Chapter 4

DECLARATIVE PROBLEM-SOLVING USING THE DLV SYSTEM

Thomas Eiter
Technische Universität Wien, Austria
eiter@kr.tuwien.ac.at

Wolfgang Faber
Technische Universität Wien, Austria
faber@kr.tuwien.ac.at

Nicola Leone
Technische Universität Wien, Austria
leone@dbai.tuwien.ac.at

Gerald Pfeifer
Technische Universität Wien, Austria
pfeifer@dbai.tuwien.ac.at

Abstract The need for representing indefinite information led to disjunctive deductive databases, which also fertilized work on disjunctive logic programming. Based on this paradigm, the **DLV** system has been designed and implemented as a tool for declarative knowledge representation. In this paper, we focus on the usage of **DLV** for solving problems in a declarative manner and report on experiments that we have run on a suite of benchmark problems. We discuss how problems can be solved in a natural way using a "**Guess&Check**"-paradigm where solutions are guessed and verified by parts of the program. Furthermore, we describe various front-ends that can be used for solving problems in specific applications. The experiments show that due to the ongoing implementation efforts, which include fine-tuning of the underlying algorithms, successive and significant performance improvements have been achieved during the last two years. The results indicate that **DLV** is capable of solving some complex problems quite efficiently.

Keywords: Disjunctive logic programming, disjunctive databases, nonmonotonic reasoning

1 INTRODUCTION

During the past two decades, much research has been devoted to formal theories for advanced knowledge representation, and systems which are tailored to solving specific knowledge representation (KR) problems have been implemented. Only more recently, work has been spent on implementations of systems that are applicable to a wider range of KR problems. In the area of nonmonotonic reasoning and pure (non-Prolog) logic programming, implementations of such systems have been described e.g. in (Aravindan et al., 1997; Eiter et al., 1997c; Eiter et al., 1998b; Cholewiński et al., 1996; Cholewiński et al., 1999; Niemelä and Simons, 1996; Niemelä and Simons, 1997; Chen and Warren, 1996; Rao et al., 1997; Seipel and Thöne, 1994) and also (McCain, 1999), as described in (Lifschitz, 1999b). These systems allow problems to be represented from various application fields in declarative languages.

The development of the **DLV** system (datalog plus vel, i.e., disjunction[1]) (Eiter et al., 1997c; Eiter et al., 1998b) is an ongoing effort whose goal is to provide a KR tool which can be used for declarative problem-solving in an efficient way. The system is based on disjunctive logic programming (Lobo et al., 1992; Minker, 1994) without function symbols under the consistent answer set semantics (Gelfond and Lifschitz, 1991; Przymusinski, 1991; Eiter et al., 1997b; Leone et al., 1997), and has the following important features:

Advanced knowledge modeling capabilities. **DLV** provides support for declarative problem solving in severals respects:

- It is capable of dealing with extended disjunctive logic programs that allow strong negation (Gelfond and Lifschitz, 1991), which can be easily expressed through stable models of disjunctive logic programs (Przymusinski, 1991).
- It has high expressiveness in a formally precise sense (Σ_2^P), which means that problems can be uniformly solved by a fixed program over varying instances that can not be expressed by languages with lower expressiveness such as Smodels (Niemelä and Simons, 1996), whose expressiveness is limited to NP (See Remark A at the end of Section 2). Moreover, some Σ_2^P-complete problems can be expressed in an elegant way.
- It allows for declarative problem solving following a "**Guess&Check**" paradigm where a solution to a problem is guessed by one part of a program and then verified through another part of the program. This will be discussed in more detail shortly below and in Section 3
- It offers a number of front-ends for dealing with specific AI applications.

[1] "Vel" is Latin for "or".

Solid Implementation. The high expressiveness of **DLV** comes at the price of a high computational cost in the worst case, and the performance of a naïve implementation would thus be infeasible for applications. Much effort has been spent on sophisticated algorithms and techniques for improving the performance, including the implementation of

- deductive database optimization techniques, and
- nonmonotonic reasoning optimization techniques.

Note that the Σ_2^P-expressiveness of **DLV** implies that a flat intelligent backtracking algorithm where each node has polynomial complexity is not feasible, and thus requires a more complicated algorithm with involved optimization techniques. The **DLV** system is the state-of-the-art implementation of answer set semantics for extended disjunctive logic programs.

Database Interfaces. The **DLV** system provides an experimental interface to a relational database (Oracle) and, by means of a special query tool (Koch, 1999), also to an object-oriented database (Objectivity), which is useful for the integration of a specific problem solver developed in **DLV** into a more complex system.

In the present paper, we focus on the usage of the **DLV** system for declarative problem solving, and we report on experiments that we have run on a suite of benchmark problems, which show that the ongoing work on implementation of **DLV** led to significant performance improvements within the last years.

After recalling the syntax and semantics of **DLV**'s core language in Section 2, we discuss in Section 3 how problems can be expressed following a "**Guess&Check**" paradigm (which is a close relative of the generate-and-test paradigm in the AI community (Winston, 1992)). This paradigm has been used and described in the context of logic programming using stable models: e.g. in (Eiter et al., 1997b; Marek and Truszczyński, 1999); implicitly also in (Buccafurri et al., 1997); in (Niemelä, 1999) Datalog rules are viewed as constraints over the Herbrand Base; in (Cadoli et al., 1999) a declarative specification language based on Datalog is described which allows for specifying guessing predicates and constraints; in (Lifschitz, 1999b) plans are guessed and their validity is checked.

Informally, a candidate for the solution of a problem is nondeterministically generated through guessing rules of the program and verified using a checking part which consists of constraints and further rules. An important point is that, different from other programming languages, the guessing and checking parts of the program are purely declarative. The major strength of **DLV** compared to other systems such as Smodels (Niemelä and Simons, 1996; Niemelä and Simons, 1997) is that owing to the expressiveness of disjunction, also checking parts which solve co-NP complete problems can be specified; combined with a nondeterministic guessing part, this leads to Σ_2^P-expressiveness. Note that the DeReS system (Cholewiński et al., 1996; Cholewiński et al., 1999) offers similar expressiveness, but does not allow to uniformly solve varying instances of the same problem through a fixed program in a declarative language, as it lacks support for variables. Also Smodels' support for variables is limited in the sense that all variables must occur in at least one positive body literal whose predicate is non-recursively defined, which makes many problem encodings less straightforward and more cumbersome, and causes some inefficiencies in the evaluation of recursive programs. An example of this are programs computing transitive closures of input relations. The efficient support for programs with variables is one of the strong points of **DLV**, and makes **DLV** more amenable than the above mentioned systems for deductive database applications (see also Section 7).

In Section 4, we briefly describe some front-ends which we have developed for various applications. In particular, we review front-ends for major non-monotonic reasoning modes, model-based diagnosis, SQL3, and inheritance. Our experience with developing these front-ends shows that **DLV** is a suitable declarative programming engine on top of which other declarative formalisms for knowledge representation having high expressiveness can be implemented.

After describing the user interface of **DLV** in Section 5, and providing an overview of the implementation of **DLV** in Section 6, we report in Section 7, the improvement of the system performance during the last two years, and we briefly address ongoing and future work, as well as the current dissemination of **DLV**.

2 CORE LANGUAGE

In this section, we provide a formal definition of the syntax and semantics of the kernel language of **DLV**: disjunctive Datalog extended with strong negation. For further background, see (Lobo et al., 1992; Eiter et al., 1997b; Gelfond and Lifschitz, 1991).

2.1 SYNTAX

Strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. 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 $\texttt{not}\ l$, where l is a classical literal; in the former case, it is *positive*, and in the latter case *negative*. Unless stated otherwise, by *literal* we mean a classical literal.

Given a classical literal l, its *complementary* literal 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.

In addition to literals as defined above, **DLV** also supports built-ins like equality, less-than, and greater-than, as well as various arithmetic predicates like addition or multiplication. For details, we refer to (Faber and Pfeifer, 1996).

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

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

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 `v` $\cdots$ `v` a_n is the *head* of r, while the conjunction $b_1, ..., b_k$, `not` $b_{k+1}, ...,$ `not` b_m is the *body* of r. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. If the body is empty (i.e. $k = m = 0$), we usually omit the "`:-`" sign.

We denote by $H(r) = \{a_1, \ldots, a_n\}$ the set of literals in the head, and by $B(r) = B^+(r) \cup B^-(r)$ the set of the body literals, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$ are the sets of positive and negative body literals, respectively.

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

As usual, a term (an atom, a rule,...) is called *ground*, if no variables appear in it. A ground program is also called a *propositional* program.

2.2 SEMANTICS

In this section we describe the semantics of consistent answer sets, originally defined in (Gelfond and Lifschitz, 1991).

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

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

Ground Instantiation. Given a 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 program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r \in \mathcal{P}} Ground(r)$. 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 two steps, following (Lifschitz, 1996): First we define the answer sets of positive programs, then we give a reduction of programs containing negation to positive ones and use it to define answer sets of arbitrary programs, possibly containing negation as failure.

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

---

[2]Note that we only consider *consistent answer sets*, while in (Lifschitz, 1996) also the inconsistent set of all possible literals is a valid answer set.

Example 1 For the positive program $\mathcal{P}_1 = \{$`a v ¬b v c.`$\}$, the answer sets are $\{$`a`$\}$, $\{$`¬b`$\}$, and $\{$`c`$\}$.

For the positive program $\mathcal{P}_2 = \{$`a v ¬b v c. ; :- a.`$\}$, the sets $\{$`¬b`$\}$ and $\{$`c`$\}$ are the answer sets.

Finally, for the positive program $\mathcal{P}_3 = \{$`a v ¬b v c. ; :- a. ; ¬b :- c. ; c :- ¬b.`$\}$, the set $\{$`¬b, c`$\}$ is the only answer set.

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 Given the program $\mathcal{P}_4 = \{$`a v ¬b :- c. ; ¬b :- not a, not c. ; a v c :- not ¬b.`$\}$ and $I = \{$`¬b`$\}$, the reduct $\mathcal{P}_4^I$ is $\{$`a v ¬b :- c. ; ¬b.`$\}$. It is easy to see that I is an answer set of $\mathcal{P}_4^I$, and for this reason it is also an answer set of $\mathcal{P}_4$.

Now consider $J = \{$`a`$\}$. The reduct $\mathcal{P}_4^J$ is $\{$`a v ¬b :- c. ; a 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 $K = \{$`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 v ¬b :- c.` It holds that $B(r) \subseteq K$, but $H(r) \cap K \neq \emptyset$ is violated). Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_4$.

Remark A. In some cases it is possible to emulate disjunction by non-stratified normal rules. However, this is not possible in general.

For example, consider $\mathcal{P}_5 = \{$`a v b.`$\}$ and $\mathcal{P}_5' = \{$`a :- not b. ; b :- not a.`$\}$. Both programs have the same answer sets, namely $\{$`a`$\}$ and $\{$`b`$\}$.

On the other hand, $\mathcal{P}_6 = \{$`a v b. ; a :- b. ; b :- a.`$\}$ has $\{$`a,b`$\}$ as its single answer set, while $\mathcal{P}_6' = \{$`a :- not b. ; b :- not a. ; a :- b. ; b :- a.` $\}$ has no answer set at all.

3 THE "GUESS&CHECK" PARADIGM

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

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

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

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

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

Without imposing restrictions on which rules G and C may contain, in the extremal case we might set G to the full program and let C be empty, i.e., all checking is moved to the guessing part such that solution candidates are always solutions. This is certainly not intended. 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 solving a number of problems, however, it is possible to design a natural **Guess&Check** program in which the two parts are clearly identifiable and have a simple structure:

- The guessing part G consists of a disjunctive rule which "guesses" a solution candidate S.
- The checking part C consists of integrity constraints which check the admissibility of S, possibly using auxiliary predicates which are defined by normal stratified[3] rules.

Thus, the disjunctive rule defines the search space[4], in which rule applications are branching points, while the integrity constraints prune illegal branches.

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

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

[3]For a definition of stratification, see (Apt et al., 1988).

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

3-Colorability is a classical NP-complete problem. Assuming that the set of nodes V and the set of edges E are specified by means of predicates *node* (which is unary) and *edge* (binary), respectively, it can be encoded by the following **Guess&Check** program:

```
r: col(X,r) v col(X,g) v col(X,b) :- node(X).   } Guess

c: :- edge(X,Y), col(X,C), col(Y,C).            } Check
```

The rule r nondeterministically guesses color assignments for the nodes in the graph, and the constraint c checks that these choices are legal, i.e., that no two nodes which are connected by an edge have the same color.[5]

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

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

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

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

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

```
inPath(X,Y) v outPath(X,Y) :- arc(X,Y).                    } Guess

 :- inPath(X,Y), inPath(X,Y1), Y <> Y1.  \
 :- inPath(X,Y), inPath(X1,Y), X <> X1.   } Constraints  \
 :- node(X), not reached(X).             /                } Check
 reached(X) :- start(X).                 \  Auxiliary    /
 reached(X) :- reached(Y), inPath(Y,X).  /  Predicate
```

The first rule guesses a subset S of all given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian

[5]In this example, we assume that G contains no loops, i.e., edges from a node to itself. Such loops can be easily handled by adding $X <> Y$ to the constraint.

Path. Here, the checking part C uses an auxiliary predicate `reached`, which is defined using positive recursion.

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

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

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

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

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

The problems 3COL and HPATH are popular examples of NP-complete problems. We next show that even some harder problem, which is located at the second level of the polynomial hierarchy, can be encoded by the **Guess&Check** technique. To this end, we consider the following problem *Strategic Companies*.

STRATCOMP: Given the collection $C = \{c_1, \ldots c_m\}$ of companies c_i owned by a holding, together with information about the products each company produces and company control, compute the set of the strategic companies in the holding.

Let us recall from (Cadoli et al., 1997) what a "strategic company" is in this context. Each company $c_i \in C$ in the holding is producing a collection G_i of goods, such that the holding produces a collection of goods $G = \bigcup_{c_i \in C} G_i$. Company control information models that a set of companies $D \subseteq C$ jointly may have control (e.g., by majority in shares) over another company $c_i \in C$. (Companies not in C, which we do not model here, might have shares in companies as well.) The company control information in STRATCOMP lists records of such control information in terms of "controlling sets" D for "controlled" companies c_i. Note that, in general, a company might have more than one controlling set, and only irredundant controlling sets (i.e., no proper subset is a controlling set) are recorded then.

Now, some companies should be sold by the holding, while the following two conditions have to be maintained: (1) After the transaction, the remaining set of companies $C' \subset C$ still allows one to produce all goods, i.e., $\bigcup_{c_i \in C'} G_i = G$.

(2) No company is sold which would still be controlled by the holding after the transaction, i.e., if D is a controlling set for $c_i \in C$ and $D \subseteq C'$ holds, then also $c_i \in C'$ holds.

A set $C' \subseteq C$ is called a *strategic set*, if it is minimal with respect to inclusion, that is, it satisfies both (1) and (2) and no proper subset of C' satisfies both (1) and (2). In general, the strategic set is not unique, and multiple solutions for C' exist. A company $c_i \in C$ is called *strategic*, if it belongs to at least one of these strategic sets.

Computing the set of all strategic companies is relevant when companies should be sold, as selling any company which is not strategic for sure does not lead to a violation of any of the conditions (1) and (2). This problem is Σ_2^P-hard in general (Cadoli et al., 1997); reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is the only KR problem from the business domain of this complexity that has been considered so far.

We next present a program, $\mathcal{P}1$, which solves the complex problem STRAT-COMP in a surprisingly elegant way by a few rules:

$$
\begin{array}{lll}
r: & \texttt{strat(Y) v strat(Z) :- produced\_by(X,Y,Z).} & \left.\right\}\ \textbf{Guess} \\
c: & \left.\begin{array}{l}\texttt{strat(W) :- controlled\_by(W,X,Y,Z),} \\ \qquad\qquad\ \ \texttt{strat(X), strat(Y), strat(Z).}\end{array}\right\} & \textbf{Constraint}
\end{array}
$$

Here $\texttt{strat(X)}$ means that $\texttt{X}$ is strategic, $\texttt{produced\_by(X,Y,Z)}$ that product $\texttt{X}$ is produced by companies $\texttt{Y}$ and $\texttt{Z}$, and $\texttt{controlled\_by(W,X,Y,Z)}$ that $\texttt{W}$ is jointly controlled by $\texttt{X}, \texttt{Y}$ and $\texttt{Z}$. We assume that a set of facts for *company*, *controlled_by* and *produced_by* is part of the input and have adopted the setting from (Cadoli et al., 1997) where each product is produced by at most two companies and each company is jointly controlled by at most three other companies (in this case, the problem is still Σ_2^P-hard).

The guessing part G of the program consists of the disjunctive rule r, and the checking part C consists of the normal rule c. The program $\mathcal{P}1$ exploits the minimization which is inherent to the semantics of (consistent) 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 intuitively selects one of the companies $\texttt{c}_1$ and $\texttt{c}_2$ that produce some item $\texttt{g}$, which is described by $\texttt{produced\_by}(\texttt{g}, \texttt{c}_1, \texttt{c}_2)$. If there were no company control information, minimality of answer sets would then naturally ensure that the answer sets of $F \cup \{r\}$ correspond to the strategic sets; no further checking would be needed. However, in case such control information (given by facts $\texttt{controlled\_by}(\texttt{c}, \texttt{c}_1, \texttt{c}_2, \texttt{c}_3)$) is available, the rule c 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 $\mathcal{P}1 \cup F$ correspond one-to-one to the strategic sets of the holding. Thus, the set of all strategic companies is given by the set of all companies c for which the fact strat(c) is true under brave reasoning (see Section 4.1 for a description).

An important note here is that the checking constraint c interferes with the guessing rule r: applying c may "spoil" the minimal answer set generated by rule r. For example, suppose the guessing part gives rise to a ground rule

$$r' : \mathtt{strat(c1)\ v\ strat(c2)\ :\!-\ produced\_by(g, c1, c2).}$$

and the fact produced_by(g, c1, c2) is given in 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 c is applied which derives strat(c2), then the application of the rule r' 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'.

"Feedback" or "influence" of the checking part C on the guessing part G, in terms of literals which are derived in C and invalidate the application of rules in G or make further rules in G applicable (and thus change the guess), can be made precise in terms of a "potentially uses" relation (Eiter et al., 1994; Eiter et al., 1997b) and a "splitting set" (Lifschitz and Turner, 1994)). Such interference is in fact needed to solve STRATCOMP in the way we have outlined, if C does not contain disjunctive rules. This follows from complexity considerations: If C had no influence on G (augmented by F), then the answer sets of $G \cup C \cup F$ would just be the answers sets of $C \cup A_0$, where A_0 is an answer set of $G \cup F$. Furthermore, if C contains no disjunctive rule and G has complexity in NP (e.g., if G is, as in this case, head-cycle-free (Ben-Eliyahu and Dechter, 1994)), along with A_0 an answer set A of $C \cup A_0$ 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 $\Sigma_2^P = \mathrm{NP}$, a hypothesis that is believed to be false by the experts, we can thus conclude that it is impossible to solve the STRATCOMP problem by a program whose answer sets correspond to the strategic sets, but in which C has no influence on G.

In general, if a program encodes a problem that is Σ_2^P-complete, then the checking part C must contain disjunctive rules unless C has influence on the guessing part G (or G has Σ_2^P-complexity). In particular, if the above program $\mathcal{P}1$ is rewritten to eliminate such influence, then further disjunctive rules must be added. This is exemplified by the following program $\mathcal{P}2$, which expresses the strategic sets in the generic **Guess&Check**-paradigm, where the guessing and the checking part are clearly separated:

```
strat(X) v ¬strat(X) :- company(X).                          } Guess
:- produced_by(X,Y,Z), not strat(Y), not strat(Z).
:- controlled_by(W,X,Y,Z), not strat(W),
   strat(X), strat(Y), strat(Z).
:- not min(X), strat(X).
:- strat′(X,Y), ¬strat(Y).
:- strat′(X,X).                                                 Check
min(X) v strat′(X,Y) v strat'(X,Z) :- produced_by(G,Y,Z),
                                       strat(X).
min(X) v strat′(X,C) :- controlled_by(C,W,Y,Z), strat(X),
                        strat′(X,W), strat′(X,Y), strat′(X,Z).
strat′(X,Y) :- min(X), strat(X), strat(Y), X<>Y.
```

Here company(X) means that X is a company. The guessing rule selects a subset C' of the companies that serves as a candidate strategic set. The checking part of $\mathcal{P}2$ is far more complicated than in the previous examples. The first two constraints check whether all goods are still produced and whether company control is maintained with respect to the candidate set. The remaining part of the program then checks for the minimality of C'. For each company $c \in C'$, the predicate $\mathtt{min(c)}$ must be true. Informally, this predicate means that $C' \setminus \{\mathtt{c}\}$ does not contain a strategic set, i.e., for each subset $C'' \subseteq C' \setminus \{\mathtt{c}\}$ either some good is no longer produced or company control is not obeyed. This is checked by using a binary predicate $\mathtt{strat'(c,Y)}$, where the second argument describes those companies from $S \setminus \{\mathtt{c}\}$ which are contained in C''. The atom $\mathtt{min(c)}$ is derived if such a violation is detected, and in this case $\mathtt{strat'(c,c')}$ is set to true for each $\mathtt{c'} \in C' \setminus \{\mathtt{c}\}$ by the last rule. The remaining two constraints enforce that $\mathtt{strat(c,c')}$ must be false for each $\mathtt{c'} \notin C' \setminus \{\mathtt{c}\}$.

We do not discuss the program $\mathcal{P}2$ in further detail here. It can be seen that its answer sets for a given instance correspond one-to-one to the strategic sets. Note, however, that the checking part of $\mathcal{P}2$ contains disjunctive rules. As we have discussed above, this is actually necessary given the one-way relationship of the guessing and the checking parts in $\mathcal{P}2$.

Further examples of programs in which the checking part necessarily contains disjunctive rules can be found in (Eiter et al., 1997b).

4 FRONT-ENDS

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

As of this writing, **DLV** contains front-ends for brave and cautious reasoning, model-based diagnosis (which we have described in detail in (Eiter et al., 1999)), SQL3, and a front-end extending the core language with objects and inheritance as described in (Buccafurri et al., 1999).

Each of these front-ends reads input in its language and transforms it into a disjunctive Datalog program. It then invokes the **DLV** kernel, post-processes the answer sets (if any) 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.

4.1 BRAVE AND CAUTIOUS REASONING

The brave and cautious reasoning front-ends are simple extensions of the kernel language. In addition to a disjunctive Datalog program, the user also specifies a query Q, which essentially looks like a ground body of a rule, terminated by a question mark:

$$b_1, \cdots, b_k, \ \texttt{not}\ b_{k+1}, \cdots, \ \texttt{not}\ b_m?$$

where each b_i $(1 \leq i \leq m)$ is a ground literal.

When doing brave reasoning, **DLV** looks for an answer set M where the query Q evaluates to true. Upon success, the system replies: "Q is bravely true, evidenced by M"; otherwise the answer is "Q is bravely false".

Similarly, in the cautious case the system answers: "Q is cautiously true" if and only if Q is true in all answer sets. In case of a negative answer, we get "Q is cautiously false, evidenced by M", where M is an answer set witnessing the falsity of Q.

4.2 DIAGNOSIS

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

DLV provides two different diagnosis front-ends: one for *abductive diagnosis* (Poole, 1989; Eiter et al., 1997a), and one for *Reiter's consistency-based diagnosis* (Reiter, 1987), both of which currently support three different modes: general diagnosis, where all diagnoses are computed, subset minimal diagnosis[6], 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 3 (Network Diagnosis) Suppose that machine a is online in the computer network depicted in Figure 4.1, 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

[6] For positive non-disjunctive theories only.

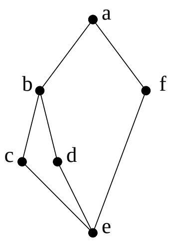

Figure 4.1 Computer network

```
reaches(X,X)  :-  node(X), not offline(X).
reaches(X,Z)  :-  reaches(X,Y), connected(Y,Z),
                  not offline(Z).
```

the observations are

```
not offline(a).  not reaches(a,e).
```

and the hypotheses are

```
offline(a).  offline(b).  offline(c).
offline(d).  offline(e).  offline(f).
```

The front-end for consistency-based diagnosis, as described in (Reiter, 1987), is similar to the one for abductive diagnosis.

4.3 INHERITANCE

In (Buccafurri et al., 1999) a novel language, DLP$^<$, has been defined. It extends disjunctive Datalog by grouping rules into objects and defining a partial order among these objects.

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 (Buccafurri et al., 1999).

Example 4 Consider the following program $\mathcal{P}$(`tweety`):

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

$\mathcal{P}$(`tweety`) consists of three objects (`bird`, `penguin`, and `tweety`). The partial order $<$ on the objects is specified in the program by the transitive reduction, which is denoted by the colon. The transitive reduction in the example is `tweety` : `penguin` and `penguin` : `bird`, so the corresponding partial

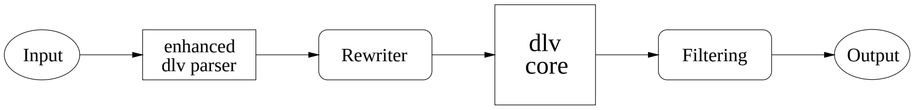

Figure 4.2 Inheritance Front-end.

order consists of `tweety` $<$ `penguin`, `penguin` $<$ `bird`, and `tweety` $<$ `bird`. Each object is followed by its rules which are enclosed by { }.

$\mathcal{P}$(`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 DLP$^<$ directly generalizes the kernel language of **DLV** since an ordinary disjunctive Datalog program exactly corresponds to a 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 (Buccafurri et al., 1999).

In the same paper a reduction from DLP$^<$ to disjunctive Datalog under the answer set semantics is described. We have implemented this transformation as part of a complete DLP$^<$ front-end on top of **DLV**.

A schematic visualization of the architecture of this front-end is shown in Figure 4.2.

Basically, the front-end works as follows: First of all, we have extended the **DLV** parser to incorporate the DLP$^<$ syntax. In this way, all the advanced features of **DLV** (e.g. bounded integer arithmetics, comparison built-ins, etc.) are also available in DLP$^<$. The *Rewriter* module implements the translation described in (Buccafurri et al., 1999). Once the rewritten version $\pi(\mathcal{P})$ of $\mathcal{P}$ is generated, its answer sets are then computed using the **DLV** core. Before the output is shown to the user, internal predicates are stripped from the output, which have been introduced by π.

On the web page (Faber, 1999) the prototype is described in detail, and executables are available for various platforms.

4.4 SQL3

In addition to the logic based front-ends described above, **DLV** also has an SQL3 front-end based on the current working draft of the standard. This front-end translates a subset of SQL3 query expressions to Datalog queries and supports recursive query processing such as the list-of-materials problem, where we have items that consist of other items that again consist of other items and so forth, and have to transform this tree structure into a list of all dependencies.

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

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

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

Example 5 (List of Materials) Consider the canonical list of materials query:

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

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

This query is translated into

```
listofmaterials(A,B) :- consists_of(A,B).
listofmaterials(A,B) :- consists_of(A,C), listofmaterials(C,B).
sql2dl__intern0(A,B) :- listofmaterials(A,B).
```

and the elements of `sql2dl__intern0` are printed in tabular form as the result.

5 USAGE

While **DLV** also comes with a graphical user interface (GUI), the computational engine and its front-ends are implemented as a command-line tool, that is, the system is controlled by command-line options; it reads input from text files whose names are provided on the command-line, and any output is written to the standard output.

In its default mode, **DLV** simply computes all answer sets of the program read from the input files; specific front-ends are invoked by means of command-line options starting with `-F`.

The brave reasoning front-end is invoked by the `-FB` command-line option, while the cautious front-end is invoked by `-FC`.

For the diagnosis front-ends, hypotheses and observations have to be stored in files whose names carry the extensions `.hyp` and `.obs`, respectively, while the program is stored in files with any extension different from these.

The type of diagnostic reasoning (abductive or consistency-based) and the desired minimality criterion (none, single error, subset minimality) is selected by specifying one of the options depicted in Table 4.1 on the command-line.

The SQL front-end, finally, is selected by means of the `-FS` command-line option. Input files whose names end with `.sql` are processed with an SQL parser, all other files can be used to specify "databases" in the form of Datalog facts and non-disjunctive positive rules.

6 THE DLV PROBLEM SOLVING ENGINE

Since this paper focuses mainly on knowledge representation, we cannot provide a detailed description of **DLV**'s system interna, but only give an overview.

Table 4.1 Command-line switches

Option	*Front-end/Mode*
-FB	brave reasoning
-FC	cautious reasoning
-FD	generic abductive diagnosis
-FDsingle	single error abductive diagnosis
-FDmin	subset minimal abductive diagnosis
-FR	"Reiter's" (consistency-based) diagnosis
-FRsingle	single error "Reiter's" diagnosis
-FRmin	subset minimal "Reiter's" diagnosis
-FS	SQL3

More details on the basic algorithms implemented in the system can be found in (Leone et al., 1997; Eiter et al., 1998a; Faber et al., 1999b; Koch and Leone, 1999).

The system architecture of **DLV** is shown in Figure 4.3. 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 all or some answer sets of its input. In addition to various front-ends (described in Section 4), there is a Graphical User Interface (GUI) that provides convenient access both to the system itself and these front-ends.

Ideally, the performance of a system reflects the complexity of the problem at hand, such that "easy" problems (say, those of polynomial complexity) are solved fast, while more time is spent for more difficult problems (like NP-hard problems), and indeed the **DLV** system is designed in this direction. For example, stratified normal programs (which have polynomial complexity) are evaluated fast using techniques from the fields of deductive databases and logic programming. The SQL3 front-end, which only generates such programs, benefits from this and has good performance. Note that in general, that is without syntactical restrictions, it is impossible to detect whether a program uniformly encodes an "easy" (even trivial) problem, since this task is clearly undecidable.

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

Upon startup, the **DLV** 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 takes only linear time and memory.

Using differential (and other advanced) database techniques together with suitable data structures, the *Intelligent Grounding* module then efficiently gen-

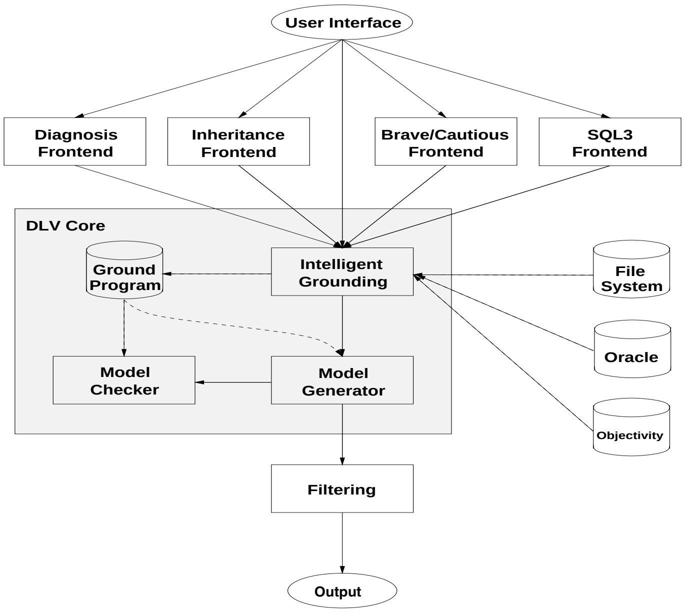

Figure 4.3 The System Architecture of **DLV**

erates a subset of the ground instantiation of that input which has the same answer sets, but is much smaller in general. For example, in case of a stratified program, the IG module already computes the single answer set.

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.

The generation of the answer sets of a program $\mathcal{P}$ relies on a monotonic operator $\mathcal{W}_{\mathcal{P}}$ (Leone et al., 1997) which extends the well-founded operator of (van Gelder et al., 1991) 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 (Leone et al., 1997).

Briefly, the algorithm works as follows: First, $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ (the fixpoint of $\mathcal{W}_{\mathcal{P}}$) is computed, which is contained in every answer set. If $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ is a total model, it is returned as the unique answer set. Otherwise, moving from $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ towards the answer sets, a literal (called *possibly-true literal* in (Leone et al., 1997)), the truth of which allows to infer new atoms, is assumed true. Cleary 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. The computation of the MG proceeds by iteratively applying the disjunctive version of the Fitting operator

Table 4.2 **DLV**'s Progress

Problem instance	*07 '98*	*02 '99*	*04 '99*	*06 '99*	*04 '00*
3COL[a]	> 1000s	26.0s	13.6s	2.0s	0.5s
Hamiltonian Path[b]	> 1000s	> 1000s	620.4s	10.6s	0.3s
Prime implicants[c]	— [z]	20.8s	12.9s	10.0s	0.8s
Strategic Companies[d]	> 1000s	171.6s	140.2s	55.0s	34.9s
Blocksworld P4[e]	> 1000s	> 1000s	> 1000s	31.4s	2.7s
Blocksworld Split P4[f]	> 1000s	> 1000s	272.3s	10.1s	1.3s
Ancestor[g]	47.9s	47.8s	48.3s	47.3s	17.9s

[a]find one coloring on a random graph with 150 nodes and 350 edges
[b]find one Hamiltonian Path on a random graph with 25 nodes and 120 arcs
[c]find all prime implicants on a random 3CNF with 546 clauses and 127 variables
[d]find all strategic sets two randomly chosen companies occur in (150 companies, 450 products)
[e]find one plan of length 9 involving 11 blocks, see (Erdem, 1999)
[f]optimized program for [e]
[g]compute the ancestor relation using double recursion (196 persons and 338 parenthoods)
[z]could not handle input size

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 generated. In the latter case, backtracking is performed; in the former case, the Model Checker is invoked.

The *Model Checker* (MC) verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is well-known to be co-NP-complete (cf. (Eiter et al., 1997b)). However, recent studies (Ben-Eliyahu and Dechter, 1994; Ben-Eliyahu and Palopoli, 1994; Leone et al., 1997) showed that minimal model checking — the hardest part of answer set (or stable model) checking — can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs (Ben-Eliyahu and Dechter, 1994).

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

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

7 ONGOING WORK AND CONCLUSION

As already pointed out, the high expressivity of **DLV**, which allows to express *all* KR problems with Σ_2^P-complexity, comes at the cost of a high worst case complexity, and requires sophisticated algorithms to obtain acceptable performance on average.

Improving the computational engine of **DLV** has been one of the main tasks during the last two years. The **DLV** team has been spending quite some efforts in this direction, and we have no space here to discuss the algorithms and implementation techniques that have been incorporated into the system, see (Eiter et al., 1997c; Eiter et al., 1997d; Eiter et al., 1998b; Eiter et al., 1998a; Faber et al., 1999b; Faber et al., 1999a; Koch and Leone, 1999; Leone et al., 1997). We just report in Table 4.2 the progress that has been achieved during the last two years, by comparing the performance for a few arbitrarily chosen problem instances at various stages of **DLV**'s development (The numbers there are seconds of execution time; the smaller, the better). We also refer to (Erdem, 1998), where some of **DLV**'s improvements are documented as well.

The problems 3COL, Hamiltonian Path and Strategic Companies are encoded using the programs from Section 3 The programs of the other problems are contained in (Eiter et al., 1998b; Faber et al., 1999a; Erdem, 1999). The experiments have been performed on a Pentium II with 400 MHz and 128MB memory, running FreeBSD 3.4. The way the timings were obtained is described more in detail in (Faber and Pfeifer, 1996). The improvements are quite impressive and the **DLV** system is nowadays able to solve medium-sized instances of hard problems in reasonable time.

DLV can (most likely) not compete with specialized graph algorithms, but as the results show, it can solve large instances of the problems in acceptable time. The big advantage of **DLV** is, however, that a program for solving a graph problem might be developed and run rather quickly.

The only other answer set system which is able to evaluate all benchmark programs above is GnT (Janhunen et al., 2000) – a disjunctive extension of Smodels.[7] We did not carry out a detailed performance comparison of the the systems yet, but, roughly, first experimental results show that: (i) **DLV** sensibly outperforms GnT on deductive database benchmarks like Ancestor or Reachability, (ii) on Σ_2^P-complete problems (like Strategic Companies above), **DLV** is on average faster than GnT, (iii) there are NP-complete problems where GnT is faster than **DLV** and vice versa. Comments (i) and (iii) apply also to standard Smodels, which, however, is not able to solve Σ_2^P-complete problems.[8]

It is, at this stage, not clear to which extent the rate of performance improvement in the past can be sustained. Obviously, there will be some natural limit, and increasingly more sophistication has to be put into the system for constant performance gain. To the best of our knowledge, it is unknown how the Σ_2^P problem complexity generally behaves in this respect, and no experience from thorough studies on this subject are reported in the literature.

Further improvements might be gained by refining the algorithms. An approach that seems promising in this respect is the use of refined heuristics for

[7] Actually, the input programs must be slightly modified, mainly because GnT requires that the range of each variable is restricted by a domain predicate.

[8] Note that these experimental results consider only standard features of Answer Set Programming, disregarding special extensions.

pruning the search space of answer set candidates. In particular, the choice of the next "possibly-true literal" (see (Faber et al., 1999b)) considered by the algorithm can be done in a number of ways. Currently, the use of heuristics is at an early stage. Our ongoing and future work includes the development and evaluation of different heuristics.

Apart from heuristics, we are also working on various other tasks. On the system side, possible refinements of the data structures that are used by the computation engine will be explored. Another task is the implementation of weak constraints (Buccafurri et al., 1997; Faber, 1998), which allow for expressing optimization problems such as the well-known traveling salesman problem in a natural way. On the application side, we are currently developing a front-end for declarative planning. Note that planning by using answer sets of logic programs has been proposed in (Subrahmanian and Zaniolo, 1995; Dimopoulos et al., 1997; Turner, 1997; Lifschitz, 1999a; Lifschitz, 1999b). The front-end should support the declarative specification of modest-sized planning problems under incomplete information, such as moves of a mobile robot or the actions taken by an information agent.

The **DLV** system is disseminated in academia, and presumably soon also in industry. It is widely used for educational purposes in courses on Databases and on AI, both in European and American universities. Furthermore, **DLV** is currently under evaluation at CERN, the European Laboratory for Particle Physics located near Geneva in Switzerland and France, 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 in practice.

Acknowledgments

This work has been supported by FWF (Austrian Science Funds) under the projects P11580–MAT and Z29–INF.

We would like to thank Georg Gottlob and all other people who participated in the project: Robert Bihlmeyer, Francesco Buccafurri, Antonella Capalbo, Simona Citrigno, Christoph Koch, Cristinel Mateis, Simona Perri, Axel Polleres, and Francesco Scarcello.

References

Apt, K. R., Blair, H. A., and Walker, A. (1988). Towards a theory of declarative knowledge. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Los Altos, California.

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

Ben-Eliyahu, R. and Dechter, R. (1994). Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87.

Ben-Eliyahu, R. and Palopoli, L. (1994). 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.

Buccafurri, F., Faber, W., and Leone, N. (1999). Disjunctive logic programs with inheritance. In Schreye, D. D., editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP '99)*, pages 79–93, Las Cruces, New Mexico, USA. The MIT Press.

Buccafurri, F., Leone, N., and Rullo, P. (1997). Strong and weak constraints in disjunctive Datalog. In Dix, J., Furbach, U., and Nerode, A., 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. Springer Verlag.

Cadoli, M., Eiter, T., and Gottlob, G. (1997). Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463.

Cadoli, M., Palopoli, L., Schaerf, A., and Vasile, D. (1999). NP-SPEC: An executable specification language for solving all problems in NP. In Gupta, G., editor, *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 16–30. Springer.

Chen, W. and Warren, D. S. (1996). Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757.

Cholewiński, P., Marek, V. W., Mikitiuk, A., and Truszczyński, M. (1999). Computing with default logic. *Artificial Intelligence*, 112(2–3):105–147.

Cholewiński, P., Marek, V. W., and Truszczyński, M. (1996). Default reasoning system DeReS. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pages 518–528. Cambridge, Massachusetts, USA. Morgan Kaufmann Publishers.

Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the European Conference on Planning 1997 (ECP-97)*, pages 169–181. Springer Verlag.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. (1999). The diagnosis frontend of the `DLV` system. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111.

Eiter, T., Gottlob, G., and Leone, N. (1997a). Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177.

Eiter, T., Gottlob, G., and Mannila, H. (1994). Adding disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT SIGMOD-SIGART Symposium on Principles of Database Systems (PODS-94)*, pages 267–278. ACM Press.

Eiter, T., Gottlob, G., and Mannila, H. (1997b). Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):315–363.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1997c). A deductive system for nonmonotonic reasoning. In Dix, J., Furbach, U., and Nerode, A., editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin. Springer.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1997d). The architecture of a disjunctive deductive database system. In Falaschi, M., Navarro, M., and Policriti, A., editors, *Proceedings Joint Conference on Declarative Programming (APPIA-GULP-PRODE '97)*, pages 141–151.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1998a). Progress report on the disjunctive deductive database system `DLV`. In Andreasen, T., Christiansen, H., and Larsen, H. L., editors, *Proceedings International Conference on Flexible Query Answering Systems (FQAS'98)*, number 1495 in Lecture Notes in AI (LNAI), pages 148–163, Roskilde University, Denmark. Springer.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1998b). The KR system `DLV`: Progress report, comparisons and benchmarks. In Cohn, A. G., Schubert, L., and Shapiro, S. C., editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers.

Erdem, E. (1999). Applications of logic programming to planning: Computational experiments. Unpublished draft.
`http://www.cs.utexas.edu/users/esra/papers.html`.

Erdem, E. (since 1998). Website for applications of logic programming to planning: Computational experiments.
`<URL:http://www.cs.utexas.edu/users/esra/experiments/experiments.html>`.

Faber, W. (1998). Disjunctive Datalog with strong and weak constraints: Representational and computational issues. Master's thesis, Institut für Informationssysteme, Technische Universität Wien.

Faber, W. (1999). DLV homepage. <URL:http://www.dbai.tuwien.ac.at/proj/dlv/inheritance/>.

Faber, W., Leone, N., Mateis, C., and Pfeifer, G. (1999a). Using database optimization techniques for monmonotonic reasoning. In Committee, I. O., editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan.

Faber, W., Leone, N., and Pfeifer, G. (1999b). Pushing goal derivation in DLP computations. In Gelfond, M., Leone, N., and Pfeifer, G., editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 177–191, El Paso, Texas, USA. Springer Verlag.

Faber, W. and Pfeifer, G. (since 1996). dlv homepage. <URL:http://www.dbai.tuwien.ac.at/proj/dlv/>.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385.

Janhunen, T., Niemela, I., Simons, P., and You, J.-H. (2000). Partiality and disjunctions in stable model semantics. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 411–419, Breckenridge, Colorado, USA. Morgan Kaufmann Publishers, Inc.

Koch, C. (1999). A simple query facility for the objectivity/db persistent object manager. Unpublished draft, available via the author's homepage <URL:http://home.cern.ch/~chkoch/>.

Koch, C. and Leone, N. (1999). Stable model checking made easy. In Dean, T., editor, *The International Joint Conferences on Artificial Intelligence (IJCAI) 1999*, pages 70–75, Stockholm, Sweden. Morgan Kaufmann Publishers.

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

Lifschitz, V. (1996). Foundations of logic programming. In Brewka, G., editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford.

Lifschitz, V. (1999a). Action languages, answer sets and planning. In Apt, K., Marek, V. W., Truszczyński, M., and Warren, D. S., editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 357–373. Springer Verlag.

Lifschitz, V. (1999b). Answer set planning. In Schreye, D. D., editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP '99)*, pages 23–37, Las Cruces, New Mexico, USA. The MIT Press.

Lifschitz, V. and Turner, H. (1994). Splitting a logic program. In Van Hentenryck, P., editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy. MIT Press.

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

Marek, V. W. and Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In Apt, K., Marek, V. W., Truszczyński, M., and Warren, D. S., editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag.

McCain, N. (1999). The clausal calculator homepage. `<URL:http://www.cs.utexas.edu/users/mccain/cc/>`.

Minker, J. (1994). Overview of disjunctive logic programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24.

Niemelä, I. (1999). Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273.

Niemelä, I. and Simons, P. (1996). Efficient implementation of the well-founded and stable model semantics. In Maher, M. J., editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96)*, pages 289–303, Bonn, Germany. MIT Press.

Niemelä, I. and Simons, P. (1997). Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Dix, J., Furbach, U., and Nerode, A., 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. Springer Verlag.

Poole, D. (1989). Explanation and prediction: An architecture for default and abductive reasoning. *Computational Intelligence*, 5(1):97–110.

Przymusinski, T. C. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424.

Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., and Freire, J. (1997). XSB: A system for efficiently computing well-founded semantics. In Dix, J., Furbach, U., and Nerode, A., 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. Springer Verlag.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95.

Seipel, D. and Thöne, H. (1994). DisLog – A system for reasoning in disjunctive deductive databases. In Olivé, A., editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politecnica de Catalunya (UPC).

Subrahmanian, V. and Zaniolo, C. (1995). Relating stable models and AI planning domains. In Sterling, L., editor, *Proceedings of the* 12^{th} *International Conference on Logic Programming*, pages 233–247, Tokyo, Japan. MIT Press.

Turner, H. (1997). Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31(1–3):245–298.

van Gelder, A., Ross, K., and Schlipf, J. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650.

Winston, P. H. (1992). *Artificial Intelligence*. Addison Wesley.