database*).

The language of DLV extends disjunctive datalog by another construct, which is called *weak constraint* [14]. We define weak constraints as a variant of integrity constraints. In order to differentiate clearly between them, we use for weak constraints the symbol ':∼' instead of ' :– '. Additionally, a weight and a priority level of the weak constraint are specified explicitly.

Formally, a weak constraint wc is an expression of the form

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

where for $m \geq k \geq 0$, $b_1, \ldots, b_m$ are classical literals, 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 regular integrity constraints.

A DLV *program* $\mathcal{P}$ (*program*, for short) is a finite set of rules (possibly including integrity constraints) and weak constraints. In other words, a DLV program $\mathcal{P}$ is a disjunctive datalog program possibly containing weak constraints. For a program $\mathcal{P}$, let $WC(\mathcal{P})$ denote the set of weak constraints in $\mathcal{P}$, and let $Rules(\mathcal{P})$ denote the set of rules (including integrity constraints) in $\mathcal{P}$.

As usual, 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.

2.2 Semantics

The semantics of DLV programs extends the (consistent) answer sets semantics of disjunctive datalog programs, originally defined in [55], to deal with weak constraints. The semantics provided in this section is a slight generalization of the original semantics proposed for weak constraints in [14]. In particular, the original language allowed only "prioritized weak constraints", while the DLV language allows both priority levels (layers) and weights for weak constraints.

Herbrand Universe For any program $\mathcal{P}$, let $U_\mathcal{P}$ (the Herbrand Universe) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_\mathcal{P}$.

Herbrand Literal Base For any program $\mathcal{P}$, let $B_\mathcal{P}$ be the set of all ground (classical) literals constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$ (note that, for each atom p, $B_\mathcal{P}$ contains also the strongly negated literal $\neg p$).

[3] In positive programs negation as failure (not) does not occur, while strong negation (¬) may be present.

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

$$GroundRules(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r) \quad \text{and} \quad GroundWC(\mathcal{P}) = \bigcup_{w \in WC(\mathcal{P})} Ground(w).$$

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 disjunctive datalog programs, then we give a reduction of disjunctive datalog programs containing negation as failure to positive ones and use it to define answer sets of arbitrary disjunctive datalog programs, possibly containing negation as failure. Finally, we specify the way how weak constraints affect the semantics, defining the semantics of general DLV programs.

An interpretation I is a set of ground classical literals, i.e. $I \subseteq B_{\mathcal{P}}$ w.r.t. a program $\mathcal{P}$. A consistent interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive datalog 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 datalog program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.[4]

Example 2.1 The positive program $\mathcal{P}_1 = \{a \mathtt{\ v\ } \neg b \mathtt{\ v\ } c.\}$ has the answer sets $\{a\}$, $\{\neg b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \mathtt{\ v\ } \neg b \mathtt{\ v\ } c. \ ; \ \mathtt{:-}\ a.\}$ has the answer sets $\{\neg \mathtt{b}\}$ and $\{\mathtt{c}\}$. Finally, the positive program $\mathcal{P}_3 = \{a \mathtt{\ v\ } \neg b \mathtt{\ v\ } c. \ ; \ \mathtt{:-}\ a. \ ; \ \neg b \mathtt{\ :-\ } c. \ ; \ c \mathtt{\ :-\ } \neg b.\}$ has the single answer set the set $\{\neg b, c\}$.

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 2.2 Given the general program $\mathcal{P}_4 = \{a \mathtt{\ v\ } \neg b \mathtt{\ :-\ } c. \ ; \ \neg b \mathtt{\ :-\ } \mathrm{not}\ a, \mathrm{not}\ c. \ ; \ a \mathtt{\ v\ } c \mathtt{\ :-\ } \mathrm{not}\ \neg b.\}$ and $I = \{\neg \mathtt{b}\}$, the reduct $\mathcal{P}_4^I$ is $\{a \mathtt{\ v\ } \neg b \mathtt{\ :-\ } c. \ ; \ \neg b.\}$. 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 $J = \{\mathtt{a}\}$. The reduct $\mathcal{P}_4^J$ is $\{a \mathtt{\ v\ } \neg b \mathtt{\ :-\ } c. \ ; \ a \mathtt{\ v\ } c.\}$ 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 $K = \{\mathtt{c}\}$, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$: for the rule $r : a \mathtt{\ v\ } \neg b \mathtt{\ :-\ } c$, $B(r) \subseteq K$ 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 $\mathcal{P}_4$.

[4] Note that we only consider *consistent answer sets*, while in [55] also the inconsistent set of all possible literals can be a valid answer set.

Remark 2.3 In some cases, it is possible to emulate disjunction by unstratified normal rules by "shifting" the disjunction to the body [8, 26, 69], as shown in the following example. Consider $\mathcal{P}_5 = \{a \vee b.\}$ and its "shifted version" $\mathcal{P}'_5 = \{a \text{ :- not } b. \ ; \ b \text{ :- not } a.\}$. Both programs have the same answer sets, namely $\{a\}$ and $\{b\}$.

However, this is not possible in general. For example, consider $\mathcal{P}_6 = \{a \vee b. \ ; \ a \text{ :- } b. \ ; \ b \text{ :- } a.\}$ It has $\{a, b\}$ as its single answer set, while its "shifted version" $\mathcal{P}'_6 = \{a \text{ :- not } b. \ ; \ b \text{ :- not } a. \ ; \ a \text{ :- } b. \ ; \ b \text{ :- } a. \}$ has no answer set at all. □

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,[5] 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:

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

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; $N^{\mathcal{P}}_i(A)$ 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 DLV 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 2.4 Consider the following program $\mathcal{P}_{wc}$, which contains three weak constraints.

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

$Rules(\mathcal{P}_{wc})$ admits three answer sets: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, \neg d\}$, 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.

[5] 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.

3 Knowledge Representation in DLV

A main strength of DLV, compared to other answer set programming systems, is the wide range of its applicability. While other systems are somehow specialized on a particular class of problems (e.g., NP-complete problems), DLV is more "general purpose", as it is able to deal, with a reasonable degree of efficiency, with different kinds of applications, ranging from the more "database oriented" deductive database applications (where larger input data have to be dealt with), to NP search and optimization problems, up to very complex problems whose complexity resides at the second layer of the Polynomial Hierarchy (more precisely, in Σ_2^P and even in Δ_3^P).

In this section, we illustrate the usage of DLV as a tool for knowledge representation and reasoning. We first deal with a couple of classical deductive database applications. Then, we present a new programming methodology, which allows us to encode also hard queries and, more generally, search problems in a simple and highly declarative fashion; even optimization problems of complexity up to Δ_3^P may be declaratively encoded by using this methodology. Finally, we illustrate the usage of the methodology on a number of computationally hard problems.

3.1 Deductive Database Applications

First, we will present two problems motivated by classical deductive database applications, namely Reachability and Same Generation. Both can be encoded by using only positive datalog rules, thus just scratching the surface of DLV's expressiveness for knowledge representation.

3.1.1 Reachability

Given a finite directed graph $G = (N, A)$, we want to compute all pairs of nodes $(a, b) \in N \times N$ such that b is reachable from a through a nonempty sequence of arcs in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

In the DLV encoding, we assume that A is represented by the binary relation $arc(X, Y)$, where a fact $arc(a, b)$ means that there is an arc from a to b in G, i.e., $(a, b) \in A$; the set of nodes N is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts.

The following program then computes a relation $reachable(X, Y)$ containing all facts of the form $reachable(a, b)$ such that b is reachable from a through the arcs of the input graph G:

$$reachable(X, Y) \text{ :- } arc(X, Y).$$
$$reachable(X, Y) \text{ :- } arc(X, U), reachable(U, Y).$$

3.1.2 Same Generation

Given a parent-child relationship (an acyclic directed graph), we want to find all pairs of persons belonging to the same generation. Two persons are of the same generation, if (i) either they are siblings, or (ii) they are children of two persons of the same generation.

The input is provided by a relation $parent(X, Y)$ where a fact $parent(thomas, moritz)$ means that $thomas$ is a parent of $moritz$.

This problem can be encoded by the following program, which computes a relation $samegeneration(X, Y)$ containing all facts such that X is of the same generation as Y:

$$samegeneration(X, Y) \text{ :- } parent(P, X), parent(P, Y).$$
$$samegeneration(X, Y) \text{ :- } parent(P1, X), parent(P2, Y), samegeneration(P1, P2).$$

We stress at this point that different from Prolog, the order of the rules and the subgoals (literals) in the rule bodies does not matter and has no effect on the DLV program semantics. In particular, it does not affect termination of the program when it is evaluated, which is not guaranteed under Prolog semantics if, e.g., the last literal in the second rule is reordered to the first position. Thus, like other ASP systems, DLV supports a fully modular, rule-by-rule encoding of problems where the semantics of any rule is not affected by its context.

3.2 The GCO Declarative Programming Methodology

The core language of DLV can be used to encode problems in a highly declarative fashion, following a "**GCO**" (Guess/Check/Optimize) paradigm. In this section, we will describe this technique and we will then illustrate how to apply it on a number of examples. Many problems, also problems of comparatively high computational complexity (Σ_2^P-complete and Δ_3^P-complete problems), can be solved in a natural manner with DLV 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 uniformly 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, in a way such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

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

Optimization Part The optimization part $\mathcal{O} \subseteq \mathcal{P}$ of the program assigns costs to the individual resources represented in $\mathcal{P}$ (by means of weak constraints), permitting to filter solutions for minimality w.r.t. total cost.

The three layers above can also use additional auxiliary predicates, which can be seen as a background knowledge.

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

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

than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For many problems, however, a natural **GCO** program can be designed, in which the three parts are clearly identifiable and have the following simple structure:

- The guessing part $\mathcal{G}$ consists of some disjunctive rules which "guess" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints which check the admissibility of S.
- The optimization part $\mathcal{O}$ consists of weak constraints expressing the "desiderata" for the solutions, that is, expressing which properties the solutions *should possibly have*. The set $\mathcal{O}$ of weak constraints implicitly defines an objective function over the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$. The semantics of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ optimizes this function, providing the answer sets minimizing its value.

All three layers may also use additional auxiliary predicates, which are defined by normal stratified rules.[6] Such auxiliary predicates may also be associated with the guess for a candidate, and defined in terms of other guessed predicates, leading to a more "educated guess" which reduces blind guessing of auxiliary predicates; this will be seen in some examples below.

Thus, the disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches. Apart from the point made in Remark 2.3 regarding their difference from rules using unstratified negation in terms of semantics, disjunctive rules also have the advantage of higher compactness and usually also of a more natural reading. The weak constraints induce a modular ordering on the answer sets, allowing the user to specify the best solutions according to an optimization criterion.

Remark 3.1 The **GCO** programming methodology has positive implications also from the Software Engineering viewpoint. Indeed, the modular program structure in **GCO** allows for developing programs incrementally providing support for simpler testing and debugging activities. Indeed, one first writes the Guess module $\mathcal{G}$ and tests that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one deals with the Check module and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ are the admissible problem 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.

3.3 Applications of the GCO Programming Technique

In this section, we illustrate the declarative programming methodology described in Section 3.2 by showing its application on a number of concrete examples.

3.3.1 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 of this problem are provided by a set $\mathcal{F}$ 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.

[6] For a definition of stratification, see Section 4.2.

Several exams can be assigned to the same time slots (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 on the same time slot.

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

$S2_1$ first of all, minimize the overlapping between exams of the same year of C,

$S2_2$ then, minimize the overlapping between exams of different years of C.

The above problem can be encoded in the DLV language by the following **GCO** program $\mathcal{P}_{sch}$:

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

$$\left.\begin{array}{l} \text{:- } assign(Id, T), assign(Id', T), \\ \quad Id <> Id', exam(Id, P, C, Y), exam(Id', P, C', Y'). \end{array}\right\} \textbf{ Check}$$

$$\left.\begin{array}{l} \text{:}\sim assign(Id, T), assign(Id', T) \\ \quad exam(Id, P, C, Y), exam(Id', P', C, Y), Id <> Id'. \ [:2] \\ \text{:}\sim assign(Id, T), assign(Id', T) \\ \quad exam(Id, P, C, Y), exam(Id', P', C, Y'), Y <> Y', Id <> Id'. \ [:1] \end{array}\right\} \textbf{ Optimize}$$

The Guess module $\mathcal{G}$ consists of a single disjunctive rule, which defines the search space. It is evident that the answer sets of $\mathcal{G} \cup \mathcal{F}$ are the possible assignments of exams with time slots.

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

Finally, the Optimize module $\mathcal{O}$ consists of two weak constraints having different priority. Both weak constraints state that exams of the same curriculum should *possibly not* be assigned with the same time slot. However, the first one, which has higher priority (level 2), states this desire for the exams of the same year, while the second one, which has lower priority (level 1) states it for the exams of different years. The semantics of weak constraints, as given in Section 2.2, implies that $\mathcal{O}$ captures precisely the constraint $S2$ of the scheduling problem specification. Thus, the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F} \cup \mathcal{O}$ correspond precisely to the desired schedules.

3.3.2 Hamiltonian Path

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

Definition 3.2 (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 predicates $node$ (unary) and arc (binary), and the starting node is specified by the predicate $start$ (unary). Then, the following **GCO** program $\mathcal{P}_{hp}$ solves the problem

HAMPATH. [7]

$$
\left.\begin{array}{l} inPath(X,Y) \vee outPath(X,Y) \text{ :- } start(X), arc(X,Y). \\ inPath(X,Y) \vee outPath(X,Y) \text{ :- } reached(X), arc(X,Y). \end{array}\right\} \textbf{Guess}
$$

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

$$
\left.\begin{array}{l} reached(X) \text{ :- } inPath(Y,X). \end{array}\right\} \begin{array}{l}\textbf{Auxiliary} \\ \textbf{Predicate}\end{array}
$$

The two disjunctive rules guess a subset S of the given arcs to be in the path, while the rest of the program checks whether that subset 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 predicate $inPath$, and no further guessing is needed: from outside, we might see this assignment to $reached$ as an "educated guess" for this predicate, which is more appealing than a blind guess, using e.g. a rule $reached(X) \vee \neg reached(X) \text{ :- } node(X)$., and subsequent checking through rules and constraints whether $reached$ is compatible with the guess for $inPath$.

In turn, the predicate $reached$ influences through the second rule 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, and then a guess on an arc leaving from a reached node by the second rule, which is repeated until all reached nodes are treated.

In the Checking Part, the first two constraints check whether 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.

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 a Hamiltonian Cycle, and by dropping the last arc, we have a Hamiltonian Path.

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

This encoding is very flexible, and can be easily adapted to solve both the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result is 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), then we can easily impose openness by adding a further constraint $\text{:- } start(Y), inPath(_, Y)$. to the program (like in Prolog, in DLV the symbol '_' stands for an anonymous variable, whose value is of no interest). Then, the set S of selected arcs in an answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a. If, on the other hand, we want to compute a Hamiltonian Cycle, then we have just to strip off the literal $\text{not } start(X)$ from the last constraint of the program.

3.3.3 Traveling Salesperson

The Traveling Salesperson Problem (TSP) is a well-known optimization problem, widely studied also in Operation Research.

[7] Note that the optimization part is empty in this program.

Definition 3.3 (TSP) *Given a weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph, find a minimum-cost cycle (closed path) in G starting at a and passing through each node in V exactly once.*

It is well-known that finding an optimal solution to the Traveling Salesperson Problem (TSP) is intractable. Computing an optimal tour is both NP-hard and co-NP-hard. In fact, in [81] it was shown that deciding whether the cost of an optimal tour is even is Δ_2^P-complete.

A DLV encoding for the Traveling Salesperson Problem (TSP) can be easily obtained from an encoding of Hamiltonian Cycle by imposing optimization: each arc in the graph carries a weight and the tour is selected which has the minimum total weight.

Suppose then that the graph G is specified by using the predicates $node$ (unary) and arc (ternary), and that the starting node is specified by the predicate $start$ (unary).

We first modify the HAMPATH encoding $\mathcal{P}_{hp}$ in Section 3.3.2 to compute Hamiltonian Cycles, by stripping off literal $\text{not } start(X)$ from the last constraint of the program, as explained above. We then add an optimization part consisting of a single weak constraint

$$:\sim inPath(X, Y, C).\ [C : 1]$$

which tells to possibly avoid taking arcs having high cost in the path, and has the effect to filter out all answer sets for which the sum of the costs of arcs selected (using $inPath$) is minimal.

The full **GCO** program $\mathcal{P}_{tsp}$ solving the TSP problem is thus as follows:

$$\left.\begin{array}{l} inPath(X, Y, C) \text{ v } outPath(X, Y, C) \text{ :- } start(X), arc(X, Y, C). \\ inPath(X, Y, C) \text{ v } outPath(X, Y, C) \text{ :- } reached(X), arc(X, Y, C). \end{array}\right\} \textbf{Guess}$$

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

$$\left.\begin{array}{l} :\sim inPath(X, Y, C).\ [C : 1] \end{array}\right\} \textbf{Optimize}$$

$$\left.\begin{array}{l} reached(X) \text{ :- } inPath(Y, X, C). \end{array}\right\} \textbf{Auxiliary Predicate}$$

Given a set of facts $\mathcal{F}$ for $node$, arc, and $start$ which specifies the problem input, it is easy to see that the (optimal) answer sets of program $\mathcal{P}_{tsp} \cup \mathcal{F}$ are in a one-to-one correspondence with the optimal tours of the input graph.

3.3.4 Winners in Combinatorial Auctions

The weak constraints, specifying the Optimization module of a **GCO** program, permit to naturally express optimization criteria based on minimization. In this example, we show how also maximization can be expressed in the Optimize module, by a suitable use of weak constraints.

A combinatorial auction has a set $\mathcal{O}$ of objects for sale and a set $\mathcal{B}$ of bidders. Bidders may offer bids on a set of items, in terms of a price they are willing to pay for them. For example, bidder b_1 may bid \$500 for items a, b, c together and \$200 for item a alone. Formally, a *bid* is a triple of the form (b, X, p) where $b \in \mathcal{B}$, $X \subseteq \mathcal{O}$, and $p > 0$ is a price. The auctioneer receives a set **bids** of bids. Without loss of generality, we may assume that **bids** does not contain two different triples (b_1, X, p_1) and (b_2, X, p_2) (i.e., with the same set X

of items). This is because if two bids are received for the same set X of objects, the one with the lower price can be eliminated.[8]

The task of the auctioneer is to determine which bids to accept. Given **bids**, a *potential winner* is a subset $\mathbf{win} \subseteq \mathbf{bids}$ such that

$$(b_1, X_1, p_1), (b_2, X_2, p_2) \in \mathbf{win} \rightarrow X_1 \cap X_2 = \emptyset.$$

Each potential winner set represents a possible scenario, i.e., a possible outcome of the auction.

Potential winners may be of very different quality according to the auctioneer's point of view. Indeed, she clearly wants to maximize her revenue, and thus to determine a potential winner **win** maximizing the revenue $R(\mathbf{win})$, that is, maximizing the following function:

$$R(\mathbf{win}) = \sum_{(b,X,p)\in\mathbf{win}} p.$$

The *winner determination problem* is to find a potential winner $\mathbf{win}$ such that $R(\mathbf{win})$ is maximum.

We construct a **GCO** program $\mathcal{P}_{\mathbf{bids}}$ solving this problem as follows. Suppose that two input predicates *bids* and *requires* encode the input data about the bids $\mathbf{bids}$ received by the auctioneer. In particular, for each bid $(b, x, p) \in \mathbf{bids}$, $\mathcal{F}$ contains the fact $bids(b, x, p)$., and the facts describing the set of items x required by the bidder b, namely, a fact $requires(x, i)$., for each item $i \in x$.

The predicate *accept* encodes the potential winner set of bids chosen by the auctioneer. The rules of program $\mathcal{P}_{\mathbf{bids}}$ are:

$$accept(B, X) \vee reject(B, X) \text{ :- } bids(B, X, P). \quad \} \textbf{ Guess}$$

$$\left.\begin{array}{l} \text{:- } accept(B_1, X_1), accept(B_2, X_2), \\ \quad X_1 <> X_2, requires(X_1, I), requires(X_2, I). \end{array}\right\} \textbf{ Check}$$

$$\left.\begin{array}{l} \text{:}\sim reject(B, X), bids(B, X, P). \ [P : 1] \end{array}\right\} \textbf{ Optimize}$$

It is easy to see that the answer sets of $Rules(\mathcal{P}_{\mathbf{bids}}) \cup \mathcal{F}$ correspond exactly to the potential winners of the auction. Then, the weak constraint $\text{:}\sim\ reject(B, X), bids(B, X, P).\ [P : 1]$ minimizes the sum of the prices of the rejected bids, and it therefore maximizes the revenue, since the maximum of $R(\mathbf{win})$ corresponds precisely to the minimum of

$$\hat{R}(\mathbf{win}) = \left(\sum_{(b,X,p)\in\mathbf{bids}} p\right) - R(\mathbf{win}).$$

Thus, the optimal answer sets of $\mathcal{P}_{\mathbf{bids}} \cup \mathcal{F}$ correspond precisely to the optimal potential winners.

3.3.5 Ramsey Numbers

In the previous examples, we have seen how a search problem can be encoded in a DLV program whose answer sets correspond to the problem solutions. We next see another use of the **GCO** programming technique. We build a DLV program whose answer sets witness that a property does not hold, i.e., the property

[8] It is still possible that there are two bids of the form $(b_1, X, p_1), (b_2, X, p_2)$ with $b_1 \neq b_2$ and $p_1 = p_2$. In such a case most auctioneers eliminate one of the two bids using some pre-announced protocol – e.g. the bid received at a later time may be discarded.

at hand holds if and only if the DLV program has no answer set. Such a programming scheme is useful to prove the validity of co-NP or Π_2^P properties, respectively. We next apply the above programming scheme to a well-known problem of number and graph theory.

Definition 3.4 (RAMSEY) *The Ramsey number $R(k, m)$ is the least integer n such that, no matter how we color the arcs of the complete 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 [89]. We next show a program which allows us to determine whether a given integer n is the Ramsey Number $R(3, 4)$, knowing that no integer smaller than n is $R(3, 4)$. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. Consider the following **GCO** program $\mathcal{P}_{ramsey}$.

$$blue(X,Y) \vee red(X,Y) \text{ :- } arc(X,Y). \qquad \} \textbf{ Guess}$$

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

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red complete graph (i.e., a clique) on 3 nodes, and the second constraint eliminates the colorings containing a blue clique on 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, $n < R(3, 4)$. The smallest n such that no answer set is found is the Ramsey number $R(3, 4)$.

3.3.6 Strategic Companies

The problems considered so far are at the first level of the Polynomial Hierarchy (their complexities, in the ground case, do not exceed Δ_2^P, which we associate with the first level). We next show that also harder problems, located at the second level of the Polynomial Hierarchy, can be encoded by the **GCO** technique. To this end, we consider the following problem, which is known under the name *Strategic Companies*.

Definition 3.5 (STRATCOMP [15]) *Suppose there is a collection $C = \{c_1, \ldots, c_m\}$ of companies c_i owned by a holding, a set $G = \{g_1, \ldots, g_n\}$ of goods, for each c_i a set $G_i \subseteq G$ of goods produced by c_i, and a set $O_i \subseteq C$ of companies controlling (owning) c_i. This control can be thought of as a majority in shares; companies not in C, which we do not model here, might have shares in companies as well. O_i is referred to as the* controlling set *of c_i. Note that, in general, a company might have more than one controlling set, and only non-redundant controlling sets (i.e., no proper subset is a controlling set) are recorded then. Let the holding produce all goods in G, i.e. $G = \bigcup_{c_i \in C} G_i$.*

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

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

Computing the set of all strategic companies is relevant when companies should be sold, as selling any company which is strategic would for sure lead to a violation of any of conditions (1) and (2). This problem is Σ_2^P-hard in general [15]; reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is one of the rare KR problems from the business domain of this complexity that have been considered so far.

In the following, we adopt the setting from [15] where each product is produced by at most two companies (for each $g \in G \,|\{c_i \mid g \in G_i\}| \leq 2$) and each company is jointly controlled by at most three other companies, i.e. $|O_i| \leq 3$ for $i = 1, \ldots, m$ (in this case, the problem is still Σ_2^P-hard). Assume that $\mathcal{F}$ for a given instance of STRATCOMP contains the following facts:

- $company(c)$ 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 $c_j \neq c_k$,
  $prod_by(g, c_j, c_j)$, if $\{c_i \mid g \in G_i\} = \{c_j\}$,
- $contr_by(c_i, c_k, c_m, c_n)$, if $c_i \in C$, $O_i = \{c_k, c_m, c_n\}$, and $c_k \neq c_m \neq c_n$, $c_k \neq c_n$,
  $contr_by(c_i, c_k, c_m, c_m)$, if $c_i \in C$, $O_i = \{c_k, c_m\}$, and $c_k \neq c_m$,
  $contr_by(c_i, c_k, c_k, c_k)$, if $c_i \in C$ and $O_i = \{c_k\}$.

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

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

$$r_{s2}: \quad strat(W) \,\mathtt{:-}\, contr_by(W, X, Y, Z), strat(X), strat(Y), strat(Z). \qquad \Big\} \textbf{ Check}$$

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 $\mathcal{P}_{strat}$ encoding STRATCOMP, is surprisingly succinct, given that STRATCOMP is a hard (Σ_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 then naturally ensure that the answer sets of $\mathcal{F} \cup \{r_{s1}\}$ correspond to the strategic sets; no further checking would be needed. However, in case control information is available, the rule r_{s2} in the program checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets.

The answer sets of $\mathcal{P}_{strat} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$. Thus, a company c is strategic, if $strat(c)$ 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 rule r_{s1}. For example, suppose the guessing part gives rise to a ground rule r_{sg1}

$$strat(c1) \;\mathtt{v}\; strat(c2) \,\mathtt{:-}\, prod_by(g, c1, c2).$$

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 not both $strat(c1)$ and $strat(c2)$ can be derived from r_{sg1}.

"Feedback" or "influence" of the checking part $\mathcal{C}$ on the guessing part $\mathcal{G}$, in terms of literals which are derived in $\mathcal{C}$ and invalidate the application of rules in $\mathcal{G}$ or make further rules in $\mathcal{G}$ applicable (and thus change the guess), can be made precise in terms of a "potentially uses" relation [45] and a "splitting set" [71]. Such interference is in fact needed to solve STRATCOMP in the way we have outlined, if $\mathcal{C}$ does not contain disjunctive rules. This follows from complexity considerations: If $\mathcal{C}$ had no influence on $\mathcal{G}$ (augmented by $\mathcal{F}$), then the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}$ would just be the answers sets of $\mathcal{C} \cup A_0$, where A_0 is an answer set of $\mathcal{G} \cup \mathcal{F}$. Furthermore, if $\mathcal{C}$ contains no disjunctive rule and $\mathcal{G}$ has complexity in NP (e.g., if $\mathcal{G}$ is, as in this case, head-cycle free [8]), along with A_0 an answer set A of $\mathcal{C} \cup A_0$ can be guessed and both A_0 and A can be verified in polynomial time. The complexity of deciding whether an atom belongs to some answer set of such a feedback-free program is thus in NP. Now we recall that a particular company is strategic iff it belongs to some strategic set, and deciding this is a Σ_2^P-complete problem. Unless $\Sigma_2^P = \text{NP}$, a hypothesis that is widely believed to be false, we can thus conclude that it is impossible to solve the STRATCOMP problem by a program whose answer sets correspond to the strategic sets of the problem instance, but in which $\mathcal{C}$ has no influence on $\mathcal{G}$.

In general, if a program encodes a problem that is Σ_2^P-complete, then the checking part $\mathcal{C}$ must contain disjunctive rules unless $\mathcal{C}$ has influence on the guessing part $\mathcal{G}$ (or $\mathcal{G}$ has Σ_2^P-complexity). In particular, if the above program $\mathcal{P}_{strat}$ is rewritten to eliminate such influence, then further disjunctive rules must be added. An encoding which expresses the strategic sets in the generic **GCO**-paradigm, where the guessing and the checking part are clearly separated, can be found in [33].

3.3.7 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, if feasible, a decision 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 Σ_2^P to Δ_3^P. Nevertheless, DLV still can handle this quite naturally. Let us assume for simplicity 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 which belongs to every strategic set, and top is the highest priority number. Then, we can easily represent this more complicated problem, by just adding to the above encoding for STRATCOMP the following weak constraint:

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

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

4 The Complexity of the DLV Language

In this section, we analyze the computational complexity of the DLV language and of some relevant fragments thereof. The exploitation of the analysis of the computational complexity of DLV programs is one of the key factors of DLV's efficiency. Indeed, as we will point out in Section 6, the implementation of DLV recognizes syntactic subclasses of the language of DLV and employs "ad hoc" evaluation algorithms if the subclass has a lower complexity.

In the sequel of this section, we first provide some preliminaries on complexity theory. Subsequently, we define a couple of relevant syntactic properties of DLV programs, which allow us to single out computationally simpler subclasses of our language.Then, we define the main computational problems under consideration and derive their precise complexity. We conclude with a discussion, paying attention to the impact of syntactic restrictions.

4.1 A Reminder of the Polynomial Hierarchy

We assume that the reader has some acquaintance with the concepts of NP-completeness and complexity theory. We next provide a very short reminder of the complexity classes of the Polynomial Hierarchy which are relevant to this section. The book [82] is an excellent source for deepening the knowledge in this field.

The classes Σ_k^P, Π_k^P, and Δ_k^P of the *Polynomial Hierarchy (PH)* (cf. [63]) 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 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 π 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 π 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.

4.2 Relevant Fragments of the DLV Language

In this section, we introduce syntactic classes of DLV programs with a number of useful and interesting properties. First we need the following

Definition 4.1 Functions $||\ || : B_{\mathcal{P}} \rightarrow \{0, 1, \ldots\}$ from the ground (classical) literals of the Herbrand Literal 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 4.2 A disjunctive logic program $\mathcal{P}$ is called *(locally) stratified* [3], [87], 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 4.3 Consider the following two DLV programs.

$$\begin{array}{lll} \mathcal{P}_7: & p(a) \vee p(c) \text{ :- not } q(a). & \\ & p(b) \text{ :- not } q(b). & \end{array} \qquad \begin{array}{lll} \mathcal{P}_8: & p(a) \vee p(c) \text{ :- not } q(b). & \\ & q(b) \text{ :- not } p(a). & \end{array}$$

It is easy to see that program $\mathcal{P}_7$ is stratified, while program $\mathcal{P}_8$ is not. A suitable level mapping for $\mathcal{P}_7$ is the following:

$$||p(a)||_s = 2 \quad ||p(b)||_s = 2 \quad ||p(c)||_s = 2 \quad ||q(a)||_s = 1 \quad ||q(b)||_s = 1 \quad ||q(c)||_s = 1$$

As for $\mathcal{P}_8$, 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 4.4 A program $\mathcal{P}$ is called *head-cycle free (HCF)* [8], if there is a level mapping $||\ ||_h$ of $\mathcal{P}$ such that, for every rule r of $\mathcal{P}$,

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 4.5 Consider the following program $\mathcal{P}_9$.

$$\begin{array}{lll} \mathcal{P}_9: & a \vee b. & \\ & a \text{ :- } b. & \end{array}$$

It is easy to see that $\mathcal{P}_9$ is head-cycle free; an admissible level mapping for $\mathcal{P}_9$ is given by $||a||_h = 2$ and $||b||_h = 1$.

Consider now the program

$$\mathcal{P}_{10} = \mathcal{P}_9 \cup \{b \text{ :- } a.\}$$

Program $\mathcal{P}_{10}$ is not head-cycle free, since a and b should belong to the same level by Condition (1) of Definition 4.4, while they cannot by Condition (2) of the definition. Note, however, that $\mathcal{P}_{10}$ is stratified.

4.3 Main Problems Considered

As for the classical nonmonotonic formalisms, three important decision problems, corresponding to three different reasoning tasks, arise in the context of the DLV language:

Brave Reasoning. Given a DLV 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 DLV 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 DLV 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 the above decision problems, which are strongly relevant to the tasks performed by the DLV computational engine. Brave Reasoning is strictly related also to the problem of *finding* an answer set, which is to be solved, for instance, when a search problem (like Hamiltonian Path in Section 3) is encoded in a DLV program.

In the following, we analyze the computational complexity of the two decision problems specified above for ground (i.e., propositional) DLV programs; we shall address the case of non-ground programs at the end of this section.

An interesting issue, from the viewpoint of system implementation, 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 (nonmonotonic) negation (not_s),
- arbitrary (nonmonotonic) negation (not),
- head-cycle free disjunction ($\mathtt{v}_h$),
- arbitrary disjunction ($\mathtt{v}$),
- weak constraints (w). [9]

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 DLV[X] the fragment of the DLV language where the elements in X are allowed. For instance, DLV[$\mathtt{v}_h, \mathrm{not}_s$] denotes the fragment of the DLV language allowing head-cycle free disjunction and stratified negation (no weak constraints).

4.4 Derivation of Complexity Results

The computational complexity of the above problems for a number of fragments of the DLV language has been previously analyzed, cf. [45, 59, 14, 48, 43]. To obtain a full picture of the complexity of the fragments of the DLV language, we establish in this subsection the complexity characterization of the fragments which have not been studied yet. The reader who is not interested in the technicalities might jump directly to Section 4.5, which provides a summary and discussion of the results.

Throughout this section, we consider the ground case, i.e., we assume that programs $\mathcal{P}$ are ground, and unless stated otherwise, also atoms, literals etc. are ground. Furthermore, to simplify matters and stay in line with results from the literature, we shall tacitly restrict here the language fragments by disregarding strong negation and integrity constraints in programs. However, this is insignificant inasmuch as the results in presence of these constructs are the same (see, e.g., [14]). Some remarks on the complexity and expressiveness of non-ground programs are provided at the end of Section 4.5.

We start by analyzing the complexity of cautious reasoning for the full DLV language (i.e., for the fragment DLV[$\mathtt{v}, \mathrm{not}, w$]). To determine the upper complexity bound, we first prove two lemmas.

Lemma 4.6 *Given a* DLV[$\mathtt{v}, \mathrm{not}, w$] *program* $\mathcal{P}$ *and an integer* $n \geq 0$ *as input, deciding whether some answer set* M *of* $Rules(\mathcal{P})$ *exists such that* $H^{\mathcal{P}}(M) \leq n$ *is in* Σ_2^P.

[9] Following [14], 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 the DLV language we consider.

Proof. We can decide the problem as follows. Guess $M \subseteq B_{\mathcal{P}}$, and check that: (1) M is an answer set of $Rules(\mathcal{P})$, and (2) $H^{\mathcal{P}}(M) \leq n$. Clearly, property (2) can be checked in polynomial time; while (1) can be decided by a single call to an NP oracle, cf. [74, 45]. The problem is therefore in Σ_2^P. □

Lemma 4.7 *Given a* DLV[$\mathtt{v}$, not, w] *program $\mathcal{P}$, a positive integer n, and an atom A as input, deciding whether there exists an answer set M of $Rules(\mathcal{P})$ such that $H^{\mathcal{P}}(M) = n$ and $A \notin M$ is in Σ_2^P.*

Proof. Similar as above, guess $M \subseteq B_{\mathcal{P}}$ and check that: (1) M is an answer set of $Rules(\mathcal{P})$, (2) $H^{\mathcal{P}}(M) = n$, and (3) $A \notin M$. Clearly, property (2) and (3) can be checked in polynomial time; while (1) can be decided by a single call to an NP oracle , cf. [74, 45]. The problem is therefore in Σ_2^P. □

We are now in the position to determine the precise complexity of cautious reasoning over full DLV programs.

Theorem 4.8 *Given a* DLV[$\mathtt{v}$, not, w] *program $\mathcal{P}$, and an atom A as input, deciding whether A is true in all answer sets of $\mathcal{P}$ is Δ_3^P-complete. Hardness holds even if $\mathcal{P}$ is a* DLV[$\mathtt{v}$, w] *program.*

Proof. *Membership*. We prove that the complementary problem is in Δ_3^P as follows.

Given $\mathcal{P}$, let $u = u(\mathcal{P})$ be the value that the objective function $H^{\mathcal{P}}(M)$ from Section 2.2 takes when an interpretation M would violate all weak constraints in $WC(\mathcal{P})$ (in each layer). More precisely, $u(\mathcal{P}) = \sum_{i=1}^{l_{max}^{\mathcal{P}}}(f_{\mathcal{P}}(i) \cdot \sum_{w \in WC_i} weight(w))$, where WC_i is the set of weak constraints in layer i (see Section 2.2).

Clearly, u is an upper bound on the cost s^* of any optimal answer set of $\mathcal{P}$. By a binary search on $[0..u]$, we can compute s^* using an oracle which decides, given $\mathcal{P}$ and an integer $n \geq 0$, the existence of an answer set M of $Rules(\mathcal{P})$ such that $H^{\mathcal{P}}(M) \leq n$ (on the first call, $n = k/2$; then if the oracle answers "yes," $n = k/4$; otherwise, n is set to $k/2 + k/4$, and so on, according to standard binary search).

Observe that u is computable in polynomial time but, because of binary number representation, its *value* might be exponential in the size of the input. However, the number of calls to the oracle is logarithmic in u, and thus polynomial in the size of the input. Furthermore, the oracle employed is in Σ_2^P by virtue of Lemma 4.6. Thus, computing s^* as above is possible in polynomial time with polynomially many calls to a Σ_2^P oracle.

Finally, a further call to a Σ_2^P oracle verifies that there is an answer set M of $Rules(\mathcal{P})$ such that $A \notin M$ and $H^{\mathcal{P}}(M) = s^*$ (this is feasible in Σ_2^P from Lemma 4.7).

In summary, disproving that A is true in all answer sets of $\mathcal{P}$ is in Δ_3^P. Since co-$\Delta_3^P = \Delta_3^P$, cautious reasoning on full DLV programs is in Δ_3^P as well.

Hardness. We reduce brave reasoning on DLV[$\mathtt{v}$, w] programs, which was shown to be Δ_3^P-hard in [14], to cautious reasoning on DLV[$\mathtt{v}$, w] programs.

Given a DLV[$\mathtt{v}$, w] program $\mathcal{P}$ and an atom A, we build a DLV[$\mathtt{v}$, w] program $\mathcal{P}'$ such that $\mathcal{P} \models_b A$ if and only if $\mathcal{P}' \models_c A$.

Consider first the program $\mathcal{P}_1$, which is obtained from $\mathcal{P}$ by the following transformations: (i) shift up the priority level of each weak constraint by 1; (ii) add the weak constraint $:\sim \mathrm{not}\ A.\ [:1]$ (note that this is the only weak constraint in the lowest layer).

If $\mathcal{P} \not\models_b A$, then nothing changes, the (optimal) answer sets of $\mathcal{P}_1$ are the (optimal) answer sets of $\mathcal{P}$. If $\mathcal{P} \models_b A$, then the newly added weak constraint filters out the answer sets of $\mathcal{P}$ where A is false. In this case, the optimal answer sets of $\mathcal{P}_1$ are precisely the optimal answer sets of $\mathcal{P}$ containing A. However, the equivalence "$\mathcal{P} \models_b A$ iff $\mathcal{P}_1 \models_c A$" does not hold if $\mathcal{P}$ has no answer set, as $\mathcal{P}_1 \models_c A$ while $\mathcal{P} \not\models_b A$ in this

case. To handle this case, we transform $\mathcal{P}_1$ into $\mathcal{P}'$ as follows: (i) add the disjunctive fact $w \vee w'.$, where w and w' are fresh atoms; (ii) add the atom w' to the body of every rule of $\mathcal{P}_1$ (including constraints); and (iii) add the weak constraint $:\sim w.\ [:\ top]$, where top is higher than the maximum layer of $\mathcal{P}_1$. If $\mathcal{P}$ has some answer set, then the (optimal) answer sets of $\mathcal{P}'$ are precisely the same as the optimal answer sets of $\mathcal{P}_1$. Otherwise, $\mathcal{P}'$ has precisely one answer set, namely $\{w\}$. We have therefore reached our goal, as $\mathcal{P} \models_b A$ iff $\mathcal{P}' \models_c A$ and $\mathcal{P}'$ is obviously constructible from $\mathcal{P}$ in polynomial time (in fact, in logarithmic space). □

We next investigate the impact of limiting the form of disjunction by disallowing positive recursion through disjunction (HCF programs). We first derive the analogs of Lemmas 4.6 and 4.7 for the fragment DLV$[\vee_h, \mathrm{not}, w]$, which follows by a similar proof using the fact that answer set checking for a head-cycle free program can be done in polynomial time [8], rather than being co-NP-complete as in the general case.

Lemma 4.9 *Given a* DLV$[\vee_h, \mathrm{not}, w]$ *program $\mathcal{P}$, and a positive integer n, deciding whether there exists an answer set M of $Rules(\mathcal{P})$ such that $H^{\mathcal{P}}(M) \leq n$ is in NP.*

Proof. We guess $M \subseteq B_{\mathcal{P}}$, and check that: (1) M is an answer set of $Rules(\mathcal{P})$, and (2) $H^{\mathcal{P}}(M) \leq n$. Both of the above properties can be checked in polynomial time. Indeed, since the program is head-cycle free, answer-set checking is feasible in polynomial time [8]. The problem is therefore in NP. □

Lemma 4.10 *Given a* DLV$[\vee_h, \mathrm{not}, w]$ *program $\mathcal{P}$, a positive integer n, and an atom A as input, deciding whether there exists an answer set M of $Rules(\mathcal{P})$ such that $H^{\mathcal{P}}(M) = n$ and $A \notin M$ is in NP.*

Proof. We guess $M \subseteq B_{\mathcal{P}}$, and check that: (1) M is an answer set of $Rules(\mathcal{P})$, (2) $H^{\mathcal{P}}(M) = n$, and (3) $A \notin M$. Clearly, property (1), (2) and (3) can be checked in polynomial time. In particular, for property (1), it follows from the fact that the program is head-cycle free [8]. The problem is therefore in NP. □

Theorem 4.11 *Given a* DLV$[\vee_h, \mathrm{not}, w]$ *program $\mathcal{P}$, and an atom A as input, deciding whether A is true in all answer sets of $\mathcal{P}$ is Δ_2^P-complete. Hardness holds even if $\mathcal{P}$ is either a* DLV$[\vee_h, w]$ *or a* DLV$[\mathrm{not}, w]$ *program.*

Proof. *Membership.* To prove that the complementary problem is in Δ_2^P, we proceed as in the membership proof of Theorem 4.8. From Lemma 4.9 and Lemma 4.10, this time the oracle needed is in NP (instead of Σ_2^P). Therefore, cautious reasoning on DLV$[\vee_h, \mathrm{not}, w]$ programs is in Δ_2^P.

Hardness. We reduce brave reasoning on DLV$[\vee_h, w]$ programs, which was shown to be Δ_2^P-hard in [14], to cautious reasoning on DLV$[\vee_h, w]$ programs. The reduction is precisely the same as in the Hardness proof of Theorem 4.8. Note that the program $\mathcal{P}'$ resulting from the reduction is in DLV$[\vee_h, w]$, if the original program $\mathcal{P}$ is in DLV$[\vee_h, w]$: the addition of the disjunctive fact does not affect head-cycle freeness, and negation in weak constraints is allowed in the fragment DLV$[\vee_h, w]$.

Concerning the case of DLV$[\mathrm{not}, w]$, recall that brave reasoning on DLV$[\mathrm{not}, w]$ is also Δ_2^P-hard [14]. Now, assume that $\mathcal{P}$ is a DLV$[\mathrm{not}, w]$ program, and apply again the same reduction as in the hardness part of the proof of Theorem 4.8, but for the following slight change. Since disjunction is not allowed in DLV$[\mathrm{not}, w]$ programs, we replace the disjunctive fact $w \vee w'.$ by two rules with (unstratified) negation: w :– not $w'.$ and w' :– not w. Evidently, the meaning of the program remains unchanged, and we obtain a reduction from brave reasoning on DLV$[\mathrm{not}, w]$ programs to cautious reasoning for DLV$[\mathrm{not}, w]$. □

Next, we point out the complexity of cautious reasoning over the "easy" fragments of the DLV language, where existence and uniqueness of an answer set is guaranteed.

Theorem 4.12 *Given a* DLV$[\mathrm{not}_s, w]$ *program* $\mathcal{P}$*, and an atom* A *as input, deciding whether* A *is true in all answer sets of* $\mathcal{P}$ *is* P*-complete. Hardness holds even if* $\mathcal{P}$ *is a* DLV[] *program.*

Proof. Each program in the fragments DLV$[\mathrm{not}_s, w]$ and DLV$[w]$ has precisely one answer set. Consequently, on these fragments brave and cautious reasoning coincide, and the statement follows from the results on the complexity of brave reasoning derived in [14]. (The introduction of priority levels for weak constraints does not affect these results, as priority levels can be easily converted to plain weights as we have seen in Section 2.2 and the beginning of Section 4.4.) □

Finally, we address the problem of answer set checking. In particular, we study here the complexity of all fragments of the DLV language where weak constraints are allowed; for the other ones, such results are (often implicitly) contained in preliminary papers.

First we show an interesting correspondence between the problem of cautious reasoning for programs without weak constraints and the problem of answer set checking for the corresponding fragment where weak constraints may occur in the programs.

Lemma 4.13 *Let* $\mathcal{P}$ *be a* DLV[v] *program and* q *an atom. Then, there exists a* DLV[v , w] *program* $\mathcal{P}'$ *and a model* M *for* $\mathcal{P}'$*, such that* M *is an answer set for* $\mathcal{P}'$ *if and only if* $q \notin M'$ *for each answer set* M' *of* $\mathcal{P}$*. Moreover,* $\mathcal{P}'$ *and* M *are computable in polynomial time from* $\mathcal{P}$ *and* q*, and, if* $\mathcal{P}$ *belongs to* DLV[v $_h$], *then* $\mathcal{P}'$ *belongs to* DLV[v $_h$, w].

Proof. Let $\bar{q}$, $\bar{q}'$, and $n\bar{q}$ be fresh atoms. Then, define the following program $\mathcal{P}'$:

$\bar{q}$ v $n\bar{q}$.
$\bar{q}'$:– $\bar{q}$.
p :– $\bar{q}$. for all $p \in B_{\mathcal{P}} - \{q\}$,
q v $\bar{q}'$:– $B(r)$. for all $r \in \mathcal{P}$ such that $q \in H(r)$,
$H(r)$:– $B(r)$. for all $r \in \mathcal{P}$ such that $q \notin H(r)$,
:∼ $\bar{q}$. $[1:1]$
:∼ $n\bar{q}$, not q. $[1:1]$

First, consider the interpretation $\hat{M} = B_{\mathcal{P}'} - \{q, n\bar{q}\}$. Note that $\hat{M}$ is an answer set for $Rules(\mathcal{P}')$, because it is clearly closed under $Rules(\mathcal{P}')$ and it is minimal. Indeed, $\hat{M}$ is the only minimal closed interpretation for $\mathcal{P}'$ containing $\bar{q}$, and all closed interpretations not containing $\bar{q}$ must contain $n\bar{q}$ and are hence incomparable with $\hat{M}$ (w.r.t. set inclusion). Moreover, observe that the cost of $\hat{M}$ is $H^{\mathcal{P}'}(\hat{M}) = 1$.

We show that $\hat{M}$ is not an (optimal) answer set for $\mathcal{P}'$ if and only if there is some answer set M' for $\mathcal{P}$ containing q.

Indeed, suppose that $\hat{M}$ is not an optimal answer set for $\mathcal{P}'$. Then, there is an (optimal) answer set M for $\mathcal{P}'$ having cost $H^{\mathcal{P}'}(M) = 0$, i.e., all weak constraints are satisfied. By the first one, we conclude that $\bar{q} \notin M$, which, by the first rule in $\mathcal{P}'$, implies that $n\bar{q} \in M$. Thus, by the last weak constraint, we have $q \in M$. By the minimality of M, we furthermore obtain that $\bar{q}' \notin M$. Now it is easy to see that $M' = M - \{n\bar{q}\}$ is an answer set of $\mathcal{P}$ which contains q. Conversely, consider any answer set M' for $\mathcal{P}$ containing the atom q. It is easy to see that $M = M' \cup \{n\bar{q}\}$ is an answer set for $Rules(\mathcal{P}')$. Moreover, $H^{\mathcal{P}'}(M) = 0$; hence, $\hat{M}$ is not an optimal answer set for $\mathcal{P}'$. This proves the claim.

Clearly, $\mathcal{P}'$ is computable in polynomial time from $\mathcal{P}$ and q. Moreover, if $\mathcal{P}$ is head-cycle free, then $\mathcal{P}'$ is head-cycle free, too. Indeed, a new bad cycle would imply that one of the fresh atoms $\bar{q}$, $\bar{q}'$, and $n\bar{q}$ occurs

in the body of some rule and also in the head of some rule with nonempty body. Obviously, this is not the case. □

Theorem 4.14 *Checking whether a given model M is an answer set for a* DLV *program $\mathcal{P}$ is*

1. P*-complete, if $\mathcal{P}$ belongs to* DLV$[\mathrm{not}_s, w]$. *Hardness holds even if $\mathcal{P}$ is positive.*
2. *co-NP-complete, if $\mathcal{P}$ belongs to* DLV$[\mathtt{v}_h, \mathrm{not}, w]$. *Hardness holds even if $\mathcal{P}$ is either positive or non-disjunctive.*
3. Π_2^P*-complete, if $\mathcal{P}$ belongs to* DLV$[\mathtt{v}, \mathrm{not}, w]$. *Hardness holds even if $\mathcal{P}$ is positive.*

Proof. 1). A DLV$[\mathrm{not}_s, w]$ program $\mathcal{P}$ has a unique answer set, computable in polynomial time. Thus, weak constraints do not affect the complexity of this problem, which remains the same as for DLV$[\mathrm{not}_s]$ and DLV[] programs, namely P-completeness. The hardness part for the latter is obtained by an easy reduction from deciding whether an atom A is in the answer set of a DLV[] program $\mathcal{P}$: just add clauses p :– A. for all $p \in B_\mathcal{P}$ and check whether $B_\mathcal{P}$ is the answer set of the resulting program.

2). *Membership.* Let $\mathcal{P}$ be a program in DLV$[\mathtt{v}_h, \mathrm{not}, w]$, and M be a set of ground literals. We can check that M is not an answer set for $\mathcal{P}$ as follows. First we verify in polynomial time whether M is an answer set for $Rules(\mathcal{P})$. If this is not the case, we stop. Otherwise, we compute its cost, say $c = H^\mathcal{P}(M)$, and then decide whether there exists an answer set M' for $\mathcal{P}$ such that $H^\mathcal{P}(M') < c$. From Lemma 4.9, this latter task is feasible in NP, and thus the checking problem is in co-NP.

Hardness. Recall that, given a DLV$[\mathtt{v}_h]$ program $\mathcal{P}$ and an atom q, deciding whether $q \notin M$ for each answer set of $\mathcal{P}$ is co-NP-complete, cf. [48]. From Lemma 4.13, this problem can be reduced to answer set checking for DLV$[\mathtt{v}_h, w]$ programs.

Moreover, it is well-known that, for each DLV$[\mathtt{v}_h]$ program $\mathcal{P}$, we can construct in polynomial time a DLV$[\mathrm{not}]$ program having the same answer sets as $\mathcal{P}$, by replacing disjunction by unstratified negation [8]. The same reduction clearly allows us to reduce answer set checking for DLV$[\mathtt{v}_h, w]$ programs to answer set checking for DLV$[\mathrm{not}, w]$ programs. Thus, the latter problem is co-NP-hard, as well.

3). *Membership.* Let $\mathcal{P}$ be a program in DLV$[\mathtt{v}, \mathrm{not}, w]$, and M be a set of ground literals. We show that the complementary problem of checking that M is not an answer set for $\mathcal{P}$ is in Σ_2^P. First we decide in co-NP whether M is an answer set for $Rules(\mathcal{P})$ or not. If this is not the case, we stop. Otherwise, we compute its cost, say $c = H^\mathcal{P}(M)$, and then decide whether there exists an answer set M' for $\mathcal{P}$ such that $H^\mathcal{P}(M') < c$. This is feasible in Σ_2^P, according to Lemma 4.6.

Hardness. Deciding whether a given literal is not contained in any answer set of a DLV$[\mathtt{v}]$ program is Π_2^P-complete [45]. From Lemma 4.13 it follows that this problem can be reduced to answer set checking for DLV$[\mathtt{v}, w]$ programs. □

4.5 Summary of Results and Discussion

The complexity of Brave Reasoning and Cautious Reasoning from ground DLV programs are summarized in Table 1 and Table 2, respectively. In Table 3, we report both well-known (for the weak constraint-free case) and new 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

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	NP	Δ_2^P
$\{\mathtt{v_h}\}$	NP	Δ_2^P	NP	Δ_2^P	NP	Δ_2^P
$\{\mathtt{v}\}$	Σ_2^P	Δ_3^P	Σ_2^P	Δ_3^P	Σ_2^P	Δ_3^P

Table 1: The Complexity of Brave Reasoning in fragments of the DLV Language

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	co-NP	Δ_2^P
$\{\mathtt{v_h}\}$	co-NP	Δ_2^P	co-NP	Δ_2^P	co-NP	Δ_2^P
$\{\mathtt{v}\}$	co-NP	Δ_3^P	Π_2^P	Δ_3^P	Π_2^P	Δ_3^P

Table 2: The Complexity of Cautious Reasoning in fragments of the DLV Language

specify the support for negation and weak constraints. For instance, $\{w, \mathrm{not}_s\}$ denotes weak constraints 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 1, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under polynomial time reductions, and in fact under LOGSPACE reductions. All results reported in the tables are either proved in Section 4.4 or emerge from [45, 59, 48, 43, 14]. 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 [14] 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 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 Σ_2^P-complexity, if 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. Indeed, the presence of weak constraints increases the complexity of reasoning from NP and Σ_2^P to Δ_2^P and Δ_3^P, respectively.

Table 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-P $=$ 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 Σ_2^P, full disjunction alone is not sufficient to raise the complexity of cautious reasoning from co-NP to Π_2^P. Cautious reasoning remains in co-NP if default negation is disallowed. Intuitively, to disprove

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	P	co-NP
$\{\mathrm{v_h}\}$	P	co-NP	P	co-NP	P	co-NP
$\{\mathrm{v}\}$	co-NP	Π_2^P	co-NP	Π_2^P	co-NP	Π_2^P

Table 3: The Complexity of Answer Set Checking in fragments of the DLV Language

that an atom A is a cautious consequence of a program $\mathcal{P}$, it is sufficient to find a *model* M of $\mathcal{P}$ (which need not be an answer set or a minimal model) which does not contain A. Indeed, for 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 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, then the complexity of brave reasoning jumps up to the second level of the Polynomial Hierarchy (Σ_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 (Δ_2^P). Indeed, brave reasoning on full DLV 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 3); and 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 (Δ_3^P) on the fragments preserving the 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 Σ_2^P. The complexity goes down to the first level of PH if source (s_2) is eliminated, being in the class NP or Δ_2^P according to whether weak constraints (source (s_3)) are present or not. Finally, avoiding source (s_1) the complexity falls down to P, as source (s_2) is automatically eliminated, and source (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, Δ_2^P to EXPTIME$^{\mathrm{NP}}$, Σ_2^P to NEXPTIME$^{\mathrm{NP}}$, etc. For disjunctive programs and certain fragments of the DLV language, complexity results in the non-ground case have been derived e.g. in [45, 48]. For the other fragments, the results can be derived using complexity upgrading techniques [45, 60]. Answer Set Checking, however, increases only exponentially in presence of weak

constraints up to co-NEXPTIME$^{\text{NP}}$; 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.

Finally, we remark that, viewed as a database-style query language (cf. [22]), the DLV language without weak constraints captures the classes of Σ_2^P and Π_2^P queries under brave and cautious reasoning, respectively, as follows from the results of [45]. As can be seen, DLV with weak constraints is more expressive, and captures the class of Δ_3^P queries. Thus, the full language of DLV is strictly more expressive than Disjunctive Datalog (unless the polynomial hierarchy collapses). For instance, the Preferred Strategic Companies problem, described in Section 3.3.7, can be naturally expressed in the full DLV language; while it cannot be expressed at all if weak constraints are disallowed.

5 DLV Front-Ends

The DLV system consists of a kernel, which takes as input a DLV program as defined in Section 2, and several front-ends for various other knowledge representation and reasoning formalisms.

As of this writing, the DLV distribution contains front-ends for inheritance, model-based diagnosis, planning, and SQL3. A number of further, "external" front-ends have been developed on top of DLV by other research teams.

Each of these front-ends reads input in its language and transforms it into a DLV program. It then invokes the DLV kernel, post-processes the answer sets generated by the kernel, and possibly also controls the number of answer sets the kernel generates by indicating whether to continue or not after some answer set has been found and processed. We next provide a short overview of such front-ends, giving appropriate references to the sources of in-depth descriptions, and to the ways how executables can be obtained for experiments.

5.1 Inheritance Front-End

The inheritance front-end of DLV supports an extension of the language presented in Section 2, called DLP$^<$ [13]. This extension allows for grouping rules into objects and for defining a partial order among these objects, which mimics inheritance and specifies the "plausibility" of the rules, according to which possible conflicts should be solved.

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

The program $\mathcal{P}(o)$ for an object o is the set $\{o' \mid o < o'\}$ for a given partial order $<$ on $\mathcal{O}$, which defines specificity among objects.

In addition to the closure condition defined in Section 2, a DLP$^<$ rule may also be overridden, which informally means that for all literals in the head of this rule, a complementary literal is supported by a more specific (w.r.t. the partial ordering $<$) rule. For details, we refer to [13].

Example 5.1 Consider the following program $\mathcal{P}_{tweety}$:

$$\begin{array}{ll} bird & \{\ flies.\ \} \\ penguin : bird & \{\ \neg flies.\ \} \\ tweety : penguin & \{\ \ \} \end{array}$$

$\mathcal{P}_{tweety}$ consists of three objects ($bird$, $penguin$, and $tweety$). The partial order $<$ on the objects is specified in the program by its transitive reduction, which is denoted using the colon. The transitive reduction in the example is $tweety : penguin$ and $penguin : bird$, so the corresponding partial order consists of $tweety < penguin$, $penguin < bird$, and $tweety < bird$. Each object is followed by its rules, which are enclosed in "{" and "}".

$\mathcal{P}_{tweety}$ has exactly one answer set: $\{\neg flies\}$, which fully captures the intuitive meaning of the program. The rule $flies$ in object $bird$ would violate the closure condition in Section 2, but this violation is allowed because the rule is overridden by $\neg flies$ in the more specific object $penguin$.

It is important to note that $\mathrm{DLP}^<$ directly generalizes the kernel language of DLV since an ordinary DLV program exactly corresponds to a $\mathrm{DLP}^<$ program which involves a single object. Equally important is the fact that this language extension comes "for free" in terms of complexity (which is exactly the same as for the core language), while it serves as a more natural representation tool for reasoning with exceptions. Also frame axioms can be represented very naturally, as shown in [13].

In the same paper a reduction from $\mathrm{DLP}^<$ to the DLV kernel language is described. We have implemented this transformation as part of a complete $\mathrm{DLP}^<$ front-end on top of DLV. On the web page [49] the prototype is described in detail, and executables are available for various platforms.

5.2 Diagnosis Front-End

In diagnosis, given a background theory, some actual observations, and a set of hypotheses that usually indicate possible failures, one tries to find a subset of the hypotheses which can explain the observations by means of the background theory.

DLV provides two different diagnosis front-ends: one for *abductive diagnosis* [84, 44], and one for *Reiter's consistency-based diagnosis* [91], both of which currently support three different modes: general diagnosis, where all diagnoses are computed, subset minimal diagnosis[10], and single failure diagnosis.

The diagnostic theory of the abductive diagnosis front-end obeys the syntax described in Section 2.

Hypotheses (resp. observations) are lists of atoms (resp. NAF literals) separated by a dot (the "." character).

Intuitively, the front-end works as follows: After reading the input, rules for guessing possible diagnosis candidates are generated from the hypotheses, while the observations become constraints that forbid the generation of inconsistent diagnoses. In the case of subset minimal diagnosis, further rules are added for pruning non-minimal solutions. Finally, the DLV kernel is invoked, and for each reported answer set, the corresponding diagnosis is returned.

Example 5.2 (Network Diagnosis) Suppose that machine *a* is online in the computer network depicted in Figure 1, but we observe that it cannot reach machine *e*. Which machines are offline?
This reasoning task can be easily modeled as a diagnosis problem, where the theory is

$$reaches(X, X) \text{ :- } node(X), \text{not } \mathit{offline}(X).$$
$$reaches(X, Z) \text{ :- } reaches(X, Y), connected(Y, Z), \text{not } \mathit{offline}(Z).$$

the observations are

$$\text{not } \mathit{offline}(a). \text{ not } reaches(a, e).$$

[10] For positive non-disjunctive theories only.

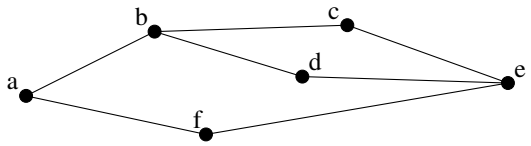


Figure 1: Computer network

and the hypotheses are

$$\mathit{offline}(a).\ \mathit{offline}(b).\ \mathit{offline}(c).$$
$$\mathit{offline}(d).\ \mathit{offline}(e).\ \mathit{offline}(f).$$

Using the diagnosis front-end, we can now easily determine all possible diagnoses, among them $\{\mathit{offline}(e)\}$, $\{\mathit{offline}(b), \mathit{offline}(f)\}$, or $\{\mathit{offline}(c), \mathit{offline}(d), \mathit{offline}(f)\}$.

The front-end for consistency-based diagnosis, as described in [91], is similar to the one for abductive diagnosis, but adopts the consistency-based diagnosis model instead of the abductive one.

5.3 Planning Front-End

The planning front-end of DLV, called DLV$^{\mathcal{K}}$, implements answer set planning on top of the DLV system. It supports the declarative language $\mathcal{K}$ [36, 38] and its extension $\mathcal{K}^c$ [39]. The language $\mathcal{K}$ is syntactically similar to the action language $\mathcal{C}$ [58], but semantically closer to answer set programming (by including default negation, for example). $\mathcal{K}$ and $\mathcal{K}^c$ offer the following distinguishing features:

- *Handling of incomplete knowledge:* for a fluent f, neither f nor its opposite $\neg f$ need to be known in any state.
- *Nondeterministic effects:* actions may have multiple possible outcomes.
- *Optimistic and secure (conformant) planning:* construction of a "credulous" plan or a "skeptical" plan, which works in all cases.
- *Parallel actions:* More than one action may be executed simultaneously.
- *Optimal cost planning:* In $\mathcal{K}^c$, one can assign an arbitrary cost function to each action, and the total costs of the plan are minimized.

To give a flavor of DLV$^{\mathcal{K}}$, we show the encoding of a planning problem in the well-known blocks world. We assume that the reader is acquainted with action languages and the notion of actions, fluents, goals, and plans; see e.g. [56], for further background.

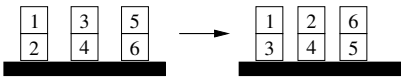


Figure 2: A blocks world example.

Example 5.3 (Blocks World) The (static) background knowledge of our blocks world is the following logic program:

$block(1).\ block(2).\ block(3).\ block(4).\ block(5).\ block(6).$
$location(table).\ location(B)$:- $block(B).$

Referring to Figure 2, we want to turn the initial configuration of blocks into the goal state in three steps, concurrent moves allowed.

The $\mathcal{K}$ domain description uses an action `move` and the fluents `on` and `blocked`. In this simple example, we assume that in the initial state the locations of all blocks are known. The domain is described by the following $\texttt{DLV}^{\mathcal{K}}$ program:

$fluents:$ $on(B, L)\ requires\ block(B),\ location(L).$
$blocked(B)\ requires\ block(B).$

$actions:$ $move(B, L)\ requires\ block(B),\ location(L)\ costs\ 1.$

$always:$ $executable\ move(B, L)\ if\ B <> L.$
$nonexecutable\ move(B, L)\ if\ blocked(B).$
$nonexecutable\ move(B, L)\ if\ blocked(L).$
$nonexecutable\ move(B, L)\ if\ move(B1, L), B <> B1, block(L).$
$nonexecutable\ move(B, L)\ if\ move(B, L1), L <> L1.$
$nonexecutable\ move(B, B1)\ if\ move(B1, L).$
$caused\ on(B, L)\ after\ move(B, L).$
$caused\ \neg on(B, L)\ after\ move(B, L1),\ on(B, L),\ L <> L1.$
$inertial\ on(B, L).$
$caused\ blocked(B)\ if\ on(_, B).$

$initially:$ $on(1, 2).\ on(2, table).\ on(3, 4).\ on(4, table).\ on(5, 6).\ on(6.table)$

$goal:$ $on(1, 3),\ on(3, table),\ on(2, 4),\ on(4, table),\ on(6, 5),\ on(5, table)?(3)$

Actions may have assigned costs, which should be minimized. In our case, each `move` has cost 1, resulting in plans, where a minimum number of moves is executed to achieve the plan. The rules following the declarations of actions and fluents describe the transitions and constraints on the initial states of the domain. Finally, the `goal:` section defines the goal to be reached, which is a conjunction of ground fluent literals left of "?", and the plan length which is given in parentheses right of "?" (in this case, 3).

A detailed account of the language $\mathcal{K}^c$ is given in [39]. The planning front-end, described in [37], is available, ready for experiments, at `http://www.dlvsystem.com/K/`.

5.4 SQL3 Front-End

The front-end for SQL3 was the first front-end developed for DLV. At the time when it was realized, no system fully supported the SQL3 query language, and this front-end allowed the users to play with some of the novel features of SQL3.

The SQL3 front-end translates a subset of SQL3 query expressions to DLV queries and supports recursive query processing such as the "list of materials" problem, where we have items that consist of other items that again consist of other items and so forth, and have to transform this tree structure into a list of all dependencies.

Since there are no column names in datalog, we introduced the construct

```
DATALOG SCHEMA Relationname(Columnname, ...);
```

which creates a connection between the parameters of a datalog predicate and the column names of an SQL table.

Example 5.4 (List of Materials) Consider the classical "list of materials" query:

```
DATALOG SCHEMA consists_of(major,minor);

WITH RECURSIVE listofmaterials(major,minor) AS
     (
     SELECT c.major, c.minor FROM consists_of AS c
     UNION
     SELECT c1.major, c2.minor
       FROM consists_of AS c1, listofmaterials AS c2
       WHERE c1.minor = c2.major
     )
SELECT major, minor FROM listofmaterials;
```

This query is translated into

$$listofmaterials(A, B) \text{ :- } consists_of(A, B).$$
$$listofmaterials(A, B) \text{ :- } consists_of(A, C), listofmaterials(C, B).$$
$$sql2dl_intern0(A, B) \text{ :- } listofmaterials(A, B).$$

and the elements of $sql2dl_intern0$ are printed in tabular form as the result.

5.5 Meta-Interpreter Front-End

In [34, 35] we have presented a kind of meta-interpretation technique for prioritized logic programs, which can be used to compute preferred and weakly preferred answer sets as defined in [11]. In this setting, a special program – the meta-interpreter – is used in conjunction with a suitably transformed prioritized program, which forms a fact base. The answer sets of the meta-interpreter and the transformed input then correspond to the preferred answer sets of the original prioritized program.

Note that this is a very lightweight kind of front-end, as the initial transformation is quite simple and the meta-interpreter is the same for any input. The output transformation is constituted by a simple filter.

In [35] we have elaborated on this approach and have provided additional meta-interpreters for variants of preferred answer set semantics, based on a unifying characterization defined in [94]. All these meta-interpreters, several examples, and further information are available at `http://www.dlvsystem.com/preferred/`.

5.6 External Front-Ends

Besides the front-ends above, which are included in the standard distribution of DLV, further front-ends have been created, some of them by other researchers.

The most notable of these front-ends, the *plp* system, has been described in [23] and is available on its web-page at `http://www.cs.uni-potsdam.de/~torsten/plp/`. It can use DLV or Smodels as a back-end for computing a particular notion of preferred answer sets.

A front-end for update logic programs, which are related to inheritance logic programs described above, is described in [41]. Update logic programs are sequences $\mathbf{P} = (P_0, \ldots, P_n)$ of extended logic programs P_i, where each P_i, $i > 0$, represents an update of the initial program P_0, which arrive at time points $i = 1, 2, \ldots, n$. The answer set semantics of extended logic programs is extended in [42] to such update programs. The front-end, which supports different reasoning tasks, is available at `http://www.kr.tuwien.ac.at/staff/giuliana/project.html`. It has been used to realize policies for updating knowledge bases [40].

Another external front-end has recently been implemented. It is called *nlp* and transforms logic programs with nested expressions to Answer Set Programs. DLV is then used as the kernel to compute answer sets, which are post-processed to yield the answer sets of the original program with nested expressions. A description of this front-end can be found in [83] and on its web-page `http://www.cs.uni-potsdam.de/~torsten/nlp/`, from which it can be downloaded.

6 The Implementation of the DLV System: An Overview

The high expressiveness of the DLV language together with the ambition to deal efficiently also with simpler problems on larger problem instances, made the implementation of DLV system a challenging task.

The DLV core has three layers (see Figure 3), each of which is a powerful subsystem per se: The *Intelligent Grounder* (also *Instantiator*) (IG) 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. These layers have been improved over the years through the implementation of sophisticated data structures and of advanced optimization techniques. It is infeasible to give a detailed description of the implementation of DLV in this paper. Instead, we provide a description of the general architecture of DLV, we illustrate the inspiring principles underlying the DLV system, and we give an overview of the main techniques which were employed in the implementation, pointing the reader to specific papers that provide the full details.

The system architecture of DLV is shown in Figure 3. The internal system language is the one described in Section 2, and the DLV Core (the shaded part of the figure) is an efficient engine for computing answer sets (one, some, or all) of its input. In addition to various front-ends (described in Section 5), there is a Graphical User Interface (GUI) that provides convenient access to some of these front-ends as well as the system itself.

The implementation of the DLV system is based on very solid theoretical foundations, and exploits the results on the computational complexity of DLV language fragments discussed in Section 4. 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 4. Note that in general, without syntactic restrictions, it is impossible to detect whether a program uniformly encodes an "easy" problem, since this task is clearly undecidable.

For example, stratified normal programs (which have polynomial complexity, as reported in Table 1) are evaluated just using techniques from the field of deductive databases, without employing at all the more complex techniques which are needed to evaluate full DLV programs (such normal stratified programs are evaluated without generating the program instantiation at all). The SQL3 front-end, which only generates such programs, and datalog programs encoding deductive problems like Reachability or Same Generation (cf. Sections 3.1.1 and 3.1.2) benefit from this and exhibit good performance.

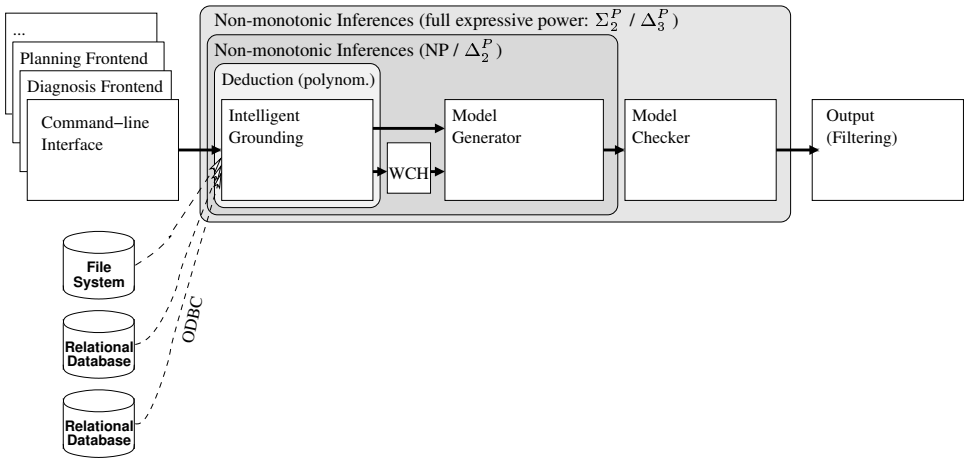


Figure 3: The System Architecture of DLV

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 (e.g., normal stratified programs are completely solved by the IG); the Model Generator (together with the Grounding) is capable of solving NP-complete problems. Adding the Model Checker is needed to solve Σ_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 4, 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 have polynomial complexity. They are evaluated [11] solely 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) without any call to the MC layer. 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 Δ_2^P. Here, also the WCH module is employed, which calls the MG iteratively; however, no calls to the MC module are made.

- 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 Σ_2^P, and the programs are evaluated by the MG module (besides the IG) **with** calls to the MC module. Note that a flat backtracking algorithm is not sufficient to evaluate Σ_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 Δ_3^P.

[11] In this section, for *evaluation* we mean the process of generating an answer set of the input program; brave reasoning corresponds to this task.

The evaluation proceeds as for L_4, but also the WCH module is employed for handling the weak constraints.

Thus, the three DLV modules, MG, MC, and WCH, deal with the three sources of complexity denoted by (s_1) (s_2) and (s_3) in Section 4, respectively; each of them is activated *only if* the respective source of complexity is present in the program at hand.

Let us look now at the evaluation flow of the DLV computation. 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 in a very efficient way (linear memory and time). The input is usually read from text files, but DLV provides also a bridge to relational databases, which allows, through an ODBC interface, for retrieving facts stored in relational database tables and for storing the result of a DLV computation into an external database.

Using differential and other advanced database techniques (see [50, 68]) together with suitable data structures, the *Intelligent Grounding* (IG) module then efficiently generates a ground instantiation, denoted $Ground(\mathcal{P})$, 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 performed by the Model Generator and the Model Checker. Roughly, the *Model Generator* (MG) produces some "candidate" answer sets. The stability of each of them is subsequently verified by the Model Checker. 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 in the section.

The generation of the answer sets of a program $\mathcal{P}$ relies on a monotonic operator $\mathcal{W}_\mathcal{P}$ [69] which extends the well-founded operator of [99] for normal programs to disjunctive programs. It is defined in terms of a suitable notion of *unfounded set*. Intuitively, an unfounded set for a disjunctive program $\mathcal{P}$ w.r.t. an interpretation I is a set of positive literals that definitely cannot be derived from $\mathcal{P}$ assuming the facts in I [69].

Briefly, the algorithm works as follows: First, $\mathcal{W}_\mathcal{P}^\omega(\emptyset)$ (the fixpoint of $\mathcal{W}_\mathcal{P}$) is computed, which is contained in every answer set. If $\mathcal{W}_\mathcal{P}^\omega(\emptyset)$ is a total model, it is returned as the (unique) answer set. Otherwise, moving from $\mathcal{W}_\mathcal{P}^\omega(\emptyset)$ towards the answer sets, a literal (called *possibly-true literal* in [69]), the truth of which allows to infer new atoms, is assumed true. Clearly the choice of "good" possibly-true literals at each step (i.e., a sequence of possibly-true literals that quickly leads to an answer set) is crucial for an efficient computation, so we employ novel heuristics with extensive lookahead and also propagate knowledge about choices that lead to inconsistency. We refer to [51] for details on the heuristics.

The computation of the MG proceeds by iteratively applying a pruning operator followed by the selection of a possibly-true literal, until either a total model of $Ground(\mathcal{P})$ is reached or two contradictory literals are derived. If a model is found, then the Model Checker is called; otherwise, 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 Model Checker disregards possible 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 possible 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. [45]). However, there are some relevant and frequently used classes of programs where answer-set checking can be efficiently performed (see Table 3 in Section 4).

The MC implements novel techniques for answer-set checking [65], which extend and complement previous results [8, 9, 69]. The MC fully complies with the complexity bounds specified in Section 4. Indeed, (a) it terminates in polynomial time on every program where answer-set checking is tractable according to Table 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 modules that are not HCF. Note that it may well happen that only a very small part of the program is not HCF [65].

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 a sense, also the language described in Section 2 is implemented by means of a front-end, even though by a very "thin" one.

In presence of weak constraints, after the instantiation of the program, the computation is governed by the WCH. The evaluation of general DLV programs (**with** weak constraints) consists of two phases: (i) the first phase determines the cost of an optimal answer set[12], 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 weak constraints do not affect answer-set checking at all.

7 Experiments and Benchmarks

7.1 Overview of Compared Systems

The goal of the DLV project is to provide an efficient implementation of Disjunctive Logic Programming. Thus, the main purpose of the experiments is to assess the efficiency of DLV as a DLP system. Unfortunately, most DLP systems in the literature (cf. Introduction) have been built for experimental purposes only and are not much concerned with efficiency; there are almost no elaborated, robust implementations of DLP, which would be natural candidates to evaluate the performance of DLV. Only the GnT system [62] (described below) is an efficient and robust implementation of DLP under the answer set semantics, which we thus selected for an experimental comparison with DLV.

Furthermore, in order to elucidate the performance of DLV in comparison with more specialized systems, we have also considered some efficient ASP engines which are close relatives of DLV, but do not support disjunction. The comparison of DLV with these systems is of other interest than its comparison with DLP systems: it should provide an idea how disjunctive systems like DLV and GnT compare (on natural disjunctive encodings) against non-disjunctive systems which use unstratified negation. It is worthwhile noting, however, that non-disjunctive ASP systems have lower expressiveness than DLV. In particular, on benchmark problems in NP, DLV (and GnT as well) has an overhead with respect to these systems since it is designed as a solver for a larger class containing much harder problems, while the non-disjunctive systems are tailored for a special fragment in this class which is, in general, not efficiently recognizable in a syntactic way.

Recently, the scientific community has made considerable efforts on the implementation of answer set programming, and a number of systems nowadays support answer set programming to some extent [1, 4, 5,

[12] 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 2.2.

18, 20, 19, 29, 28, 30, 31, 46, 62, 72, 75, 79, 90, 95, 97]. Most of these systems have common roots, but differ in several respects, especially in the languages they support.

For a comparison with DLP systems, we have considered ASP systems whose languages are as close to Disjunctive Logic Programming (and to the DLV language) as possible, since we feel that it is difficult to make a fair comparison among systems supporting very different languages. We have thus focused on systems supporting the full language of function-free logic programs under the answer set semantics, including recursion, and unstratified negation. Among those, we preferred more widely used systems which are at a more advanced engineering stage; several other ASP systems are research prototypes implemented just for experimental purposes, and would need quite some engineering work to be made efficient and robust.

In particular, in addition to GnT, we picked two further ASP systems, namely, *Smodels* [97, 80, 79] – one of the most robust and widely used ASP systems – and *ASSAT* [72], which is not as widely used and robust as Smodels yet but appeared to be very efficient in a couple of recent experimental comparisons [97, 72].[13] In the rest of this section, we provide a short description of the three systems we considered for benchmarking:

Smodels [97, 80, 79] is one of the best known and most widely used Answer Set Programming systems. Smodels implements the answer set semantics for normal logic programs extended by built-in functions as well as cardinality and weight constraints for domain-restricted programs.

Disregarding the extension for cardinality and weight constraints, the Smodels system takes as input normal ($\mathtt{v}$ -free) rules as described in Section 2. Programs must be *domain-restricted*, which intuitively amounts to the following property: the predicate symbols in the program are divided into two classes, namely *domain predicates* and *non-domain* predicates, where the former are predicates whose definition[14] does not involve rules with negative recursion [98]. Every rule must be domain-restricted in the sense that every variable in a rule must appear in a domain predicate which appears positively in the rule body.

In addition to normal rules, Smodels supports rules with cardinality and weight constraints. To illustrate the idea on an example, the cardinality constraint

$$1\{a, b, notc\}2.$$

holds in an answer set if at least 1 but at most 2 of the literals in the constraint are satisfied by that answer set.

The instantiation module of Smodels is a separate application called *Lparse*, which preprocesses the programs which are then evaluated by Smodels.

In our tests, we have used the current versions of these two at the time of this writing, namely Lparse 1.0.9 and Smodels 2.26, running them in a UNIX pipe and measuring overall CPU time.

GnT [62] is a disjunctive logic programming system that extends the Smodels language by allowing disjunction in the head of the rules, which is written as $\{a_1, \cdots, a_n\}$. GnT has been implemented on top of Smodels, by means of a rewriting technique. Roughly, a disjunctive input program $\mathcal{P}$ is rewritten to a non-disjunctive program $\mathcal{P}'$, such that the answer sets of $\mathcal{P}'$ form a superset of the answer sets of $\mathcal{P}$. Program $\mathcal{P}'$

[13] It is worthwhile noting that a novel system named Cmodels [5] has recently emerged as a very efficient ASP solver. The Cmodels system is similar in spirit to ASSAT, in that it translates the logic program to a CNF theory and uses a SAT-solver for its evaluation. We did not consider this system in our comparisons, because it requires the input program to be *acyclic*, and is not applicable on many programs we used for benchmarks in this paper.

[14] The definition of a predicate p contains the rules with head p plus the definitions of all predicates appearing in the bodies of the rules with head p.

is passed to Smodels for the evaluation, and each answer set M, produced in this phase, is then processed by a nested call to Smodels itself, to check whether M is also an answer set of $\mathcal{P}$ or not.

GnT is not a *strict* generalization of Smodels in that it does not support the full language of Smodels (in particular, it does not support cardinality constraints).

ASSAT The ASSAT system (Answer Sets by SAT solvers) is a system for computing answer sets of a logic program by using SAT solvers [72]. Given a ground logic program $\mathcal{P}$ and a SAT solver X, ASSAT(X) works as follows:

- It computes the Clark-completion [21] of $\mathcal{P}$ and converts it into a set C of clauses.
- It then repeats the following steps:
  - Call X on C to get a model M of $\mathcal{P}$, and terminate with failure if no such M exists.
  - If M is an answer set of $\mathcal{P}$, then return it as the result.
  - Otherwise, find some loops in $\mathcal{P}$ whose "loop formulas" (defined in [72]) are not satisfied by M, and add their clausal forms to C. This exploits the result that a model M is not an answer set of $\mathcal{P}$ if and only if some loop formula is not satisfied [72].

As shown in [72], this procedure is sound and complete, assuming that X is a sound and complete SAT solver. It should be noted that this approach is geared towards computing *one* answer set, rather than computing *all* (or a given number of) answer sets. While in an incremental computation, already computed answer sets can be easily excluded from re-computation by adding suitable clauses, the efficiency of this approach remains to be seen.

Like Smodels, ASSAT exploits *Lparse* for program instantiation. ASSAT accepts the core language of Smodels without cardinality and weight constraints, and imposes the domain restriction constraint on the variables required by Lparse.

For the benchmarks, we have used ASSAT version 1.0 [101], together with Lparse 1.0.9 and the SAT solver Chaff version 2001-06-16 (`http://www.ee.princeton.edu/~chaff/index1.html`). (By using Chaff, ASSAT provided the best performance in the benchmarks described by the authors of ASSAT [72].)

We have used ASSAT in its default (built-in) configuration and Chaff with the default configuration file "cherry.smj," which comes with the Chaff distribution.

7.2 Benchmark Problems and Data

We had to ponder about the construction of a suite of benchmark problems to test DLP systems, as unlike other areas, such as Satisfiability Testing, there is currently no generally recognized set of benchmark problems for DLP systems. We believe that a benchmark suite should be ample, cover different application areas, and comprise problems of different complexities, in order to highlight the suitability of the systems for different applications, and on different problem (complexity) classes.

We have studied previous experimentation work performed for benchmarking ASP systems at Helsinki University of Technology, and exploited also the precious discussions on this topic at *LPNMR'01*, the *AAAI Spring 2001 Symposium on Answer-Set Programming*, and the *Dagstuhl Seminar on Answer Set Programming* (September 2002).

We have considered a number of benchmark problems taken from various domains and with very different complexities ranging from P over NP, co-NP, and Δ_2^P to Σ_2^P. We have included most examples provided in Section 3 as well as a couple of further problems with suitable sets of benchmark data. In particular, we have evaluated the systems on the following problems.

- Reachability (REACH)
- Same Generation (SAMEGEN)
- Hamiltonian Path (HAMPATH)
- Traveling Salesperson (TSP)
- Ramsey Numbers (RAMSEY)
- Sokoban (SOKOBAN)
- Strategic Companies (STRATCOMP)
- Quantified Boolean Formulas (QBF)

It is worthwhile noting that all problems above except TSP belong to the suite of "Benchmark Problems for Answer Set Programming Systems" of the University of Kentucky (`http://cs.engr.uky.edu/ai/benchmarks.html`), which has been used to compare ASP systems at the *6th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'01)* in Vienna, September 2001. Moreover, the RAMSEY problem was used to compare ASP systems at the *AAAI Spring 2001 Symposium on Answer Set Programming* in Stanford, California, March 2001, and the problems SAMEGEN, HAMPATH, RAMSEY, and STRATCOMP also belong to the benchmark suite defined at the *Dagstuhl Seminar on Answer Set Programming*, September 2002 [12].

As far as the encodings are concerned, we did not look for tricky encodings; rather, we tried to exploit the expressiveness of the language to design simple and declarative encodings. Where possible, we have used the same encodings for all systems, keeping them as close as possible to the standard DLP language,[15] and only made minimal changes to allow for differences among the languages that DLV, GnT, Smodels, and ASSAT support. However, we have also considered further encodings which exploit specific features of Smodels to test how these affect performance. All benchmark instances and encodings we have used are available on the web at `http://www.dlvsystem.com/examples/bench-2002q3.zip`.

We next specify the data and the encodings which we have used for the experiments.

Reachability (REACH) For DLV we used the encoding presented in Section 3.1.1; for GnT, Smodels,[16] and ASSAT we had to make some modifications to respect the *domain restriction* constraint required by Lparse:

$$reachable(X,Y) \text{ :- } arc(X,Y).$$
$$reachable(X,Y) \text{ :- } arc(X,U), reachable(U,Y), vertex(Y).$$

In particular, we had to add atom $vertex(Y)$ to the body of the second rule, in order to restrict the domain of the variable Y.

The input graphs for Reachability were generated via the Stanford GraphBase [64], using the function *random_graph(#nodes, #arcs, 0,0,0,0,0,0,0,0)* with a ratio of 3:1 between #arcs and #nodes.

[15] The language adopted in [55] is generally acknowledged in the literature, and most systems support it to a large extent.

[16] Note that GnT behaves like Smodels if the input program can be solved by the instantiator.

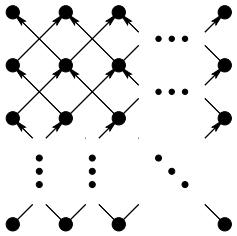


Figure 4: Input for SAMEGEN

Same Generation (SAMEGEN) For this problem, we have used the encoding described in Section 3.1.2 and instances where the parent relation is a square board as depicted in Figure 4.

We did not need to adapt the encodings for GnT, Smodels, and ASSAT, as the encoding from Section 3.1.2 already satisfies the domain restriction constraint of Lparse.

Ramsey Numbers (RAMSEY) In our experiments, we have considered the problem of deciding, for varying k, m, and n, if n is the Ramsey number $R(k, m)$. For DLV, we have used the encoding shown in Section 3.3.5.

It is worthwhile noting that for this problem we do not use a single, uniform encoding to solve all instances, rather, we have one program for each instance. In particular, for checking that n is the Ramsey number $R(k, m)$, the first constraint contains $\binom{k}{2}$ atoms with predicate red and the second constraint contains $\binom{m}{2}$ atoms with predicate $blue$.

For Smodels and ASSAT we have replaced the disjunctive rule by two normal rules using unstratified negation. We have also added some instances of $node(\dots)$ to the integrity constraints, to meet the syntactical restrictions of Lparse:

$$
\begin{aligned}
&blue(X,Y) \text{ :- } arc(X,Y), \text{not } red(X,Y).\\
&red(X,Y) \text{ :- } arc(X,Y), \text{not } blue(X,Y).\\
&\text{:- } blue(X,Y), blue(Y,Z), blue(X,Z),\\
&\quad node(X), node(Y), node(Z).\\
&\text{:- } red(W,X), red(W,Y), red(W,Z), red(X,Y), red(Y,Z), red(X,Z),\\
&\quad node(X), node(Y), node(Z), node(W).
\end{aligned}
$$

For these systems, we have also considered an alternative encoding where the domain restriction constraint is satisfied by adding an atom of the form $arc(t_1, t_2)$, for each atom $blue(t_1, t_2)$ and each atom $red(t_1, t_2)$ occurring in the constraint. This encoding seems to have a positive impact on the performance (see Section 7.3) due to the instantiation procedure employed by Lparse. However, we believe that this encoding is less intuitive, as a domain predicate is understood as the "type" of a variable; accordingly, $node(X)$ would be the natural type for each variable X occurring in this program.

For GnT, we have used the disjunctive encoding from Section 3.3.5 with the addition of the above domain predicates to satisfy the domain restriction constraint.

Finally, we have also considered an alternative encoding for Smodels where we replaced the first two guessing rules by a single rule using Smodels' enhanced syntax:

$$1\{blue(X,Y), red(X,Y)\}1 \text{ :- } arc(X,Y).$$

Hamiltonian Path (HAMPATH) For DLV, we have used the encoding described in Section 3.3.2. For Smodels and ASSAT we rewrote the disjunctive guessing rule to use unstratified negation instead, and added some instances of $arc(\ldots)$ to the integrity constraints to work around the *domain restriction* constraint required by Lparse:[17]

$$
\begin{array}{l}
inPath(X,Y) \text{ :- not } outPath(X,Y), start(X), arc(X,Y).\\
outPath(X,Y) \text{ :- not } inPath(X,Y), start(X), arc(X,Y).\\
inPath(X,Y) \text{ :- not } outPath(X,Y), reached(X), arc(X,Y).\\
outPath(X,Y) \text{ :- not } inPath(X,Y), reached(X), arc(X,Y).\\
\\
\text{:- } inPath(X,Y), inPath(X,Y1), Y <> Y1, arc(X,Y), arc(X,Y1).\\
\text{:- } inPath(X,Y), inPath(X1,Y), X <> X1, arc(X,Y), arc(X1,Y).\\
\text{:- } vertex(X), \text{not } reached(X), \text{not } start(X)\\
\\
reached(X) \text{ :- } inPath(Y,X), arc(Y,X).
\end{array}
$$

For GnT we have used the (disjunctive) encoding of DLV, with the addition of the domain predicates in the integrity constraints as above.

Again, we have also considered an alternate encoding for Smodels, using the extended language features of Smodels (which are currently not supported by GnT and ASSAT):

$$
\begin{array}{l}
\{inPath(X,Y)\} \text{ :- } start(X), arc(X,Y).\\
\{inPath(X,Y)\} \text{ :- } reached(X), arc(X,Y).\\
\\
\text{:- } 2\{inPath(X,Y) : arc(X,Y)\}, vertex(Y).\\
\text{:- } 2\{inPath(X,Y) : arc(X,Y)\}, vertex(X).\\
\text{:- } vertex(X), \text{not } reached(X), \text{not } start(X).\\
\\
reached(X) \text{ :- } inPath(Y,X), arc(Y,X).
\end{array}
$$

The graph instances were generated using a tool by Patrik Simons (`http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`) which originally was used to compare Smodels against SAT solvers [96]. This tool generates graphs with nodes labeled 0, 1, and so forth, and we assume $node(0)$ as the starting node.

Traveling Salesperson (TSP) Here we have reused the graph instances for HAMPATH described above and randomly added costs between 1 and 10 to every arc. The uniform approach described in Section 3.3.3 for encoding TSP in DLV is not feasible using the language of Smodels, since the direct counterpart of the weak constraint of DLV in Smodels (the $minimize$ statement) does not support the use of variables. Thus, we employed a more direct, instance-specific encoding in the form of

$$
\begin{array}{l}
minimize\ [\ inPath(0,3) = 9,\ inPath(0,2) = 4,\ \ldots,\ inPath(3,1) = 7\ ].\\
compute\ all\ \{\}.
\end{array}
$$

[17] We have used the domain predicate arc instead of $node$, since Lparse seems to produce a more advantageous instantiation this way; cf. the discussion on the encodings of RAMSEY.

together with the following fixed program

$$
\begin{array}{l}
inPath(X,Y) \text{ :- not } outPath(X,Y), start(X), arc(X,Y).\\
outPath(X,Y) \text{ :- not } inPath(X,Y), start(X), arc(X,Y).\\
inPath(X,Y) \text{ :- not } outPath(X,Y), reached(X), arc(X,Y).\\
outPath(X,Y) \text{ :- not } inPath(X,Y), reached(X), arc(X,Y).\\
\\
\text{:- } inPath(X,Y), inPath(X,Y1), Y <> Y1, arc(X,Y), arc(X,Y1).\\
\text{:- } inPath(X,Y), inPath(X1,Y), X <> X1, arc(X,Y), arc(X1,Y).\\
\text{:- } node(X), \text{not } reached(X).\\
\\
reached(X) \text{ :- } inPath(Y,X), arc(Y,X).
\end{array}
$$

for Smodels, and

$$
\begin{array}{l}
:\sim inPath(0,3).[9:]\\
:\sim inPath(0,2).[4:]\\
\vdots\\
:\sim inPath(3,1).[7:]
\end{array}
$$

together with the following fixed program

$$
\begin{array}{l}
inPath(X,Y) \text{ v } outPath(X,Y) \text{ :- } start(X), arc(X,Y).\\
inPath(X,Y) \text{ v } outPath(X,Y) \text{ :- } reached(X), arc(X,Y).\\
\\
\text{:- } inPath(X,Y), inPath(X,Y1), Y <> Y1.\\
\text{:- } inPath(X,Y), inPath(X1,Y), X <> X1.\\
\text{:- } node(X), \text{not } reached(X).\\
\\
reached(X) \text{ :- } inPath(_,X).
\end{array}
$$

for DLV. In addition, we have also experimented with an alternate encoding using the advanced language features of Smodels:

$$
\begin{array}{l}
1\{inPath(X,Y) : arc(X,Y)\}1 \text{ :- } vertex(X).\\
1\{inPath(X,Y) : arc(X,Y)\}1 \text{ :- } vertex(Y).\\
\\
\text{:- } vertex(X), \text{not } reached(X).\\
\\
reached(Y) \text{ :- } start(X), inPath(X,Y), arc(X,Y).\\
reached(Y) \text{ :- } reached(X), inPath(X,Y), arc(X,Y).
\end{array}
$$

Note that the *compute all* statement for Smodels only leads to a single model of least cost, while during the computation Smodels also prints non-optimal models. To the best of our knowledge, Smodels currently is not capable of generating precisely the set of models having least cost (in case there is more than one such model). Such models can only be obtained by running Smodels a second time with the original program extended by a further constraint discarding the models whose costs are higher than the least cost (computed in the first run).

ASSAT and GnT currently do not support the *minimize* statement of Lparse/Smodels (and they do not support weak constraints either), so we could not test them on this problem.

Sokoban (SOKOBAN) This is a game puzzle which has been 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). The goal is to move all boxes to storage locations. To this end, the sokoban can walk on floor locations (unless occupied by a box), and push single boxes to unoccupied floor locations. Figure 5 shows a typical configuration involving two boxes, where grey fields are storage fields and black fields are walls.

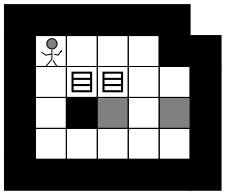


Figure 5: A Sokoban Instance

We have written programs for DLV, GnT, Smodels and ASSAT (see Appendix A) and also experimented with an efficient encoding by Tommi Syrjänen which is available at `http://www.tcs.hut.fi/~tssyrjan/software/`. These programs find solutions with a given number of push actions (where one push action can move a box over any number of fields) for a given puzzle, and we have developed scripts which iteratively run these programs with increasing numbers of push actions (starting at one) until some solution is found. In this way solutions with a minimal number of push actions are computed.

The puzzle in Figure 5 is solvable with six push actions, so each of the scripts uses the corresponding ASP system to first prove that no solutions with one to five push actions exist, and then to compute a solution with six push actions.

We have also tried to evaluate ASSAT using the same encodings as for Smodels, but due to bugs in the version available at the time of this writing (ASSAT is quite new, and thus did not yet undergo as extensive testing and use as Smodels and DLV), ASSAT either crashed or generated incorrect results, so we had to exclude it from this set of benchmarks. Still, the approach and the performance characteristics of ASSAT seem very promising, so we include it in our benchmarks where possible.

Strategic Companies (STRATCOMP) Here, we have generated tests with instances for n companies ($5 \leq n \leq 170$), $3n$ products, 10 uniform randomly chosen $contr_by$ relations per company, and uniform randomly chosen $prod_by$ relations.

To make the problem harder, we have considered (up to) four producers per product, (up to) four controlling companies per company and only strategic sets containing two fixed companies (1 and 2, without loss of generality), slightly adjusting the program $\mathcal{P}_{strat}$ from Section 3.3.6 as follows:

$$
\begin{aligned}
&strat(X1) \mathtt{v}\, strat(X2) \mathtt{v}\, strat(X3) \mathtt{v}\, strat(X4) \mathtt{:-}\\
&\qquad prod_by(X, X1, X2, X3, X4).\\
&strat(W) \mathtt{:-}\, contr_by(W, X1, X2, X3, X4),\\
&\qquad strat(X1), strat(X2), strat(X3), strat(X4).\\
&\mathtt{:-}\, \mathrm{not}\ strat(1).\\
&\mathtt{:-}\, \mathrm{not}\ strat(2).
\end{aligned}
$$

We have used the same encoding also for GnT (just rewriting the disjunction according to GnT syntax, as above).

We could not consider ASSAT and Smodels on this problem, since they do not support disjunction, which is strictly needed to solve this problem (unless P = NP).

Quantified Boolean Formulas (QBF) Our second benchmark problem residing on the second level of the Polynomial Hierarchy is 2QBF, which is well known to be Σ_2^P-complete [82]. The problem here is to decide whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, is valid.

The transformation from 2QBF to disjunctive logic programming is a variant of a reduction used in [43], where we clearly separate problem instances and the following general encoding $\mathcal{P}_{QBF}$:

$$
\begin{array}{rcl}
t(X) \,\mathtt{v}\, f(X) & \text{:-} & exists(X). \\
t(Y) \,\mathtt{v}\, f(Y) & \text{:-} & forall(Y). \\
w & \text{:-} & conjunct(X, Y, Z, Na, Nb, Nc), \\
 & & t(X), t(Y), t(Z), f(Na), f(Nb), f(Nc). \\
t(X) & \text{:-} & w, forall(X). \\
f(X) & \text{:-} & w, forall(X). \\
 & \text{:-} & \text{not } w. \qquad t(true). \qquad f(false).
\end{array}
$$

A 2QBF instance Φ as above is encoded by the following set F_Φ of facts:

- $exists(v)$, for each existential variable $v \in X$;
- $forall(v)$, for each universal variable $v \in Y$; and
- $conjunct(p_1, p_2, p_3, q_1, q_2, q_3)$, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom v_i, then $p_i = v_i$, otherwise p_i= "$true$", and (ii) if l_i is a negated atom $\neg v_i$, then $q_i = v_i$, otherwise q_i="$false$". For example, $conjunct(x_1, true, y_4, false, y_2, false)$, encodes the term $x_1 \wedge \neg y_2 \wedge y_4$.

Φ is a yes-instance, i.e., it is valid, if and only if $\mathcal{P}_{QBF} \cup F_\Phi$ has some answer set.

The encoding for GnT is the same as for DLV, just using the different syntax for disjunction.

Since this problem is Σ_2^P-complete, as with STRATCOMP we could not consider ASSAT and Smodels, which lack disjunction.

We generated two different kinds of data sets following recent works presented in the literature. Each data set was randomly generated. In both cases the number of $\forall$-variables is equal to the number of $\exists$-variables (that is, $|X| = |Y|$) and each conjunct contains at least two universal variables.[18] In the first case, the number of clauses equals the overall number of variables (that is, $|X| + |Y|$); in the second case, suggested by Gent and Walsh [57], the number of clauses is $((|X| + |Y|)/2)^{0.5}$. In the following, we will refer to instances generated according to the first schema simply as QBF, those generated according to the second schema as QBF$_{GW}$.

[18] In conjunction with the second variable ratio, this constitutes the so-called "Model A" whose hardness has been experimentally evaluated in [57].

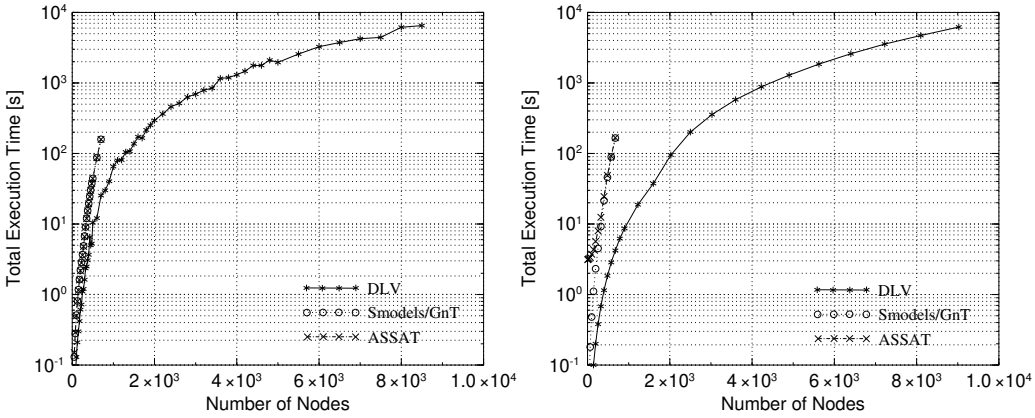
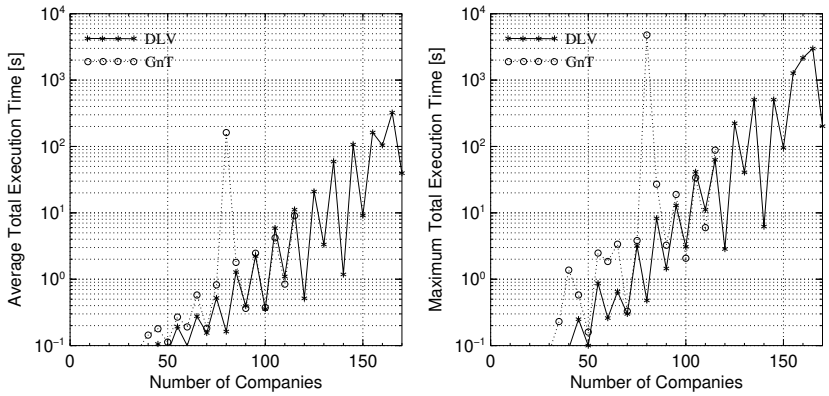


Figure 6: Reachability (REACH, left) and Same Generation (SAMEGEN, right)



Average time Maximum time

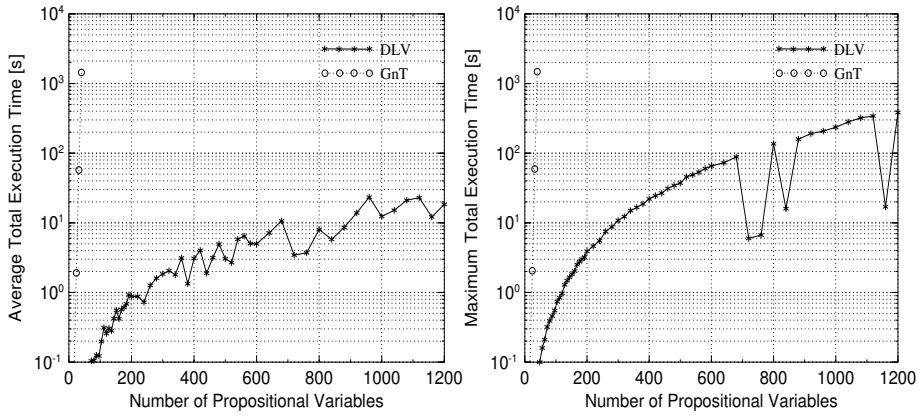
Figure 7: Strategic Companies (STRATCOMP)

7.3 Results and Discussion

For HAMPATH, TSP, STRATCOMP, QBF, and QBF$_{GW}$ we have generated 50 random instances for each problem size as indicated in the respective descriptions. For the more "deterministic" benchmarks REACH and SAMEGEN, we have used one instance per problem size.

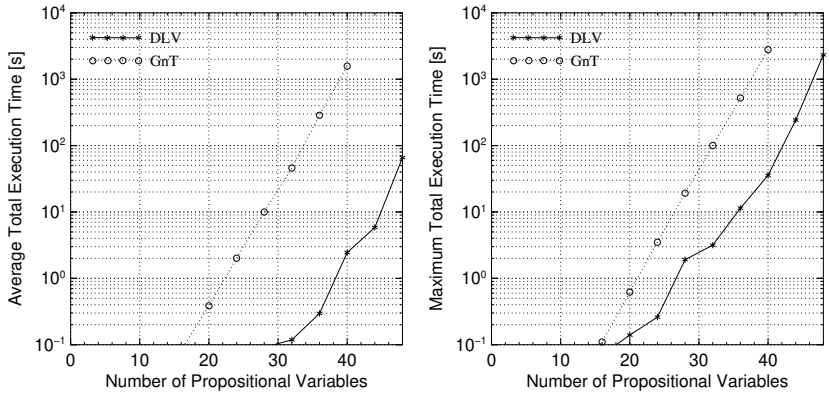
For every instance, we have allowed a maximum running time of 7200 seconds (two hours) and a maximum memory usage of 256MB (using the command "limit datasize 256mega"). In the graphs displaying the benchmark results, the line of a system stops whenever some problem instance was not solved within these time and memory limits.

In Figure 6, we show the results obtained for the problems REACH (left) and SAMEGEN (right). Figure 7 illustrates the results for STRATCOMP, while Figures 8 and 9 show the results for QBF$_{GW}$ and QBF, respectively. The results for HAMPATH and TSP are displayed in Figure 10 and in Figure 11, respectively. Table 4, finally, contains the results for SOKOBAN and RAMSEY. For the problems STRATCOMP, QBF, HAMPATH, and TSP, we report two graphs visualizing the average and the maximum computation time,



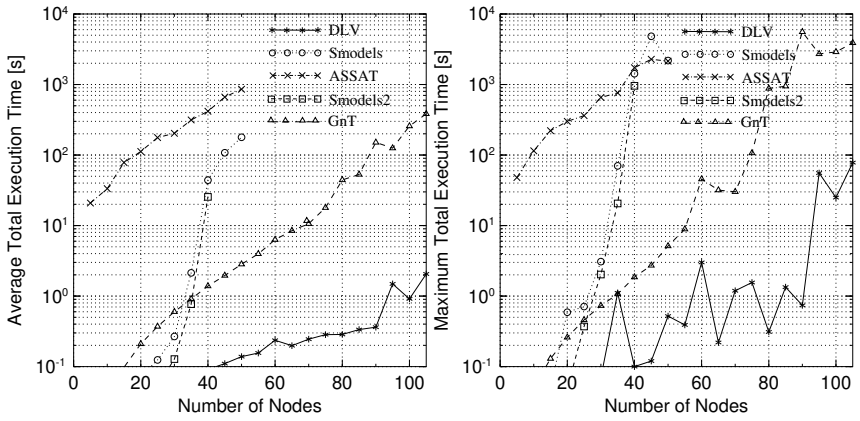
Average time Maximum time

Figure 8: QBF$_{GW}$



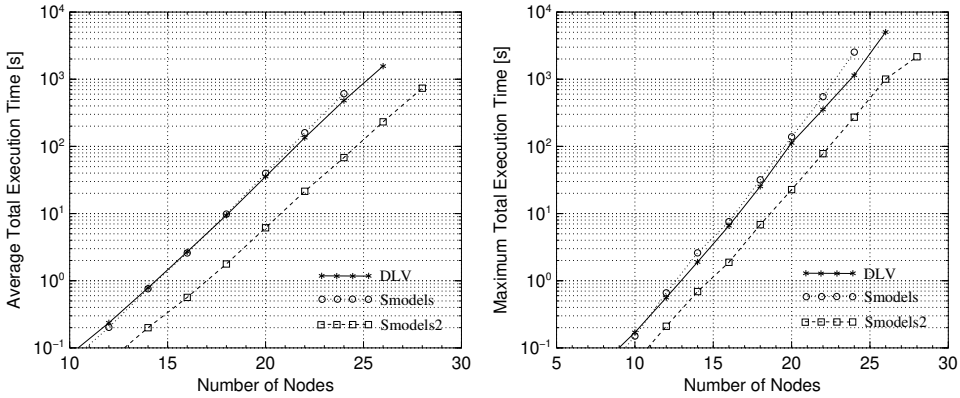
Average time Maximum time

Figure 9: QBF



Average time Maximum time

Figure 10: Hamiltonian Path (HAMPATH)



Average time Maximum time

Figure 11: Traveling Salesperson (TSP)

respectively, consumed by each system over the 50 instances of the same size. In the graphs and the tables, *Smodels* denotes the running time of the Smodels system when the standard LP encoding is used, while *Smodels2* denotes the running time when the specific features of Smodels (cardinality/weight constraints) are used.

On average, DLV outperforms the other systems on the set of proposed problems, but the results must be taken with some caveats, since they depend on the specific benchmark problems. We next discuss the results of the experiments, grouping together similar cases.

Deductive Database Applications

On the deductive database applications REACH and SAMEGEN, DLV is significantly faster than the other systems, which is due to the instantiators used. Indeed these problems can be solved completely by the instantiators of all considered systems. Since Smodels, GnT, and ASSAT adopt the same instantiator (Lparse), they show essentially the same performance on REACH and SAMEGEN. All of them exhaust at 450 nodes on REACH and on 676 nodes on SAMEGEN, respectively, while DLV goes much further, up to 8500 nodes on REACH and up to 9025 nodes on SAMEGEN. Such a relevant difference in system performance is explained by the fact that Lparse does not employ database optimization techniques, and its instantiation method requires each variable in the program to be bound over a pre-defined domain. The instantiator of DLV, instead, incorporates several database techniques, and the domains are built dynamically, which often leads to a smaller result [50, 68]; such a dynamic instantiation technique pays off in this case.

Σ_2^P Problems

On the smaller instances of STRATCOMP, DLV and GnT [19] behave similarly, but DLV scales much better and reaches 170 companies, while GnT stops at 115 companies. The systems come closer on QBF, with DLV still having a clear lead, but with the two systems scaling in a very similar way. In the case of QBF$_{GW}$ the difference between the two systems becomes significantly larger. The GnT system behaves similarly on QBF and QBF$_{GW}$ (it stops at size 40), while DLV goes much further on QBF$_{GW}$, where it is able to solve all instances up to size 1200.

The higher efficiency of DLV on Σ_2^P-complete problems is not surprising, as the treatment of disjunction in DLV is built-in, while in GnT disjunction is treated by a rewriting technique. Indeed, the performance of GnT is quite good considering that a rewriting technique is employed to implement disjunction.

HAMPATH and TSP

On HAMPATH, we have a picture of DLV and GnT similar to QBF, which shows better performance of DLV. The results for the non-disjunctive systems are more surprising, however, as we observe a clear difference in favor of the disjunctive systems: they solve all instances up to size 105, while ASSAT and Smodels stop at size 50.

Both Smodels and ASSAT are specialized for problems in NP, and we are aware that both are rather efficient on several NP-complete problems. The comparatively better performance of DLV on HAMPATH may be due to a bad behavior of the heuristics of ASSAT and Smodels on the specific class of the benchmarked problem instances. It is rather surprising that GnT, which is based on a rewriting to Smodels, is faster than

[19] Recall that we could not consider Smodels and ASSAT here, since they do support disjunction, which is strictly needed to solve Σ_2^P-complete problems like QBF and STRATCOMP.

Smodels itself on this problem; it seems that the encoding resulting from the translation of disjunction is dealt more efficiently with by the heuristics of Smodels in this case. We observe that Smodels behaves better than ASSAT on the small instances, and that the two systems scale similarly. The alternative encoding we tried for Smodels (Smodels2) does not make a real difference and actually performs slightly worse than the original one.

The picture on TSP is different.[20] DLV is slightly faster than Smodels if the standard LP encoding is used, while Smodels is fastest, if the alternative encoding (Smodels2) is used. In this case, the presence of cardinality constraints allows us to design a better encoding for Smodels, which cannot be efficiently simulated by the standard LP constructs in DLV. This test confirms that cardinality constraints, a peculiar feature of Smodels, are very useful and can have a positive impact on efficiency when they allow for a more succinct problem representation.

RAMSEY

RAMSEY is a nice example showing that the instantiation procedures can have a very strong impact on the performance of the systems also on hard problems. On the encodings of the first type, where *node* is used as domain predicate, DLV clearly outperforms all other systems, but this is mostly due to the respective instantiators. On RAMSEY, DLV solves all instances in the table,[21] while GnT, Smodels, and ASSAT solve about 40%, 30%, and 40% of the instances, respectively. The instantiator of Smodels, GnT, and ASSAT (Lparse) generates too many ground instances on this program and requires a huge amount of memory. For instance, to prove that RAMSEY3-5 is not 13, the Intelligent Grounding module of DLV consumes 5.8MB of memory and generates 1,743 ground rules (including constraints); on the same instance, Lparse requires 42.7MB of memory and generates 373,738 ground rules. Thus, the size of the instantiation generated by Lparse is two orders of magnitude higher than the instantiation generated by DLV, resulting in a huge overhead for Smodels and ASSAT. On larger instances (including all instances of RAMSEY3-6, RAMSEY3-7, and RAMSEY4-5), Lparse could not make the instantiation in the allowed amount of memory (256MB) and was halted. On the other hand, for the encoding with $edge$ as domain predicate, Lparse generates an instantiation of RAMSEY3-5 (13) with 1,821 ground rules (including facts) using only 1.3MB of memory.[22] Indeed, the performance of GnT, Smodels and ASSAT change completely on the encodings with $edge$, and the three systems perform significantly better than on the encoding with $node$. The performance of GnT comes closer to the performance of DLV, and ASSAT even performs slightly better on average (GnT and ASSAT can solve precisely the same instances in the allowed time and memory limits). It is worthwhile noting that also on this example GnT outperforms Smodels, and ASSAT, in turn, outperforms GnT.

SOKOBAN

For SOKOBAN, the situation is similar to RAMSEY, if the normal LP encoding is considered for GnT and Smodels. Again, the difference is mainly due to the instantiation modules of the systems. Lparse requires much more memory and generates a higher number of rule instances than DLV. For instance, on SOKOBAN #48, DLV requires 5.7MB of memory and generates 3,671 ground rules, while Lparse requires

[20] Recall that we could not test ASSAT and GnT on this problem, since they do not support optimization constructs.

[21] We do not report the results for larger RAMSEY instances, since no system was able to solve them in the allotted running time. (In fact, some of these instances are still well-known open problems.)

[22] The instantiator of DLV does not need the addition of the domain predicates and, if such domain predicates are added, their addition does not affect the size of the generated instantiation.

SOKOBAN:

Problem#	DLV	GnT	Smodels	Smodels2
1	10.26	-	-	4.13
2	0.25	-	-	0.84
3	0.31	-	-	0.94
4	0.23	53.68	47.48	0.69
5	0.45	-	-	1.25
6	0.19	33.40	29.48	0.62
7	0.36	81.24	72.01	0.76
8	0.61	-	-	1.39
9	0.16	-	96.05	0.47
10	0.27	-	-	0.89
11	0.25	-	149.31	0.66
12	0.26	-	150.30	0.74
13	0.19	12.80	11.24	0.41
14	0.28	33.46	29.60	0.62
15	0.26	-	-	0.80
16	1.41	-	-	2.66
17	0.24	-	105.48	0.74
18	0.97	-	-	1.61
19	0.76	-	-	1.94
20	0.96	-	-	1.36
21	0.40	-	151.30	1.00
22	0.40	-	215.10	0.89
23	0.42	-	215.20	1.13
24	0.39	-	151.87	0.99
25	0.16	10.93	9.68	0.58
26	0.54	-	206.36	1.39
27	0.12	12.80	11.30	0.44
28	0.46	-	-	1.25
29	0.74	-	184.53	1.62
30	0.53	-	-	1.45

Problem#	DLV	GnT	Smodels	Smodels2
31	0.70	-	-	1.76
32	0.12	12.89	11.39	0.43
33	0.44	-	-	1.26
34	0.66	-	-	1.43
35	1.24	-	-	2.25
36	0.33	76.66	68.14	0.90
37	0.86	-	-	1.46
38	0.62	-	205.90	1.35
39	1.09	-	-	1.49
40	0.39	-	151.07	0.97
41	0.48	-	140.06	1.25
42	0.46	103.40	92.48	1.13
43	0.95	-	-	1.53
44	0.54	-	204.99	1.06
45	0.69	-	183.23	1.49
46	0.92	-	336.76	2.04
47	1.14	-	-	2.42
48	1.07	-	336.43	2.19
49	0.57	-	-	0.97
50	612.49	-	-	100.27
51	175.77	-	-	233.78
52	-	-	-	-
53	0.75	-	267.96	1.48
54	11.52	-	-	6.93
55	237.31	-	-	32.40
56	2.84	-	-	1.95
57	-	-	-	-
58	-	-	-	-
59	18.12	-	-	26.94
60	5.43	-	-	11.43

RAMSEY:

Problem	Size	DLV	GnT (node)	GnT (edge)	Smodels (node)	Smodels2 (node)	Smodels (edge)	ASSAT (node)	ASSAT (edge)
RAMSEY3-3	5	0.01	0.02	0.00	0.01	0.02	0.01	0.03	0.02
RAMSEY3-3	6	0.01	0.02	0.00	0.01	0.02	0.01	0.03	0.03
RAMSEY3-4	8	0.02	0.26	0.02	0.24	0.24	0.02	0.20	0.03
RAMSEY3-4	9	0.88	1.25	0.78	0.95	1.01	0.53	1.58	1.65
RAMSEY3-5	12	0.19	23.53	0.19	20.60	20.64	0.16	19.02	0.18
RAMSEY3-5	13	0.32	35.36	1.74	30.91	30.93	0.27	35.76	0.27
RAMSEY3-6	16	3.50	-	57.63	-	-	5.98	-	5.04
RAMSEY3-6	17	6.35	-	17.49	-	-	11.10	-	7.84
RAMSEY3-7	15	3.98	-	54.76	-	-	52.53	-	51.76
RAMSEY3-7	16	7.40	-	96.43	-	-	90.14	-	88.31
RAMSEY3-7	17	13.47	-	164.60	-	-	149.41	-	145.47
RAMSEY3-7	18	24.17	-	283.07	-	-	-	-	232.91
RAMSEY3-7	19	42.56	-	491.34	-	-	-	-	363.39
RAMSEY3-7	20	68.64	-	763.21	-	-	602.80	-	555.35
RAMSEY3-7	21	111.95	-	1329.42	-	-	904.98	-	825.32
RAMSEY3-7	22	1513.00	-	-	-	-	-	-	1212.32
RAMSEY4-4	13	0.21	3.61	0.16	69.84	74.82	0.15	2.60	0.16
RAMSEY4-4	14	0.31	4.91	0.24	109.78	114.11	6.05	3.87	0.17
RAMSEY4-4	15	0.45	6.59	0.41	-	-	1.64	5.67	0.24
RAMSEY4-4	16	0.80	29.30	1.29	-	-	5.00	12.90	0.89
RAMSEY4-4	17	4510.36	12.61	9.75	-	-	-	20.66	1.08
RAMSEY4-5	17	2.41	-	4.05	-	-	1.85	-	1.33
RAMSEY4-5	18	3.59	-	6.53	-	-	11.10	-	1.82
RAMSEY4-5	19	5.05	-	10.03	-	-	988.64	-	2.48
RAMSEY4-5	20	7.13	-	14.22	-	-	-	-	3.31
RAMSEY4-5	21	10.03	-	20.22	-	-	-	-	4.36
RAMSEY4-5	22	14.05	-	39.97	-	-	-	-	5.85
RAMSEY4-5	23	53.71	-	3791.01	-	-	-	-	12.94

Table 4: Results for SOKOBAN (top) and RAMSEY (bottom), times in seconds.

Problem	DLV	GnT	Smodels	Smodels2	ASSAT
REACH	8500	700	700	700	700
SAMEGEN	9025	676	676	676	676
STRATCOMP	170	115	n/a	n/a	n/a
QBF$_{GW}$	1200	40	n/a	n/a	n/a
QBF	48	40	n/a	n/a	n/a
HAMPATH	105	105	50	40	50
TSP	26	n/a	24	28	n/a

(a) Maximum instance size solved

Problem	DLV	GnT	GnT (opt. enc.)	Smodels	Smodels (opt. enc.)	ASSAT	ASSAT (opt. enc.)
SOKOBAN	95.00%	16.67%	n/a	46.67%	95.00%	n/a	n/a
RAMSEY	100.00%	39.29%	96.43%	28.57%	71.43%	39.29%	100.00%

(b) Percentage of instances solved

Table 5: Problem solving capacity on the benchmark problems

125MB of memory and generates 2,130,848 ground rules. In fact, in most cases the instantiation module of Smodels and GnT exceeded the memory limit and was halted.

However, the performance of Smodels improves significantly if the alternative encoding, using the cardinality constraints of Smodels, is employed. Using this encoding, Smodels can solve precisely the same instances solved by DLV, and it is even faster than DLV on some instances.[23] The positive performance of Smodels on this alternative encoding confirms that, as for TSP, the cardinality constraint construct of Smodels is very useful also from the viewpoint of efficiency when it can be exploited in an appropriate way.

We have not reported the times for ASSAT, because this system crashed on several instances of this problem (recall that ASSAT is very recent, and could not be tested as extensively as Smodels and DLV yet).

Summary

In Table 5, we have gathered the maximum instance sizes of the benchmark problems that the different systems were able to solve (i.e., the maximum size such that a system could handle *all* instances of that size) under the above time and memory limits, and for SOKOBAN and RAMSEY the percentage of instances solved.
Summarizing, we can draw the following conclusions:

- Currently, the DLV system is better suited to deal with "database oriented" applications (in the sense of deductive databases) than the other systems, as the optimization techniques implemented in its instantiator allow DLV to handle larger amounts of data in a more efficient way.
- The built-in implementation of disjunction allows the DLV system to be faster than the other systems on hard (Σ_2^P-hard) problems. Even if GnT is slower on such problems, its performance is very good

[23] It should be noted, however, that the alternative encoding for Smodels is completely different from the other encodings, and exploits more domain knowledge.

considering that disjunction is implemented through a rewriting technique. On the other hand, on fragments where disjunction is not strictly needed but a natural construct to express choice, DLV performs better. Smodels and ASSAT cannot handle Σ_2^P-hard problems at all, since they do not support disjunction and Σ_2^P-hard problems cannot be encoded uniformly by $\mathtt{v}$-free programs, unless the Polynomial Hierarchy collapses (cf. Section 4).

- On NP / co-NP / Δ_2^P problems like HAMPATH and TSP, the performance of the systems comes closer. DLV is slightly faster than the other systems, if a standard LP encoding is used. On the other hand, Smodels gets faster if its peculiar constructs (choice rules and cardinality constraints) are employed.

- Smodels gets very fast when the knowledge modeling power of cardinality constraints can be fully exploited. Indeed, Smodels is the fastest system on TSP where the availability of cardinality constraints admits a very different encoding that cannot be efficiently simulated by standard logic programming constructs. Smodels becomes fast also on SOKOBAN when an encoding, fully exploiting cardinality constraints, is employed. On the other hand, a direct reformulation of a normal LP encoding in terms of cardinality constraints and choice rules improves the performance of Smodels only marginally. This was the case for HAMPATH and RAMSEY where the performance of Smodels on the encoding with cardinality constraints remains very close to the performance with the standard LP encoding.

- The advanced instantiation procedure of DLV has a positive impact also on DLV's performance on some (NP/co-NP) hard problems like SOKOBAN and RAMSEY. Indeed, the DLV instantiator often requires less memory than Lparse, and generates smaller ground programs, which can be evaluated more efficiently.

- Thanks to the good performance on deductive database applications and on Σ_2^P-complete problems, DLV appears to be applicable to a larger range of problems than the other systems, which have been designed for a more specific class of problems.

- The approach implemented in the recent ASSAT system is very interesting, as it allows to exploit the huge amount of work on satisfiability checkers done in AI. This approach is promising especially on NP/co-NP problems, where the well-assessed heuristics of satisfiability checkers pay off (e.g., ASSAT was fast on the RAMSEY problem). However, this system still needs some work to make it more robust; we experienced a number of crashes on the Sokoban encodings, which are more elaborated than the others.
  It would be nice to see an extension of ASSAT for dealing with optimization problems, and enhancing ASSAT to compute *all* answer sets seems very important. Indeed, while DLV, GnT, and Smodels are able to compute all (or a given number of) answer sets of a program, ASSAT cannot generate all answer sets in one computation (see Section 7.1). This limitation may have a negative impact on the efficiency of ASSAT on some relevant tasks like, for instance, computing the set of all brave or cautious consequences of a program.

- Comparing the use of disjunction versus unstratified negation or other means to express choice, the experiments showed that, in the specific benchmark problems considered, DLV, and in some cases also GnT, is competitive to Smodels and ASSAT, and the use of the more familiar disjunction connective does not come at higher computational price.

Two strong points of DLV in comparison to the other systems are: (i) the built-in implementation of disjunction, which allows DLV to efficiently solve Σ_2^P-complete problems, and (ii) its advanced instantia-

tion module, which allows DLV to efficiently deal with deductive database applications, which supports a simpler way of programming (the user does not have to bother about domain predicates), and which also tends to affect performance on hard problems (like RAMSEY and SOKOBAN) positively, as it produces a smaller instantiation than Lparse, and requires less memory.

On the other hand, a strong point of Smodels is its support for advanced constructs (cardinality constraints and choice rules), which (i) often provide a more natural problem representation, and (ii) sometimes allow for more succinct problem encodings, having a very positive impact on the efficiency of the evaluation.

8 Ongoing Projects

The DLV system is currently employed in several international projects investigating applications of computational logic. In particular, DLV plays a central role in two projects funded by the European Commission under the Fifth Framework:

- INFOMIX (Boosting Information Integration) started in April 2002 and lasts 36 months. The overall project budget is more than 1.6 million Euro. The project is coordinated by the University of Calabria represented by Nicola Leone.

- ICONS (Intelligent Content Management System) started in January 2002 and lasts 24 months. The overall project budget is more than 3 million Euro. The project is coordinated by RODAN Systems represented by Witold Staniszkis.

DLV is the main software tool to be used in the INFOMIX project. The principal objective of INFOMIX is to provide an advanced system for information integration, which is innovative in several respects, including advanced reasoning capabilities, declarative user-interaction, and handling of incomplete and inconsistent data sources.

A main problem in data integration is the frequent presence of data inconsistencies and data incompleteness [16], requiring a number of complex (co-NP-hard) reasoning tasks to guarantee the correctness of the query results. Recent studies [67, 66] show that these problems are best solved by cautious reasoning on disjunctive logic programs. Thus, using DLV comes naturally, especially considering DLV's ability to deal with larger amounts of data as highlighted by the benchmark results in Section 7.

The ICONS project, on the other hand, focuses on the area of knowledge management. It is aimed at the implementation of an advanced platform for intelligent content management. There, DLV is employed for advanced knowledge representation and reasoning on the ontology base of the ICONS platform and will serve as the intelligent query-engine of ICONS.

Further research on the DLV system is pursued also in a number of national projects such as the FWF (Austrian Science Funds) project P14781, lead by Thomas Eiter. The main goal of this project is the realization of an advanced logic-based planning system, which is able to reason also under incomplete knowledge and supports cost-based planning in a declarative way. This planning system is being implemented as a DLV front-end. Furthermore, research on DLV is pursued in the Ludwig Wittgenstein Laboratory of Information Systems directed by Georg Gottlob in Vienna (FWF project Z29-INF), and DLV technology is employed also in the Lixto system (`http://www.lixto.com`).

9 Conclusion

After an extensive period of mainly foundational and theoretical research on nonmonotonic logics and databases, during the last years several implementations became available which can be utilized as advanced and powerful tools for problem solving in a highly declarative manner. They provide a computational backend for the Answer Set Programming (ASP) paradigm, in which the alternative solutions to a problem are encoded in the models (or answer sets) of a nonmonotonic theory. ASP has been recognized as a promising approach for dealing with problems which require advanced modeling capabilities for problem representation. Most recently, the European Commission has granted funding for a special Working Group on Answer Set Programming (WASP) that is formed by a number of research groups in Europe.

In this paper, we have presented the DLV system, which is a state-of-the-art implementation of disjunctive logic programming under the answer set semantics [55], enriched by further useful language constructs. This paper is the first comprehensive document in which a wide survey over several relevant aspects of the system is provided, covering technical, methodological, and application aspects.

Starting from an exposition of the core system language for problem representation, we have illustrated how knowledge representation and reasoning problems, even of high computational complexity, can be declaratively encoded in the DLV language following a Guess-Check-Optimize (**GCO**) paradigm. Furthermore, we have addressed computational aspects of the language by providing a complete picture of the complexity of the DLV language and relevant syntactic fragments thereof. The DLV engine exploits this picture by handling fragments of lower complexity more efficiently. On the benchmarking side, as an important contribution the paper presents and discusses a thorough experimentation activity which we have carried out on an ample set of benchmark problems, taken from various application areas and with different computational complexity. These experiments provide some evidence that, in comparison with other, similar systems, DLV may have a wider range of applicability and, due to its built-in database techniques and sophisticated processing of non-ground programs, is better suited to deal with larger amounts of input data. Furthermore, by its high expressiveness (up to Δ_3^P-complete problems), it is capable of representing more complex problems than comparable systems, whose expressiveness is limited to lower classes in the Polynomial Hierarchy.

The DLV system is disseminated in academia, and presumably soon also in industry. Indeed, the industrial exploitation of DLV in the emerging areas of Knowledge Management and Information Integration is the subject of two international projects funded by the European Commission. DLV is widely used for educational purposes in courses on databases and on AI, both in European and American universities. Also Lixto, an advanced web-extraction tool, employs DLV technology. The DLV system has been employed at CERN, the European Laboratory for Particle Physics located near Geneva, for an advanced deductive database application that involves complex knowledge manipulations on large-sized real databases. We believe that the strengths of DLV — its expressivity and solid implementation — make it an attractive tool for other similar applications.

While the DLV system has been continuously extended and improved over the last years, its development has not yet reached a terminal point. Currently, several language enhancements are under development which will be available in future releases. In particular, aggregates and other set functions will be available, which are very convenient for the encoding of many practical problems. On the other hand, improvements at the algorithmic level are underway by the development and implementation of more sophisticated magic set techniques than those which are currently available for disjunctive logic programs, cf. [61]. Furthermore, refinements of the current as well as novel heuristics in the model generation process might help to improve performance, and to further push scalability to larger sets of data.

From what has been achieved and what may be expected, we are confident that DLV, as well as similar ASP systems, will provide us with more and more widely applicable tools which render computational logic an important component in advanced information technology.

Acknowledgments We would like to thank Robert Bihlmeyer, Francesco Buccafurri, Francesco Calimeri, Simona Citrigno, Tina Dell'Armi, Giuseppe Ielpa, and Axel Polleres who contributed to the DLV project, as well as Ilkka Niemelä and Patrik Simons for fruitful discussions. Furthermore, we are grateful to Leo Bertossi, Jürgen Dix, Esra Erdem, Michael Fink, Michael Gelfond, Giuliana Sabbatini, Terry Swift, and many others for sharing their experiences on DLV with us and for useful suggestions.

References

[1] C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In T. Eiter, W. Faber, and M. 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.

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

[3] K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Los Altos, California, 1988.

[4] C. Aravindan, J. Dix, and I. Niemelä. Dislop: A research project on disjunctive logic programming. *AI Communications – The European Journal on Artificial Intelligence*, 10(3/4):151–165, 1997.

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

[6] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002. In press.

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

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

[9] 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.

[10] S. Brass and J. Dix. Disjunctive Semantics based upon Partial and Bottom-Up Evaluation. In L. Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming*, pages 199–213, Tokyo, June 1995. MIT Press.

[11] G. Brewka and T. Eiter. Preferred Answer Sets for Extended Logic Programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

[12] G. Brewka, I. Niemelä, T. Schaub, and M. Truszczyński. Dagstuhl Seminar Nr. 0238, Nonmonotonic Reasoning, Answer Set Programming and Constraints, September 15-20, 2002. System Competition, 2002. `http://www.cs.uni-potsdam.de/~canger/dagstuhl.html`.

[13] F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3), May 2002.

[14] F. Buccafurri, N. Leone, and P. Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[15] M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[16] A. Cali, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. In *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002*, pages 262–279, Toronto, Canada, 2002. Lecture Notes in Computer Science.

[17] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning Operators for Answer Set Programming Systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pages 200–209, April 2002.

[18] W. Chen and D. S. Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[19] P. Cholewiński, V. W. Marek, A. Mikitiuk, and M. Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.

[20] P. Cholewiński, V. W. Marek, and M. 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.

[21] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[22] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[23] J. Delgrande, T. Schaub, and H. Tompits. plp: A Generic Compiler for Ordered Logic Programs. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 411–415. Springer, 2001.

[24] 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.

[25] J. Dix and U. Furbach. The DFG Project DisLoP on Disjunctive Logic Programming. *Computational Logic*, 2(2):89–90, 1996.

[26] J. Dix, G. Gottlob, and V. W. Marek. Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations. *Fundamenta Informaticae*, 28:87–100, 1996. (This is a full version of [27].).

[27] J. Dix, G. Gottlob, and W. Marek. Causal Models for Disjunctive Logic Programs. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 290–302, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[28] D. East and M. aw Truszczyński. System Description: aspps – An Implementation of Answer-Set Programming with Propositional Schemata. In T. Eiter, W. Faber, and M. aw Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 402–405. Springer Verlag, September 2001.

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

[30] 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.

[31] U. Egly, T. Eiter, H. Tompits, and S. 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.

[32] T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[33] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[34] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing Preferred and Weakly Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. In A. Provetti and S. T. Cao, editors, *Proceedings AAAI 2001 Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 45–52, Stanford, CA, March 2001. AAAI Press.

[35] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. Technical Report INFSYS RR-1843-02-01, Institut für Informationssysteme, Technische Universität Wien, Jan. 2002.

[36] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, Proceedings*, number 1861 in Lecture Notes in AI (LNAI), pages 807–821, London, UK, July 2000. Springer Verlag.

[37] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. Technical Report INFSYS RR-1843-01-12, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001. To appear in *Artificial Intelligence*.

[38] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. Technical Report INFSYS RR-1843-01-11, Institut für Informationssysteme, Technische Universität Wien, Dec. 2001. Submitted for journal publication, minor revision under review.

[39] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer Set Planning under Action Costs. In S. Flesca, S. Greco, G. Ianni, and N. Leone, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA)*, number 2424 in Lecture Notes in Computer Science, pages 186–197, Sept. 2002.

[40] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A Framework for Declarative Update Specifications in Logic Programs. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654. Morgan Kaufmann, 2001. Extended version Technical Report INFSYS RR-1843-02-07, TU Wien, 2002.

[41] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. An Update Front-End for Extended Logic Programs. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, number 2173 in LNCS, pages 397–401. Springer, 2001. System Description (abstract).

[42] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On Properties of Update Sequences Based on Causal Rejection. *Journal of the Theory and Practice of Logic Programming*, 2(6):721–777, 2002.

[43] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[44] T. Eiter, G. Gottlob, and N. Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[45] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[46] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[47] T. Eiter, N. Leone, and D. Saccà. On the Partial Semantics for Disjunctive Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(1–2):59–96, April 1997.

[48] T. Eiter, N. Leone, and D. Saccá. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theoretical Computer Science*, 206(1–2):181–218, October 1998.

[49] W. Faber. `dlvi` homepage, 1999. `<URL:http://www.dlvsystem.com/inheritance/>`.

[50] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In I. O. Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[51] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

[52] W. Faber and G. Pfeifer. `DLV` homepage, since 1996. `http://www.dlvsystem.com/`.

[53] J. Fernández and J. Minker. Semantics of Disjunctive Deductive Databases. In *Proceedings 4th Intl. Conference on Database Theory (ICDT-92)*, pages 21–50, Berlin, October 1992.

[54] 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.

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

[56] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2(3-4):193–210, 1998.

[57] I. Gent and T. Walsh. The QSAT Phase Transition. In *Proceedings of the 16th AAAI*, 1999.

[58] E. Giunchiglia and V. Lifschitz. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, pages 623–630, 1998.

[59] G. 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.

[60] G. Gottlob, N. Leone, and H. Veith. Succinctness as a Source of Expression Complexity. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.

[61] S. Greco. Optimization of Disjunction Queries. In D. D. Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 441–455, Las Cruces, New Mexico, USA, Nov. 1999. The MIT Press.

[62] T. Janhunen, I. Niemela, P. Simons, and J.-H. You. Partiality and disjunctions in stable model semantics. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), April 12-15, Breckenridge, Colorado, USA*, pages 411–419. Morgan Kaufmann Publishers, Inc., 2000.

[63] D. 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.

[64] D. E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. ACM Press, New York, 1994.

[65] C. Koch and N. Leone. Stable model checking made easy. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999*, pages 70–75, Stockholm, Sweden, Aug. 1999. Morgan Kaufmann Publishers.

[66] D. Lembo, M. Lenzerini, and R. Rosati. Integrating inconsistent and incomplete data sources. In *Proceedings of SEBD 2002*, Portoferraio, Isola d'Elba, 2002. 299–308.

[67] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *Proceedings of the Knowledge Representation meets Databases International Workshop (KRDB-02)*, Toulouse France, 2002. CEUR Electronic Workshop Proceedings `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-54/`.

[68] N. Leone, S. Perri, and F. Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In T. Eiter, W. Faber, and M. aw Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI). Springer Verlag, September 2001.

[69] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, June 1997.

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

[71] V. Lifschitz and H. Turner. Splitting a Logic Program. In P. 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.

[72] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[73] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[74] W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.

[75] N. McCain and H. Turner. Satisfiability Planning with Causal Theories. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[76] J. Minker. On Indefinite Data Bases and the Closed World Assumption. In D. 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.

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

[78] 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.

[79] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. 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.

[80] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. In C. Baral and M. Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[81] C. H. Papadimitriou. The complexity of unique solutions. *Journal of the ACM*, 31:492–500, 1984.

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

[83] D. Pearce, V. Sarsakov, T. Schaub, H. Tompits, and S. Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, 2002.

[84] D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5(1):97–110, 1989.

[85] T. Przymusinski. Stationary semantics for disjunctive logic programs and deductive databases. In *Proceedings of North American Conference on Logic Programming*, pages 40–62, 1990.

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

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

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

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

[90] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In J. Dix, U. Furbach, and A. 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.

[91] R. Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32:57–95, 1987.

[92] K. 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.

[93] 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.

[94] T. Schaub and K. Wang. A Comparative Study of Logic Programs with Preference. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 597–602, Seattle, WA, USA, Aug. 2001. Morgan Kaufmann Publishers.

[95] D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In A. 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.

[96] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[97] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138, June 2002.

[98] T. Syrjänen. Lparse 1.0 user's manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[99] A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[100] 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.

[101] Y. Zhao. ASSAT homepage, 2002. `http://assat.cs.ust.hk/`.

A Sokoban Encodings

The SOKOBAN problem has been encoded in DLV as follows.

$time(T) \text{ :- } \#int(T).$
$actiontime(T) \text{ :- } \#int(T), T <> \#maxint.$

$left(L1, L2) \text{ :- } right(L2, L1).$
$bottom(L1, L2) \text{ :- } top(L2, L1).$

$adj(L1, L2) \text{ :- } right(L1, L2).$
$adj(L1, L2) \text{ :- } left(L1, L2).$
$adj(L1, L2) \text{ :- } top(L1, L2).$
$adj(L1, L2) \text{ :- } bottom(L1, L2).$

$push(B, right, B1, T) \vee \neg push(B, right, B1, T) \text{ :- } reachable(L, T), right(L, B), box(B, T),$
$pushable_right(B, B1, T), good_pushlocation(B1), actiontime(T).$
$push(B, left, B1, T) \vee \neg push(B, left, B1, T) \text{ :- } reachable(L, T), left(L, B), box(B, T),$
$pushable_left(B, B1, T), good_pushlocation(B1), actiontime(T).$
$push(B, up, B1, T) \vee \neg push(B, up, B1, T) \text{ :- } reachable(L, T), top(L, B), box(B, T),$
$pushable_top(B, B1, T), good_pushlocation(B1), actiontime(T).$
$push(B, down, B1, T) \vee \neg push(B, down, B1, T) \text{ :- } reachable(L, T), bottom(L, B), box(B, T),$
$pushable_bottom(B, B1, T), good_pushlocation(B1), actiontime(T).$

$reachable(L, T) \text{ :- } sokoban(L, T).$
$reachable(L, T) \text{ :- } reachable(L1, T), adj(L1, L), \text{not } box(L, T).$

$pushable_right(B, D, T) \text{ :- } box(B, T), right(B, D), \text{not } box(D, T), actiontime(T).$
$pushable_right(B, D, T) \text{ :- } pushable_right(B, D1, T), right(D1, D), \text{not } box(D, T).$
$pushable_left(B, D, T) \text{ :- } box(B, T), left(B, D), \text{not } box(D, T), actiontime(T).$
$pushable_left(B, D, T) \text{ :- } pushable_left(B, D1, T), left(D1, D), \text{not } box(D, T).$
$pushable_top(B, D, T) \text{ :- } box(B, T), top(B, D), \text{not } box(D, T), actiontime(T).$
$pushable_top(B, D, T) \text{ :- } pushable_top(B, D1, T), top(D1, D), \text{not } box(D, T).$
$pushable_bottom(B, D, T) \text{ :- } box(B, T), bottom(B, D), \text{not } box(D, T), actiontime(T).$
$pushable_bottom(B, D, T) \text{ :- } pushable_bottom(B, D1, T), bottom(D1, D), \text{not } box(D, T).$

$sokoban(L, T1) \text{ :- } push(_, right, B1, T), \#succ(T, T1), right(L, B1).$
$sokoban(L, T1) \text{ :- } push(_, left, B1, T), \#succ(T, T1), left(L, B1).$
$sokoban(L, T1) \text{ :- } push(_, up, B1, T), \#succ(T, T1), top(L, B1).$
$sokoban(L, T1) \text{ :- } push(_, down, B1, T), \#succ(T, T1), bottom(L, B1).$
$\neg sokoban(L, T1) \text{ :- } push(_, _, _, T), \#succ(T, T1), sokoban(L, T).$

$box(B, T1) \text{ :- } push(_, _, B, T), \#succ(T, T1).$
$\neg box(B, T1) \text{ :- } push(B, _, _, T), \#succ(T, T1).$

$box(LB, T1) \text{ :- } box(LB, T), \#succ(T, T1), \text{not } \neg box(LB, T1).$

$\text{ :- } push(B, _, _, T), push(B1, _, _, T), B <> B1.$
$\text{ :- } push(B, D, _, T), push(B, D1, _, T), D <> D1.$
$\text{ :- } push(B, D, B1, T), push(B, D, B11, T), B1 <> B11.$

$good_pushlocation(L) \text{ :- } right(L, _), left(L, _).$
$good_pushlocation(L) \text{ :- } top(L, _), bottom(L, _).$
$good_pushlocation(L) \text{ :- } solution(L).$

$goal_unreached \text{ :- } solution(L), \text{not } box(L, \#maxint).$
$\text{ :- } goal_unreached.$

For GnT, we used the same encoding with the addition of domain predicates and minor syntactic changes

to conform to the different syntax.

The first encoding for Smodels, that we tried to use also for ASSAT, has been obtained from the encoding for GnT, rewriting disjunction in the "guessing" rules for $push$ by unstratified negation.

Finally, we used an optimized encoding for Smodels (Smodels2) by Tommi Syrjänen which is available at `http://www.tcs.hut.fi/~tssyrjan/software/`. This encoding exploits more omain knowledge than the previous ones.