The dlv System: Model Generator and Application Frontends*

Simona Citrigno[d] **Thomas Eiter**[b] **Wolfgang Faber**[a]
Georg Gottlob[a] **Christoph Koch**[a] **Nicola Leone**[a]
Cristinel Mateis[a] **Gerald Pfeifer**[a†] **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 Università della Calabria
I-87030 Rende, Italy

[d]DEIS, Università della Calabria
I-87030 Rende, Italy

{*citrigno,eiter,faber,gottlob,koch,leone,mateis,pfeifer,scarcell*} *@dbai.tuwien.ac.at*

Abstract

During the last years, much research has been done concerning semantics and complexity of Disjunctive Deductive Databases (*DDDBs*). While DDDBs — function-free disjunctive logic programs with negation in rule bodies allowed — 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 ([ST94, ADN97]).
This paper presents a brief overview of the architecture of the `dlv` (datalog with disjunction) system system currently developed at TU Wien in the *FWF project P11580-MAT "A Query System for Disjunctive Deductive Databases"*, especially focusing on the Model Generator – the "heart" of the `dlv` system – and the integrated frontends for diagnostic reasoning and SQL3.

Keywords: Deductive Databases Systems, Disjunctive Logic Programming, Non-Monotonic Reasoning, Implementation.

1 System Overview

An outline of the general architecture of our system is depicted in Figure 1. The internal language of our system is an extension of disjunctive datalog in the direction of [BLR97], which also allows for integrity constraints. The kernel is an efficient engine for computing all or some stable models of a program. Various frontends, i.e. translators for specific applications into the internal language, are

*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.

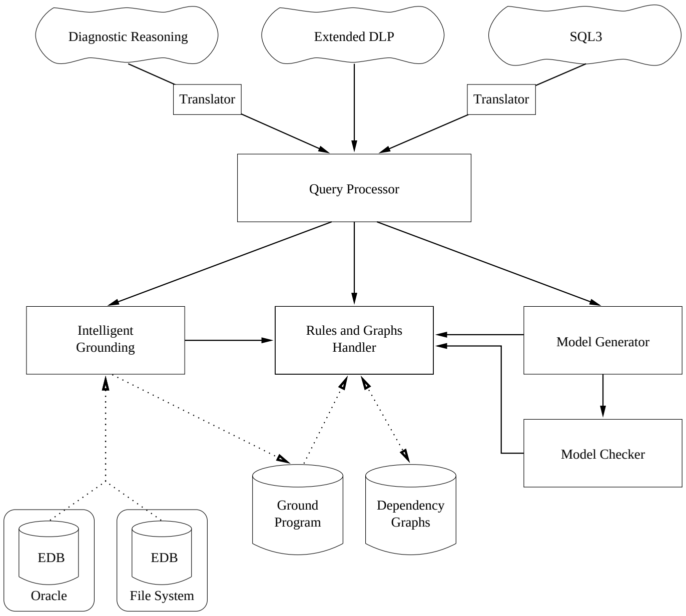


Figure 1: The System Architecture of `dlv`

currently developed on top of the kernel, some of which will be described in the second half of this paper.

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 [LRS97, ELM$^+$97].

2 Basic Definitions

For a background and unexplained concepts, see [Min94, LRS97]. A *datalog*$^{\vee,\neg}$ *rule* r is a clause of the form

$$h_1 \vee \cdots \vee h_n \leftarrow b_1, \cdots, b_k, \neg b_{k+1}, \cdots, \neg b_{k+m}, \qquad n \geq 1,\ k, m \geq 0$$

where $h_1, \cdots, h_n, b_1, \cdots, b_{k+m}$ are function-free atoms. We denote by $H(r)$ (resp. $B^+(r)$, $B^-(r)$) the set of head atoms (resp. positive body literals, negative body literals) of r; furthermore, $\neg.B^-(r) = \{b_i \mid \neg b_i \in B^-(r)\}$. If $n = 1$, then r is *normal* (i.e., $\vee$-free); if $m = 0$, then r is *positive* (or $\neg$-free). A *disjunctive datalog program* $\mathcal{P}$, denoted by *datalog*$^{\vee,\neg}$ and also called *disjunctive deductive database* (DDDB), is a finite set of rules; it is *normal* (resp. *positive*) if all its rules are normal (resp. positive). We assume that rules are *safe*, i.e., each variable occurring in a rule also has to occur in a positive body literal of that rule.

An integrity constraint, also called *(strong) constraint* is a clause of the form $\leftarrow L_1, \cdots, L_k$, where L_i, $1 \leq i \leq k$, is a literal (i.e., it is a rule with an empty head).

We define a *datalog*$^{\vee,\neg,c}$ *program* (simply "program") as a tuple $\mathcal{P} = \langle LP, S\rangle$, where LP is a datalog$^{\vee,\neg}$ program and S a (possibly empty) set of integrity constraints.

The extensional database (*EDB*) of a DDDB contains all rules of the form $a \leftarrow$ (i.e., non-disjunctive rules with an empty body), and the intensional database (*IDB*) contains all remaining rules. A predicate that appears in the head of an EDB (resp. IDB) rule is an EDB (resp. IDB) predicate; each predicate is assumed to be either an EDB predicate or an IDB predicate, but not both.

We denote by $U_{\mathcal{P}}$, B_{LP}, and *ground*$(\mathcal{P})$ the Herbrand universe, Herbrand base, and the ground instantiation of $\mathcal{P}$, respectively. Total (Herbrand) interpretations I and models of $\mathcal{P}$ are defined as usual. (During the computation we deal with three-valued interpretations, represented by sets of ground literals. The stable models returned at the end of the computation are total.)

3 Model Generator

Basically, the MG works as follows: Derive what is definitely derivable, then make an "educated" guess for one of those literals which have not been decided yet. This process is recursively applied until no further guess can be made. At that point, we have a stable model candidate and a call to the Model Checker takes place; if the candidate is not stable (or inconsistency arises at any time during the computation), backtracking is performed. To prune the search space – and to avoid the generation of duplicate models – we make heavy use of the knowledge obtained from previous computations, both failed and successful ones.

To formalize what we have called "educated guess" before, we introduce the concept of a possibly-true literal:

Definition 3.1 Let I be a (possibly partial) interpretation for $\mathcal{P}$.

A *positive possibly-true literal* of $\mathcal{P}$ w.r.t. I is a positive literal a, undefined w.r.t. I, such that there exists a rule $r \in$ *ground*$(\mathcal{P})$ for which all the following conditions hold:

1. $a \in H(r)$;
2. $H(r) \cap I = \emptyset$ (that is, the head is not true w.r.t. I);
3. $B(r) \subseteq I$ (that is, the body is true w.r.t. I).

A *negative possibly-true literal* of $\mathcal{P}$ w.r.t. I is a negative literal $\neg b \in \neg.B_{LP}$, undefined w.r.t. I, such that there exists a rule $r \in$ *ground*$(\mathcal{P})$ for which all the following conditions hold:

1. $\neg b \in B^-(r)$;
2. $H(r) \cap I = \emptyset$ (that is, the head is not true w.r.t. I);
3. $B^+(r) \subseteq I$ (that is, every positive literal of the body is true w.r.t. I);

4. $B^-(r) \cap \neg.I = \emptyset$ (that is, no negative literal of the body is false w.r.t. I).

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

Example 1 Consider the program $LP = \{a \vee b \leftarrow c, \neg d; \quad e \leftarrow c, \neg f\}$ and let $I = \{c, \neg d\}$ be an interpretation for LP. Then, we have three possibly-true literals of LP w.r.t. I: a, b and $\neg f$. $\Box$

The actual algorithm for computing stable models (sans the Model Checker) is shown in Figure 2. There we use the following additional notation:

- Given a set X of literals, X^+ denotes the set of positive literals occurring in X, and $\neg.X$ is the set containing the negation of the literals in X (the negation of a is $\neg a$ and vice versa).
- T_{LP} denotes the (skeptical version of the) immediate consequence operator:

$$T_{LP} : 2^{B_{LP} \cup \neg.B_{LP}} \rightarrow 2^{B_{LP}}$$

$$T_{LP}(I) = \{a \in B_{LP} | \quad \exists r \in ground(LP) \ s.t. \ a \in H(r), \ H(r) - \{a\} \subseteq \neg.I, and \ B(r) \subseteq I\}$$

- Φ_{LP} denotes an extension of Fitting's operator to the disjunctive case that computes the false atoms of a program with respect to an interpretation I.

$$\Phi_{LP} : 2^{B_{LP} \cup \neg.B_{LP}} \rightarrow 2^{B_{LP}}$$

$$\Phi_{LP}(I) = \{a \in B_{LP} | \quad \forall r \in ground(LP) \ with \ a \in H(r) : \ B(r) \cap \neg.I \neq \emptyset \ or \ H(r) - \{a\} \cap I \neq \emptyset\}$$

- $\mathcal{W}_{LP}$ [LRS95, LRS97] is the extension of the well-founded operator to the disjunctive case:

  $\mathcal{W}_{LP} : \mathbf{I_{LP}} \rightarrow 2^{B_{LP} \cup \neg.B_{LP}}; \qquad \mathcal{W}_{LP}(I) = T_{LP}(I) \cup \neg.GUS^{\vee}_{LP}(I)$

  where $GUS^{\vee}_{LP}(I)$ is the greatest unfounded set for LP w.r.t. I and $\mathbf{I}_{LP}$ is the set of interpretations having the greatest unfounded set [LRS95, LRS97].
- *unfounded-free* (see [LRS97]) is a function that, given a disjunctive program LP and a (total) model I of LP, returns true iff I contains no unfounded set; this condition is equivalent to checking that I is a stable model [LRS95, LRS97]

4 The native `dlv` interface

A datalog$^{\vee,\neg}$ rule is represented by

$$h_1 \texttt{ v } \ldots \texttt{ v } h_n \texttt{ :- } b_1 \texttt{,} \ldots \texttt{,} \ b_k \texttt{, not } b_{k+1} \texttt{,} \ldots \texttt{, not } b_{k+m} \texttt{.}$$

where an atom is represented by

$$a\texttt{(}t_1\texttt{,} \ldots \texttt{,}t_o\texttt{)}$$

a being the name of that particular atom. Recall that $h_1, \ldots, h_n, b_1, \ldots, b_{k+m}$ are atoms, while each t_i $(1 \leq i \leq o)$ is either a constant or a variable.

The name of an atom is a finite string starting with a letter (`A`–`Z`, `a`–`z`) followed by a possibly empty string of alphanumeric characters (`A`–`Z`, `a`–`z` and `0`–`9`) and underscores (`_`). Variables (resp. constants) are represented by a finite string starting with an uppercase letter (`A`–`Z`) (resp. lowercase letter (`a`–`z`)) followed by a possibly empty string of alphanumeric characters and underscores.

Each rule can easily span multiple input lines. The operator `not` is case insensitive and may be also written as `non`, which is case insensitive as well.

Algorithm ComputeStableModels
Input: A datalog program LP and a set S of strong constraints.
Output: The stable models of LP that satisfy all strong constraints in S (if any).

Procedure $Compute\_Stable(I' : SetOfLiterals; S' : SetofConstraints)$;
(* The procedure outputs all stable models of LP that satisfy S' and contain I' *)
var J, J', Q: SetOfLiterals; L: Literal;
    if $PT_{LP}(I') = \emptyset$ **then** (* I'^{+} is a model of LP *)
        if *unfounded-free*$(LP,\ I'^{+}) \wedge satisfied(S', I'^{+})$ **then**
            output I'^{+}; (* I'^{+} is a stable model *)
        end_if
    else
        $Q := PT_{LP}(I')$
        while $Q \neq \emptyset$ **do**
            Take a literal L from Q;
            $J := I' \cup \{L\}$; (* Assume the truth of a possibly-true literal *)
            repeat
                $J' := J$;
                $J := J' \cup T_{LP}(J') \cup \neg.\Phi_{LP}(J')$;
            until $J = J'$ or $J \cap \neg.J \neq \emptyset$;
            if $J \cap \neg.J = \emptyset$ **then** (* J is consistent *)
                $Compute\_Stable(J, S')$
            end_if
            (* At this point all stable models containing $I' \cup \{L\}$ have been generated *)
            $Q := Q - \{L\}$
            if L is a positive literal **then**
                $I' := I' \cup \{\neg L\}$ (* Assume that L is false in following computations *)
            else
                $S' := S' \cup \{\leftarrow L\}$ (* Forbid the generation of L in following computations *)
            end_if
        end_while
    end_if
end_procedure

var I, J: SetOfLiterals;
begin (* Main *)
    $I := \emptyset$;
    repeat (* Computation of $\mathcal{W}^{\omega}_{LP}(\emptyset)$ *)
        $J := I$;
        $I := \mathcal{W}_{LP}(I)$;
    until $I = J$;
    if $PT_{LP}(I) = \emptyset$ **then** (* I^{+} is the unique stable model of LP *)
        if $satisfied(S, I^{+})$ **then**
            output I^{+};
        end_if
    else
        $Compute\_Stable(I, S)$;
    end_if
end

Figure 2: Algorithm for the Computation of Stable Models of datalog$^{\vee,\neg,c}$ programs

5 Brave and Cautious Reasoning

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

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

where each b_i $(1 \leq i \leq n)$ is an atom.

The brave reasoning frontend is invoked by the `-FB` command line option. If the query evaluates to true in at least one stable model, it is considered true, else it is false.

Similarly, the cautious frontend is invoked by `-FC` and the query is considered true if and only if it evaluates to true in all stable models.

6 The SQL Frontend

This frontend translates a subset of SQL3 query expressions to datalog queries. Since the SQL3 standard has not been completed yet, we resorted to the current working draft [XD97]. 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 translated by the 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 ... ] ;
```

where { *item* } [*connective* ...] represents a list of *item*s, separated by *connective*s. Otherwise, [*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 in general have two parts: A definition part (the WITH clause), and a query part.

6.1 Example – 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.)

7 Diagnostic Reasoning

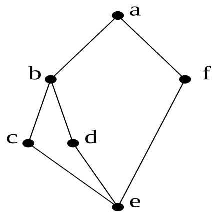
Our frontend for abductive diagnosis ([Poo89, EGL97]) currently supports three different modes: General diagnosis, where all diagnoses are computed, subset minimal diagnosis and single failure diagnosis. These modes are invoked by the command line options `-FD`, `-FDmin` and `-FDsingle`, respectively.

The diagnostic theory obeys the syntax described in 4. 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 all input has been read, the hypotheses are used to generate disjunctive rules that guess all possible diagnosis candidates, while the observations become contraints that forbid the generation of diagnoses that are not consistent with the observations. In the case of subset minimal diagnosis some rules guaranteeing minimality are added. Finally the grounding (and subsequently the MG etc.) are invoked and for each stable model found, the corresponding set of hypotheses is output.

7.1 Example – Diagnosis of a Network

On the computer network depicted below, we make the observation that, sitting at machine a which is online, we cannot reach machine e. Which machines are offline?



This can be easily modelled as the following diagnostic problem, where the theory is

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

and the hypotheses and observations are

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

respectively

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

8 Outlook and Further Work

We plan to develop further frontends for experimenting with semantics and applications of nonmonotonic reasoning. Moreover we will also continue to improve the overall efficiency of the system.

For the SQL frontend, we are currently implementing more of the language features of query expressions as defined in [XD97], in particular set operators EXCEPT and INTERSECT and explicit joins as well as additional comparison predicates like IN, $<$ and $>$. (To support the latter, we will also add arithmetic built-in predicates to our language.)

References

[ADN97] C. Aravindan, J. Dix, and I. Niemelä. Dislop: A research project on disjunctive logic programming. *AI Communications*, 10(2), 1997.

[BLR97] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR '97)*, Dagstuhl, Germany, July 1997.

[EGL97] T. Eiter, G. Gottlob, and N. Leone. Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 1997. To appear.

[ELM+97] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), Berlin, 1997. Springer.

[LRS95] N. Leone, P. Rullo, and F. Scarcello. Declarative and Fixpoint Characterizations of Disjunctive Stable Models. In *Proceedings of the International Logic Programming Symposium – ILPS'95*, pages 399–413, Portland, Oregon, December 1995. MIT Press.

[LRS97] 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.

[Min94] J. Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[Poo89] D. Poole. Explanation and Prediction: An Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5(1):97–110, 1989.

[ST94] 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.

[XD97] ANSI X3H2 and ISO DBL. (ISO-ANSI Working Draft) Foundation (SQL/Foundation) [ISO DBL:LGW-008 / ANSI X3H2-97-030], April 1997. Temporarily available at `ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sqlfound.txt`.