Progress Report on the Disjunctive Deductive Database System dlv [*]

Thomas Eiter[b], Nicola Leone[a] [**], Cristinel Mateis[a],
Gerald Pfeifer[a], and Francesco Scarcello[c]

[a]Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria

[b]Institut für Informatik, Universität Gießen
Arndtstrasse 2, D-35392 Gießen

[c]ISI-CNR, c/o DEIS Universitá della Calabria
I-87030 Rende, Italy

{*eiter,leone,mateis,pfeifer,scarcell*}@*dbai.tuwien.ac.at*

Abstract. `dlv` is a deductive database system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms. The system has been developed since the end of 1996 at Technische Universität Wien in an ongoing project funded by the Austrian Science Funds (FWF).
Recent comparisons have shown that `dlv` is nowadays a state-of-the-art implementation of disjunctive logic programming. A major strength of dlv are its advanced knowledge modelling features. Its kernel language extends disjunctive logic programming by strong negation (*a la* Gelfond and Lifschitz) and integrity constraints; furthermore, front-ends for the database language SQL3 and for diagnostic reasoning are available. Suitable interfaces allow `dlv` users to utilize base relations which are stored in external commercial database systems.
This paper provides an overview of the `dlv` system and describes recent advances in its implementation. In particular, the recent implementation of incremental techniques for program evaluation, as well as the use of relational join optimization strategies appear particularly relevant for deductive database applications. These techniques are suitably extended in `dlv` for the efficient instantiation of nonmonotonic programs.
Benchmarks on problems from different domains are also reported to give a feeling of the current system performance.

1 Introduction

Deductive databases extend relational database systems by the inferential power of logic programming. Their study has been one of the major activities of database researchers

[*] Partially supported by *FWF (Austrian Science Funds)* under the project P11580-MAT "A Query System for Disjunctive Deductive Databases" and by the Italian MURST under the project for Italia-Austria cooperation *Advanced Formalisms and Systems for Nonmonotonic Reasoning*.

[**] Please address correspondence to this author.

in the second half of the eighties. Besides important theoretical results, this study has led to the implementation of a number of deductive database systems supporting logic programming and its extensions as the query language [2, 4, 15, 20, 22]. Recently, the idea of incorporating disjunction in the deductive database languages has stimulated a renewed interest in this area, since deductive databases with disjunction, called *Disjunctive Deductive Databases (DDDBs)*, seem to be well suited to perform nonmonotonic reasoning tasks which are strongly needed in the field of Artificial Intelligence. Thus, during the last years, much research has been done concerning semantics and complexity of Disjunctive Deductive Databases. Interesting studies on the expressive power of DDDBs [8] have also shown that some concrete real world problems cannot be represented by ($\vee$-free) deductive databases, while they can be represented by DDDBs. Among these problems we can mention Strategic Companies from the business domain (that we describe in Section 2.1), the famous Travelling Salesman Problem from the field of optimization, and even the problem of finding a minimum size key of a relation (i.e., a key with minimum number of attributes) from the domain of databases. Nevertheless, while DDDBs are now generally considered a powerful tool for common-sense reasoning and knowledge representation, there has been a shortage of actual (let alone efficient) implementations ([24, 1]).

This paper focuses on the DDDB system `dlv` (datalog with disjunction) currently developed at TU Wien in the *FWF project P11580-MAT "A Query System for Disjunctive Deductive Databases"*. A major strength of `dlv` are the advanced knowledge representation languages it supports. Its kernel language extends disjunctive logic programming by strong negation (*a* la Gelfond and Lifschitz) and integrity constraints; furthermore, front-ends for the database language SQL3 and for diagnostic reasoning are available. Suitable interfaces allow `dlv` users to access relations stored in external commercial database systems, so that `dlv` can be also used as an advanced interface to relational databases.

In Section 2, we will describe the languages supported by the system and address knowledge representation issues, showing how very sophisticated (and computationally hard) queries can be easily yet elegantly specified in `dlv`.

In Section 3 we will address implementation issues. In particular, we will first present the overall architecture of `dlv`. We will then describe some optimization techniques that have been recently incorporated in `dlv`, and finally report some benchmarks to give a feeling of the current system performance.

A previous version of the system, that does not include the optimization techniques presented here, has been described in [9, 6] and compared to other systems in [10]. For up-to-date information and a download of `dlv`, please visit the project homepage at `http://www.dbai.tuwien.ac.at/proj/dlv/`.

2 The Languages of `dlv`

2.1 Extended Disjunctive Logic Programming

This is the native interface of the system and offers the full power of the internal language of `dlv`: function-free disjunctive logic programs extended by allowing both integrity constraints and strong negation [11].

The presence of these constructs makes our language well-suited to easily represent a wide range of problems in a natural and highly declarative way.

A `dlv` rule has the following form:

$$L_1 \vee \ldots \vee L_k :- L_{k+1}, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$$

where $1 \leq k \leq m \leq n$ and each L_i is a literal, i.e., an atom $a(t_1, \ldots, t_o)$ or its "classical" negation $\tilde{a}(t_1, \ldots, t_o)$, a being the predicate name of that atom, and each t_i $(1 \leq i \leq o)$ being either a constant or a variable. The convention on names is the one of Prolog.

An *(integrity) constraint* is like a rule with $k = 0$ and $n \geq 1$. A `dlv` program P is a pair $< R, IC >$, where R is a set of `dlv` rules and IC a set of constraints. A *stable model* of P is any consistent answer set of R [11] satisfying all constraints in IC.

We next present some sample problems, drawn from different domains, and their encodings in the native language of `dlv`. This should give an idea about the use of the language for knowledge representation and reasoning. Some of these problems will be used as benchmarks to evaluate the performances of the system.

Example 1 (Reachability). $connected(\_, \_)$, find for each city C in the database the cities which are reachable from C.

This is a classical deductive database example; we provide here the single recursive version.

```
reaches(X,Y) :- connected(X,Y).
reaches(X,Y) :- reaches(X,U), connected(U,Y).
```

Example 2 (Graph 3-colorability). Given a graph, represented by facts of the form `node(_)` and `edge(_,_)`, assign each node one of three colors such that no two adjacent nodes have the same color.

3-colorability is a classical NP-complete problem. In *dlv*, it can be encoded in a very easy and natural way by means of disjunction and constraints:

```
col(X,r) ∨ col(X,g) ∨ col(X,b) :- node(X).
:- edge(X,Y), col(X,C), col(Y,C).
```

The disjunctive rule nondeterministically chooses a color for each node X in the graph; the constraint enforces that the choices are legal.

Example 3 (Prime Implicants). Find the prime implicants of a propositional CNF formula $\Phi = \bigwedge_{i=1}^{n}(d_{i1} \vee \ldots \vee d_{ic_i})$, where the $d_{i,j}$ are literals over the variables $x_1, \ldots, x_m$.

Recall that a *prime implicant* of Φ is a consistent conjunction I of literals such that $I \models \Phi$ and for no subformula I' of I we have $I' \models \Phi$. As easily seen, a prime implicant exists if and only if Φ is satisfiable. Thus, finding a prime implicant of Φ subsumes the satisfiability problem for Φ.

The representation in `dlv` is natural and simple. Here, true negation in the language plays a key role. We may use a literal translation of Φ into a `dlv` program P:

$$\mathtt{d_{11}} \vee \ldots \vee \mathtt{d_{1c_1}}. \qquad \ldots \qquad \mathtt{d_{n1}} \vee \ldots \vee \mathtt{d_{nc_n}}.$$

It is easy to see that the stable models of P correspond 1-1 to the prime implicants of Φ.

Example 4 (Strategic Companies). The strategic companies problem is from [3]; it is, to the best of our knowledge, the only Σ_2^P-complete KR problem from the business domain. No experimental results for any Σ_2^P-complete KR problems are known.

Briefly, a holding owns companies, each of which produces some goods. Moreover, several companies may have jointly control over another company. Now, some companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding after the transaction. A company is *strategic*, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. Those sets are expressed by the following natural program:

```
strategic(C1) v strategic(C2) :-
            produced_by(P,C1,C2).
strategic(C) :-
            controlled_by(C,C1,C2,C3),
            strategic(C1),
            strategic(C2),
            strategic(C3).
```

Here `strategic`(C) means that C is strategic, `produced_by`$(P, C1, C2)$ that product P is produced by companies $C1$ and $C2$, and `controlled_by`$(C, C1, C2, C3)$ that C is jointly controlled by $C1, C2$ and $C3$; we have adopted here from [3] that each product is produced by at most two companies and each company may be controlled by consortia of at most three other companies each.

The problem is to find out the set of all strategic companies (i.e., under brave reasoning, for which C the fact `strategic`(C) is true).

Note that this problem can not be expressed by a fixed normal logic program *uniformly on all collections* produced_by$(p, c1, c2)$ and `controlled_by`$(c, c1, c2, c3)$ (unless NP $= \Sigma_2^P$, an unlikely event).

2.2 Brave and Cautious Reasoning

The frontends for brave and cautious reasoning are an extension of the native interface described above. In addition to the program, the user also specifies a query, which is essentially the body of a rule followed by a question mark:

$$b_1, \ldots, \; b_m, \; \texttt{not} \; b_{m+1}, \ldots, \; \texttt{not} \; b_n \; ?$$

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

The brave reasoning frontend is invoked by the `-FB` command line option. If the query Q evaluates to true in at least one stable model M, then the system replies: "Q is bravely true, evidenced by M", i.e., returns a stable model that witnesses this fact; otherwise the answer is "Q is bravely false".

Similarly, the cautious frontend is invoked by `-FC`, and the system answers "Q is cautiously true" if and only if Q is true in all stable models. In case of a negative answer, we get: "Q is cautiously false, evidenced by M", where M is a stable model witnessing the falsity of Q.

2.3 SQL3

This frontend provides the possibility to use (a subset of) SQL3 query expressions, that are internally translated into the native system language. Since the SQL3 standard has not been completed yet, we resorted to the current working draft [25]. The system automatically invokes the SQL frontend for input files whose names carry the extension `.sql`.

First of all, since there are no column names in datalog, we have to include a construct that creates a connection between the parameters of a datalog predicate and the column names of an SQL table:

```
DATALOG SCHEMA { Relationname({Columnname} [,...]) } [,...];
```

The following grammar describes the query expressions which currently can be handled by the following SQL frontend:

```
[ WITH [RECURSIVE]
  { Relationname [ ( { Columnname } [, ... ] ) ]
    AS ( QueryExpression ) }
  [, ... ]
]

{ SELECT { [(Cor-)Relationname .] Columnname } [,...]
  FROM { Relationname [ [AS] Correlationname] } [,...]
  [ WHERE { [(Cor-)Relationname .] Columnname =
            [(Cor-)Relationname .] Columnname } [AND...]]]}
[ UNION ... ] ;
```

Here, "{ *item* } [*connective*...]" represents a list of *item*s, separated by *connective*s; "[*expression*]" means that *expression* is optional; words in capital letters are keywords; `QueryExpression` refers to a construct as described above (without the trailing semicolon).

Observe that query expressions have in general two parts: A definition part (the WITH clause), and a query part.

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 as the result. Here the first two rules represent the definition part and the last rule corresponds to the query part of the query expression.

(A rule and an internal predicate name for the query part have to be generated because in general queries may consist of several parts, connected by set operators like UNION.)

We are currently implementing furher features of the languages defined in [25], in particular set operators EXCEPT and INTERSECT, explicit joins, and additional comparison predicates like IN, $<$ and $>$. (To support the latter, we will also add arithmetic built-in predicates to our language.)

2.4 Diagnostic Reasoning

`dlv` provides two kind of diagnosis frontends: the *abductive diagnosis* frontend [21, 7], and the *Reiter's diagnosis* frontend [23].

Abduction, first studied by Peirce [18], is an important kind of reasoning, having wide applicability in different areas of computer science; in particular, it has been recognized as an important principle of common-sense reasoning.

Abductive diagnosis deals with the problem of finding an explanation (diagnosis) for observations (symptoms), based on a theory, which is in our framework represented by a logic program as in [13, 12]. Roughly speaking, abduction is an inverse of modus ponens: Given the clause $a \leftarrow b$ and the observation a, abduction concludes b as a possible explanation of a. Following [7, 13, 12], a *diagnostic problem* can be formally described as a tuple $\mathbf{P} = \langle Hyp, Obs, T \rangle$, where Hyp is a set of ground atoms (called *hypotheses*), Obs is a set of ground literals (called *manifestations* or *observations*), and T is a logic program. An *explanation (diagnosis)* for $\mathbf{P}$ is a subset $E \subseteq Hyp$ which satisfies the following property:

$T \cup E$ implies Obs under brave reasoning, i.e., there exists a stable model M of $T \cup E$, where all literals in Obs are true.

In general, a diagnostic problem may have no, a single, or several explanations. In accordance with Occam's principle of parsimony [19], which states that from two explanations the simpler is preferable, some minimality criterion is usually imposed on diagnoses. A widely used criterion is *Subset Minimality*, stating that a solution A is preferable to a solution B if $A \subset B$. According to this criterion, the acceptable solutions of $\mathbf{P}$ are restricted to the minimal explanations (they are considered to the most likely ones). Another criterion works under the assumption of *Single Error* and considers only elementary diagnoses (which consist of a single atom).

Example 6 (Keys Finding). Consider a relation scheme consisting of attributes *Class, Teacher, Hour, Room, Student* on which the following functional dependencies hold:

$$\begin{array}{c} Teacher \leftarrow Class \\ Class \leftarrow Hour, Room \\ Room \leftarrow Hour, Teacher \\ Room \leftarrow Hour, Student \end{array}$$

Finding the keys of this relation can be modeled as an abductibe problem as follows. Consider the abductive diagnostic problem $\mathbf{P}_{keys} = \langle H, O, T_{keys} \rangle$, where the theory T_{keys} is:

$$\begin{array}{l} Teacher :- Class \\ Class :- Hour, Room \\ Room :- Hour, Teacher \\ Room :- Hour, Student \\ Cover :- Teacher, Class, Hour, Room, Student; \end{array}$$

it encodes the functional dependencies and states that we have covered the relation if *all* attributes have been (transitively) implied.
The hypotheses are the attributes of the relation scheme:

$$H = \{Class, Teacher, Hour, Room, Student\}.$$

and, the set of observations is $O = \{cover\}$.

We have that every diagnosis is a covering of the relation. In particular, every minimal diagnosis is a key and vice versa. $\{Hour, Student\}$ is the only minimal diagnosis which is indeed the only key of the relation. Possible single error diagnoses would correspond to single attribute keys.

Our frontend for *abductive diagnosis* currently supports three different modes: general diagnosis, where all diagnoses are computed, subset minimal diagnosis[1], and single failure diagnosis. These modes are invoked by the command line options `-FD`, `-FDmin` and `-FDsingle`, respectively.

[1] For positive non-disjunctive theories only.

The diagnostic theory obeys the syntax described in Section 2.1. Hypothesis (resp. observations) are lists of atoms (resp. literals) separated by a dot (`.`) and are stored in files whose names carry the extension `.hyp` (resp. `.obs`).

Basically, the diagnostic frontend works as follows: After reading the input, disjunctive rules for guessing possible diagnosis candidates are generated from the hypotheses, while the observations become constraints that forbid the generation of inconsistent diagnoses. In subset-minimal diagnosis case, further rules are added for pruning non-minimal solutions. Finally, the grounding (and subsequently the MG etc.) are invoked, and for each reported stable model, the corresponding diagnosis is returned. The frontend for *consistency-based diagnosis* as described by Reiter [23] works similarly. The command line options are `-FR`, `-FRmin` and `-FRsingle`, respectively.

3 Implementation Issues

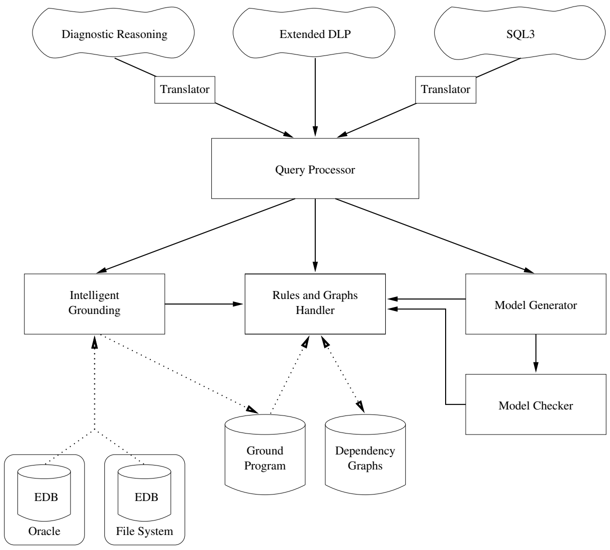


Fig. 1. Overall Architecture of the System

3.1 System Architecture

An outline of the general architecture of our system is depicted in Figure 1.

At the heart of the system lies the *Query Processor*. It controls the execution of the entire system and – in collaboration with the integrated frontends – it performs some pre-processing on the input and post-processing on the generated models, respectively.

Upon startup, the Query Processor reads the – possibly non-ground – input program and hands it over to the *Rules and Graphs Handler*, which splits it into subprograms.

Together with relational database tables, provided by an Oracle database or ASCII text files, the subprograms are then submitted to the *Intelligent Grounding Module*, which efficiently generates a subset of the grounded input program that has exactly the same stable models as the full program, but is much smaller in general.

The Query Processor then again invokes the Rules and Graphs Handler, which generates two partitionings of the ground(ed) program. They are used by the *Model Generator* (MG) and the *Model Checker*, respectively, and enable a modular evaluation of the program. This often yields a tremendous speedup.

Finally, the Model Generator is started. It generates one candidate for a stable model and invokes the Model Checker to verify whether it is indeed a stable model. Upon success, control is returned to the Query Processor, which performs post-processing and possibly invokes the MG to look for further models.

More details on the Intelligent Grounding and the Model Checker can be found in [16, 9].

3.2 Efficient Instantiation

In this section, we present the intelligent grounding (IG) module of the `dlv` system. The aim of IG is twofold:

(i) to evaluate ($\vee$-free) stratified programs components, and
(ii) to generate the instantiation of disjunctive or unstratified components (if the input program is disjunctive or unstratified)

In order to efficiently evaluate stratified programs (components), we use two principal techniques from classical deductive databases and from relational database optimization theory. The one is an improved version of the generalized semi-naive evaluation technique is linear and non-linear recursive rules. The other obtains an efficient evaluation of rules bodies (especially in case of large relations) using relational join optimization techniques (for more, see the end of this section).

If the input program is normal (i.e., $\vee$-free) and stratified, the IG completely evaluates the program and no further module is employed after the grounding; the program has a single stable model, namely the set of atoms returned by the instantiation procedure.

If the input program is either disjunctive or unstratified, then the instantiation procedure does not completely evaluate the program. However, the optimisation techniques mentioned above are useful to efficiently generate the instantiation of the nonmonotonic part of the program. Two aspects are crucial for the instantiation:

(a) the number of generated ground rules, and
(b) the time needed to generate the grounding.

The size of the grounding generated is important because it strongly influences the computation time of the other modules of the system. A slower instantiation procedure generating a smaller grounding may be preferable to a faster one that generates a large grounding. However, the time needed by the former can not be ignored; otherwise, we could not really have a gain in computation time.

The main reason of large groundings, even for small-sized input programs, is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in B_{LP}, which leads to combinatorial explosion. However, in a reasonable program semantics, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules. We present an algorithm which generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$. Denote for a rule r by $H(r)$, $B^+(r)$, and $B^-(r)$ its head, `not`-free part and `not`-part, respectively. We associate to $\mathcal{P}$ the dependency graph $G_\mathcal{P}$. The nodes of $G_\mathcal{P}$ are the predicates appearing in the head of the rules with non-empty body, and $G_\mathcal{P}$ has an arc from a node q to a node p iff there exists a rule $r \in \mathcal{P}$ such that $p \in H(r)$ and $q \in B^+(r)$.

A simple way to optimize (a) is to employ the collapsed dependency graph $\widehat{G_\mathcal{P}}$ associated to $\mathcal{P}$ which results from $G_\mathcal{P}$ by collapsing each maximal strongly connected component into a single node. Note that every node of $\widehat{G_\mathcal{P}}$ is a set of nodes of $G_\mathcal{P}$.

The optimization of (b) is realized through the techniques of the relational and deductive database optimization mentioned above.

Let $\mathcal{P}$ be a non-ground program and C a node of the graph $\widehat{G_\mathcal{P}}$. We denote by $recursive\_rules_\mathcal{P}(C)$ the set of the rules r of $\mathcal{P}$ s.t. predicates from C occur both in $H(r)$ and in $B^+(r)$, and by $exit\_rules_\mathcal{P}(C)$ the remaining set of rules r of $\mathcal{P}$ having a predicate from C in $H(r)$. Moreover, we say that a rule r of $\mathcal{P}$ is *total* if either (i) r is a fact, or (ii) r is normal stratified and every body literal is defined only by total rules. An atom is total if all rules defining it are total.

Recall that we assume that $\mathcal{P}$ is *safe*, i.e. all variables of a rule r appear in $B^+(r)$. Consequently, in order to instantiate a rule r, we merely have to instantiate $B^+(r)$, which uniquely determines the instance of r. We define the grounding of r w.r.t. a set of ground atoms $NF \subseteq B_{LP}$, denoted by $ground(r, NF)$, as the set of ground instances r' of r s.t. $B^+(r') \subseteq NF$. The set $ground(r, NF)$ is computed by the function *Evaluate* which is described at the end of this section.

Figure 2 outlines the algorithm *Instantiate*, which computes a ground program $\Pi \cup T$, where Π is the set of ground rules and T is the set of ground atoms derived from $\mathcal{P}$ (the ground rules with an empty body), having the same stable models as $\mathcal{P}$. $EDB_\mathcal{P}$ and $IDB_\mathcal{P}$ denote the extensional and intensional database of $\mathcal{P}$, respectively.

Initially, *Instantiate* sets $NF = EDB_\mathcal{P}$, $T = EDB_\mathcal{P}$ and $\Pi = \emptyset$. Then, it removes a node C from $\widehat{G_\mathcal{P}}$ which has no incoming arc (i.e., a source), and generates all instances r' of rules r defining predicates in C. This is done by calls to the Procedure *InstantiateRule*. However, the procedure does not generate <u>all</u> ground instances; rather, it generates only rules r' in $ground(r, NF)$ such that every negative total literal in $B^-(r')$ is true w.r.t. T. First, we add $H(r')$ to NF because each atom in $H(r')$ can possibly be derived. We then remove all positive literals (all negative total literals) in T from $B^+(r')$ (from $B^-(r')$). Finally, if the head of r' is disjunction-free and its body became empty after simplifications, then $H(r')$ is inserted into T, otherwise the simplified r' is added to Π.

In order to compute such an r', the function *Evaluate* proceeds by matching the atoms in $B^+(r)$ one by one with atoms in NF and binding the free variables accordingly in each step, as in the case of the relational join operation. If $r \in exit\_rules_\mathcal{P}(C)$, the set

Δ*NF* is irrelevant. If r is a linear recursive rule, the semi-naive optimization technique is used and the recursive body atom is matched only with atoms in Δ*NF*; all non-recursive atoms are matched with atoms in *NF*. If r is a non-linear recursive rule, an improved generalized semi-naive technique is used.

An efficient heuristics is to start with positive literals whose predicate occurs infrequently in *NF* (Δ*NF*) and whose variables we find in several body literals. Therefore, before starting the matching of the atoms in $B^+(r)$ one by one, we first order the positive literals of the body by the increasing cardinality of their ground occurrences in *NF* (Δ*NF*) and by the decreasing number of their shared variables. The positive literals with unique variables are placed at the end of the re-order rule body, even if the cardinalities of their ground occurrences in *NF* (Δ*NF*) are small. This is because in this case, the join operation with the rest of the body literals is equivalent to their cartesian product.

We describe next how the function *Evaluate* proceeds when r is an exit rule or a linear recursive rule.

At the i-th step, all literals L_j, $1 \leq j < i$, are already matched and we try to match the i-th body literal L_i. Note that some variables of L_i could already be bounded, due to the previous steps. There are two possibilities: (i) L_i can be matched with some atom in *NF* (if $L_i \notin C$) or in Δ*NF* (if $L_i \in C$). If L_i is not the last body literal, we compute the matching of L_i and try to match the literal L_{i+1}. If L_i is the last body literal, we add the new ground instance of r to $ground(r, NF)$ and try to match L_i with another atom. (ii) L_i can not be matched with any atom in *NF* (if $L_i \notin C$) or in Δ*NF* (if $L_i \in C$). If L_i is the first body literal ($i = 1$), no further ground instance of r can be derived and the function *Evaluate* exits and returns the set $ground(r, NF)$. If $i > 1$, we backtrack to the previous literal L_{i-1} and try to match it with another atom in *NF* (if $L_{i-1} \notin C$) or in Δ*NF* (if $L_{i-1} \in C$).

If r is a non-linear recursive rule, the function *Evaluate* proceeds in a similar way. We need to mark one recursive body literal at a time. Each time the matching of the first body literal fails (in the case of the exit rule or linear recursive rule, the function would exit and return the set $ground(r, NF)$), we unmark the current marked recursive body literal, we mark the next recursive body literal and the same steps as in the case of exit rule or linear recursive rule are followed, with some differences: (i) the marked recursive body literal can be matched only with atoms in Δ*NF*, (ii) the recursive body literals left of the marked recursive body literal can be matched only with atoms in *NF* $-$ Δ*NF*, and (iii) the recursive body literals right of the marked recursive body literal can be matched only with atoms in *NF*. The classical generalized semi-naive technique makes no difference between the recursive body literals left or right of the marked recursive body literal, and it therefore generates duplicated ground instances. Our improvement avoids to generate the same ground rule more than once. The function *Evaluate* exits only when the matching of the first body literal fails and there is no other recursive body literal to be marked. For efficiency reasons, first non-recursive rules are instantiated once and for all. Then, the recursive rules are repeatedly instantiated until *NF* remains unchanged.

After that, the next source is processed until $\widehat{G_{\mathcal{P}}}$ becomes empty. Each time we pick up a source C from $\widehat{G_{\mathcal{P}}}$ to be processed, and then all possible derivable ground instances of C are generated once and for all at this step, by using the ground instances of the sources processed previously. In this way, we optimize (a), i.e. we generate only ground rules whose head contains atoms which can possibly be derived from $\mathcal{P}$.

Note that if $\mathcal{P}$ is a normal ($\vee$-free) stratified program, the grounding is empty because the body of all ground rules is empty and their head atom is added to *T*.

Procedure *Instantiate*($\mathcal{P}$: Safe_Program; **var** Π: GroundProgram; **var** *T*: SetOfAtoms)
var C: SetOfPredicates; *NF*, *NF*1, Δ*NF*: SetOfAtoms;
begin
    $NF := EDB_{\mathcal{P}}$; $T := EDB_{\mathcal{P}}$; $\Pi := \emptyset$;
    $\widehat{G_{\mathcal{P}}}$:= collapsed dependency graph of $\mathcal{P}$;
    while $\widehat{G_{\mathcal{P}}} \neq \emptyset$ **do**
        Remove a node C of $\widehat{G_{\mathcal{P}}}$ without incoming edges;
        $NF1 := NF$;
        for each $r \in exit\_rules_{\mathcal{P}}(C)$ **do**
            InstantiateRule$(\mathcal{P}, r, \emptyset, NF, T, \Pi)$;
        $\Delta NF := NF - NF1$;
        repeat
            $NF1 := NF$;
            for each $r \in recursive\_rules_{\mathcal{P}}(C)$ **do**
                InstantiateRule$(\mathcal{P}, r, \Delta NF, NF, T, \Pi)$;
            $\Delta NF := NF - NF1$;
        until $\Delta NF = \emptyset$
    end_while
end_function;

Procedure *InstantiateRule*($\mathcal{P}$: Safe_Program; r: Rule; Δ*NF*: SetOfAtoms;
    var *NF*, *T*: SetOfAtoms;
    var Π: GroundProgram)
var H : SetOfAtoms; B^+, B^-: SetOfLiterals;
begin
    for each instance $H \leftarrow B^+, B^-$ of r in *Evaluate*$(r, \Delta NF, NF)$ **do**
        if $(\neg.B^- \cap T = \emptyset) \wedge (H \cap T = \emptyset)$ **then**
            $NF := NF \cup H$;
            Remove all positive literals in *T* from B^+;
            Remove all negative literals $\neg\, q$ from B^- s.t. q is total and $q \notin T$;
            if $(B^+ = \emptyset) \wedge (|H| = 1)$ **then**
                $T := T \cup H$
            else
                $\Pi := \Pi \cup \{H \leftarrow B^+, B^-\}$
            end_if
        end_if
end_procedure;

Fig. 2. Basic algorithm for computation of the (simplified) instantiated program

Example 7. Consider the following program $\mathcal{P}$, where a is an EDB predicate

$$p(1,2) \vee p(2,3) \leftarrow \qquad q(X) \vee q(Z) \leftarrow p(X,Y), p(Y,Z), \neg t(Y)$$
$$t(X) \leftarrow a(X) \qquad t(X) \leftarrow q(X), p(Y,X)$$

and suppose $EDB_{\mathcal{P}} = \{a(2)\}$. Then, *Instantiate* computes the following ground program Π of $\mathcal{P}$:

$$\begin{array}{rr} p(1,2) \vee p(2,3) \leftarrow & \quad q(1) \vee q(3) \leftarrow p(1,2), p(2,3), \neg t(2) \\ t(2) \leftarrow & t(3) \leftarrow q(3), p(2,3) \end{array}$$

Evaluation of node $\{\mathrm{p}\}$ yields the upper left rule of Π, and *NF* $= \{a(2), p(1,2), p(2,3)\}$. We then evaluate the node $\{q\}$ and get the upper right rule of Π, while *NF* becomes $\{a(2), p(1,2), p(2,3), q(1), q(3)\}$. Finally, we consider the node $\{t\}$. The rule $t(X) \leftarrow a(X)$ yields $t(2) \leftarrow$ and the rule $t(X) \leftarrow q(X), p(Y,X)$ yields $t(3) \leftarrow q(3), p(2,3)$.

Note that $ground(\mathcal{P})$ contains 1+ 3 + 27 + 9 = 40 instances of the rules, while *Instantiate* generates only 4 rules. □

Theorem 1. *Let $\mathcal{P}$ be a safe program, and Π, T be the output parameters returned by* Instantiate*($\mathcal{P}, \Pi, \mathrm{T}$). Then*

1. *$\mathcal{P}$ and $\Pi \cup \mathrm{T}$ have exactly the same stable models;*
2. *if $\mathcal{P}$ is a $\vee$-free stratified program, then $\Pi = \emptyset$ (and T is the stable model of $\mathcal{P}$).*

3.3 Some Benchmarks

In this section we report some experimental results on the `dlv` system. Running times are taken over benchmark problems from different domains and should give a feeling of the current performance of the system. The problems we considered are: Reachability (Example 1), 3-Colorability (Example 2) and Strategic Companies (Example 4).

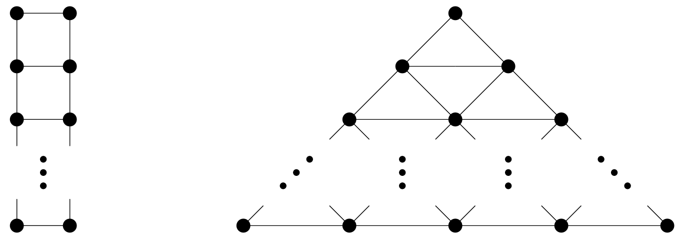
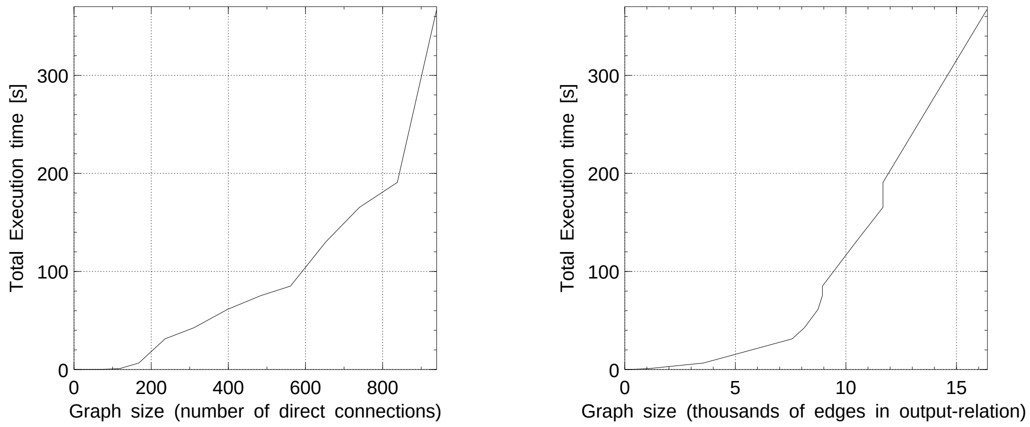


Fig. 3. Ladder Graph (left) and Simplex Graph (right)

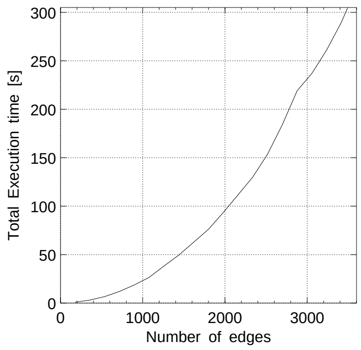
The results of our experiments are displayed in Figure 4. They were run on a SUN Ultra 1/143 under Solaris 2.6 using version 2.91.14 (980315) of the egcs C++ compiler.

For **Reachability**, we used a map of 127 cities in the US, as provided by the Stanford Graphbase ([14]). Instances have been generated by varying a parameter k; in the instance for k a pairs of cities with distance less than k are directly connected.

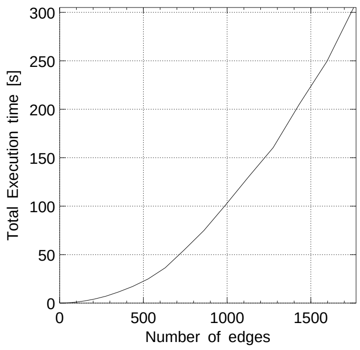
The results are displayed in two upmost tables of Figure 4. The x-axis of the left diagram shows the number n of direct connections among cities (i.e., the size – number



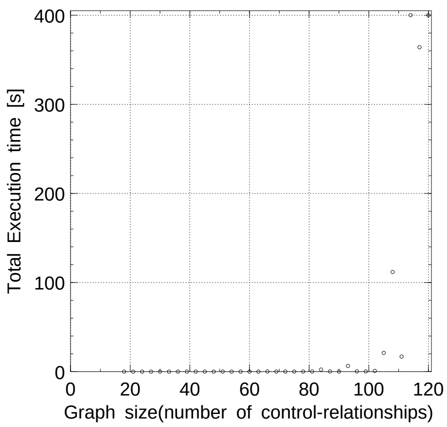
Results for **Reachability**



Results for **3-COL Ladder Graphs**



Results for **3-COL Simplex Graphs**



Results for **Strategic Companies**

Fig. 4. Experimental Results

of tuples – of the input relation). while the y-axis shows the total time taken by the system to compute the output relation. In the right diagram, the x-axis refers to the size (i.e., number of tuples) of the output. When the number of direct connections grows, the graph becomes very dense and we have a very high number of transitive connections in the result. A database instance consisting of nearly 600 direct connections, yielding some 10000 transitive connections, is processed in less than 2 minutes. When the graph becomes very dense, the performance gets worse, as for 1000 direct connections (15000 tuples in the output) `dlv` takes approximately 5 minutes. This encourages ongoing work on advanced data structures that are needed to efficiently support data-intensive applications. Nevertheless, it should be considered that **Reachability** is mainly a grounding and forward chaining benchmark which is not in the primary application domain of `dlv`.

For **Graph 3-Colorability**, we have considered two classes of graphs, namely *ladder graphs* and *simplex graphs* (see Figure 3). 3-Colorability on both classes has been used to evaluate the performance of other deductive systems, namely DeReS [5] and Smodels [17].

The results show that `dlv` is very efficient and scales well on these examples, as its advanced nonmonotonic inference mechanisms can be exploited there.

For **Strategic Companies**, we have generated instances for n companies and $3n$ products. Each company is controlled by one to five companies (Two consortia have to have at least one member in common), where the actual number of companies is uniform randomly chosen. On average there are 1.5 `controlled_by` relations per company.

The results for `dlv` are interesting if you consider that this problem is Σ_2^P-complete. On the x-axis we report the number of `controlled_by` relationships among couples of companies. Up to roughly 100, the evaluation is very fast. Then very hard instances can be met, but this is not surprising because of the high computational complexity of the problem.

We believe that further similar tests will prove useful to better understand the complex nature of Σ_2^P-hard problems.

References

1. C. Aravindan, J. Dix, and I. Niemelä. Dislop: A research project on disjunctive logic programming. *AI Communications*, 10(3/4):151–165, 1997.
2. F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm. In *Proceedings of 1990 ACM-SIGMOD International Conference*, pages 225–236, Atlantic City, NJ, May 1990.
3. 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.
4. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.
5. 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)*, Cambridge, Massachusetts, USA, 1996.

6. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The `dlv` System: Model Generator and Application Frontends. In *Proceedings of the 12th Workshop on Logic Programming (WLP '97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.
7. T. Eiter, G. Gottlob, and N. Leone. Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.
8. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.
9. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Non-monotonic Reasoning. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), Berlin, 1997. Springer.
10. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. Technical report, Institut für Informationssysteme, TU Wien, 1998.
11. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
12. A. Kakas, R. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 1993.
13. A. Kakas and P. Mancarella. Database Updates Through Abduction. In *Proceedings VLDB-90*, pages 650–661, 1990.
14. D. E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. ACM Press, New York, 1994.
15. N. Leone and P. Rullo. BQM: A System Integrating Logic, Objects, and Non-Monotonic Reasoning. In *Invited Paper on 7th IEEE International Conference on Tools with Artificial Intelligence*, Washington, November 1995.
16. 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.
17. I. Niemelä and P. Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, Sept. 1996.
18. C. S. Peirce. Abduction and induction. In J. Buchler, editor, *Philosophical Writings of Peirce*, chapter 11. Dover, New York, 1955.
19. Y. Peng and J. Reggia. *Abductive Inference Models for Diagnostic Problem Solving*. Symbolic Computation – Artificial Intelligence. Springer, 1990.
20. G. Phipps, M. A. Derr, and K. Ross. Glue-NAIL!: A Deductive Database System. In *Proceedings ACM-SIGMOD Conference on Management of Data*, pages 308–317, 1991.
21. D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5(1):97–110, 1989.
22. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL – Control, Relations and Logic. In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia, Canada, 1992.
23. R. Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32:57–95, 1987.
24. D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, 1994.
25. A. X3H2 and I. DBL. (ISO-ANSI Working Draft) Foundation (SQL/Foundation) [ISO DBL:LGW-008 / ANSI X3H2-97-030], Apr. 1997. Temporarily available at `ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sqlfound.txt`.