The Architecture of a Disjunctive Deductive Database System*

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

[a]Information Systems Department, TU Vienna
A-1040 Vienna, Austria

[b]AG Informatik, University of Giessen
Arndtstrasse 2, D-35392 Giessen

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

email addresses: {eiter,leone,mateis,pfeifer,scarcell}@dbai.tuwien.ac.at

Abstract

Disjunctive Deductive Databases (*DDDBs*) — function-free disjunctive logic programs with negation in rule bodies allowed — have been recently recognized as a powerful tool for knowledge representation and commonsense reasoning. Much research has been spent on issues like semantics and complexity of DDDBs, but the important area of implementing DDDBs has been less addressed so far. However, a thorough investigation thereof is a basic requirement for building systems which render previous foundational work on DDDBs useful for practice.

This paper presents the architecture of a DDDB system currently developed at TU Vienna in the *FWF project P11580-MAT "A Query System for Disjunctive Deductive Databases"*.

Keywords. Deductive Databases Systems, Disjunctive Logic Programming, Knowledge Representation, Non-Monotonic Reasoning.

1 Introduction

The study of integrating databases with logic programming opened in the past the research field of *deductive databases*. Basically, a deductive database is a logic program without function symbols, i.e., a datalog program (possibly extended with negation). A number of advanced deductive database systems have been developed that utilize logic programming and extensions thereof for querying relational databases, e.g., [10, 16, 19].

*Work supported in part by FWF (Austrian Science Funds) under *project P11580-MAT*, and by the *Istituto per la Sistemistica e l'Informatica, ISI-CNR.*

†Please address correspondence to this author; Phone: +43 1 588016126; Fax: +43 1 5055304

‡Work done while visiting TU Vienna.

The need for representing disjunctive (or incomplete) information led to *Disjunctive Deductive Databases* (*DDDBs*) [14]. Disjunctive deductive databases can be basically seen as disjunctive logic programs without function symbols, i.e., *disjunctive datalog programs* [8].

DDDBs are nowadays widely recognized as a valuable tool for knowledge representation and reasoning [1, 13, 24, 9]. The strong interest in enhancing deductive databases by disjunction is documented by a number of publications (cf. [13]) and even workshops dedicated to this subject (cf. [24]). An important merit of DDDBs over normal (i.e., disjunction-free) logic programming is its capability to model incomplete knowledge [1, 13].

The presence of disjunction in the rule heads makes DDDBs inherently non-monotonic, i.e., new information can invalidate previous conclusions, and renders the task of defining the semantics of a DDDB more difficult. A lot of research has been spent on the semantics of DDDBs, and several alternative approaches have been proposed, e.g. [4, 9, 14, 18, 17, 20] (see [6, 13] for comprehensive surveys). The most widely accepted semantics is the extension of the stable model semantics to DDDBs [9, 18]. It comes with high expressive power: DDDBs under stable model semantics capture the complexity class Σ_2^P over relational databases (i.e., they allow to express precisely the properties which are decidable in non-deterministic polynomial time with an oracle in NP) [1]. As shown in [8], the high expressive power has also a practical relevance, as concrete real world situations can be represented by DDDBs, while they cannot be expressed by disjunction-free programs. Furthermore, DDDBs have strong connections to Artificial Intelligence, since several non-monotonic logic languages can be equivalently translated into DDDBs (under stable model semantics), and the computation of the stable model semantics of DDDBs is thus "at the heart of the computation" of several important problems in AI [9, 21].

Despite the importance of computing the semantics of a DDDB (i.e., the set of its stable models), so far most works concentrated on the semantical side, while algorithms for computing DDDBs have been left nearly untouched.

This paper presents the architecture of a disjunctive deductive database system which is under development at TU Vienna in the *FWF project P11580-MAT "A Query System for Disjunctive Deductive Databases"*.

2 Preliminaries

A *term* is either a constant or a variable[2]. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* $\neg p$, where p is an atom.

A *(disjunctive) rule* r is a clause of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \neg b_{k+1}, \cdots, \neg b_{k+m}, \qquad n \geq 1, \quad k, m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_{k+m}$ are atoms. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \neg b_{k+1}, ..., \neg b_{k+m}$ is the *body* of r. 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 $n = 1$ (i.e., the head is $\vee$-free), then r is *normal*; if $m = 0$ (the body is $\neg$-free), then r is *positive*. A Datalog$^{\vee,\neg}$ program $\mathcal{P}$

[1] Note that (non-disjunctive) deductive databases are strictly less expressive.

[2] Note that function symbols are not considered in this paper.

is a finite set of rules (also called *disjunctive datalog* program, or *disjunctive deductive database*); $\mathcal{P}$ is *normal* (resp., *positive*) if all rules in $\mathcal{P}$ are normal (resp. positive).

A program $\mathcal{P}$ is usually divided into two parts: The extensional database (*EDB*) contains all rules of the form $a \leftarrow$, i.e. non-disjunctive rules with an empty body, whereas the intensional database (*IDB*) contains all remaining rules. We call a predicate that appears in the head of an EDB resp. IDB rule an EDB resp. IDB predicate and assume that each predicate occurring in $\mathcal{P}$ is either an EDB predicate or an IDB predicate, but not both.

As usual in the context of deductive databases, 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.

The *Herbrand universe* $U_\mathcal{P}$ of $\mathcal{P}$ is the set of all constants appearing in $\mathcal{P}$. The *Herbrand base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of all possible ground atoms constructible from the predicates appearing in $\mathcal{P}$ and the constants occurring in $U_\mathcal{P}$ (clearly, both $U_\mathcal{P}$ and $B_\mathcal{P}$ are finite). The instantiation of rules is defined in the obvious way over the constants in $U_\mathcal{P}$ and is denoted by $ground(\mathcal{P})$.

A *(total) interpretation* for $\mathcal{P}$ is a subset I of $B_\mathcal{P}$. A ground atom a is *true* (resp. *false*) w.r.t. I if $a \in I$ (resp. $a \notin I$); the literal $\neg a$ is *true* (resp., *false*) w.r.t. I if $a \notin I$ (resp., $a \in I$). A rule r in $ground(\mathcal{P})$ is *satisfied* (or *true*) w.r.t. I, if some atom in $H(r)$ is true w.r.t. I or some body literal in $B^+(r) \cup B^-(r)$ is false w.r.t. I.

Minker proposed in [14] a model-theoretic semantics for positive disjunctive programs $\mathcal{P}$, in which $\mathcal{P}$ is assigned the set $\mathrm{MM}(\mathcal{P})$ of its *minimal models*, each of which representing a possible meaning of $\mathcal{P}$; a model M for $\mathcal{P}$ is minimal, if no proper subset of M is a model for $\mathcal{P}$.

Example 2.1 For the positive program $\mathcal{P} = \{a \vee b \leftarrow\}$ the (total) interpretations $\{a\}$ and $\{b\}$ are its minimal models (i.e., $\mathrm{MM}(\mathcal{P}) = \{\ \{a\},\ \{b\}\ \}$).

The stable model semantics generalises the above approach to programs with negation. Given a program $\mathcal{P}$ and a total interpretation I, the *Gelfond-Lifschitz transform* of $\mathcal{P}$ with respect to I, denoted by $\mathcal{P}^I$, is the positive program defined as follows:

$$\mathcal{P}^I = \{H(r) \leftarrow B^+(r) \mid r \in ground(\mathcal{P}), \neg.B^-(r) \cap I = \emptyset\}$$

Then, an interpretation I is a *stable model* for $\mathcal{P}$, if $I \in \mathrm{MM}(\mathcal{P}^I)$ [18, 9]; the set of all stable models for $\mathcal{P}$ is denoted by $STM(\mathcal{P})$.

Example 2.2 Let $\mathcal{P} = \{a \vee b \leftarrow c,\quad b \leftarrow \neg a, \neg c,\quad a \vee c \leftarrow \neg b\}$. Consider $I = \{b\}$. Then, $\mathcal{P}^I = \{a \vee b \leftarrow c,\quad b \leftarrow\}$. Clearly, I is a minimal model for $\mathcal{P}^I$; thus, I is a stable model for $\mathcal{P}$. □

It is clear that $\mathcal{P}^I$ coincides with $ground(\mathcal{P})$ if $\mathcal{P}$ is positive. Hence, the minimal and stable models of a positive program coincide. In particular, every normal positive program $\mathcal{P}$ has exactly one stable model which coincides with the single minimal model of $\mathcal{P}$.

3 System Architecture

In this section we provide an outline of the general architecture of our system, which is depicted in Figure 1. It will be discussed in more detail in Section 4.

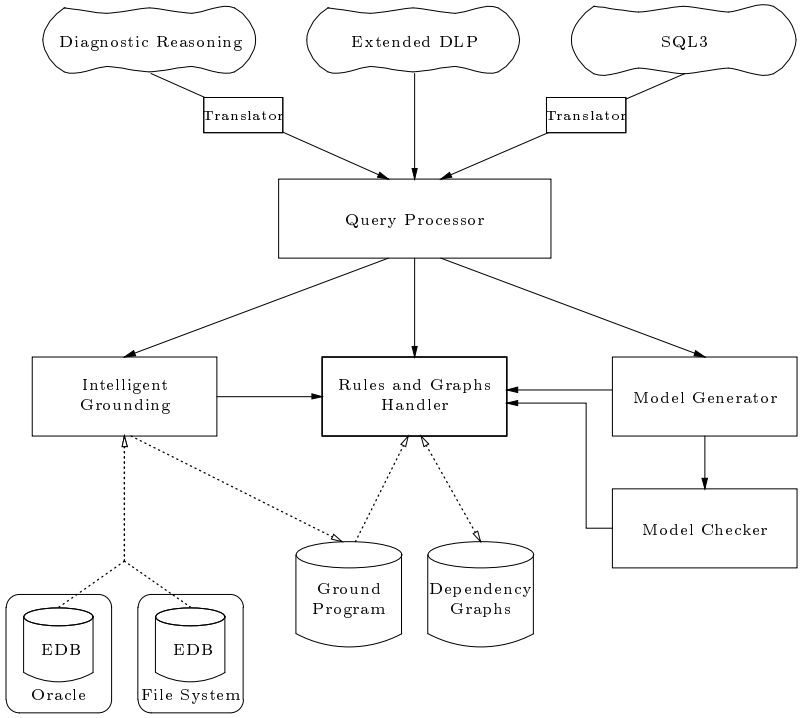

Figure 1: The General Architecture of the System

The user language of our system is an extension of disjunctive datalog in the direction of [5], which also allows for aggregates. Translators for specific applications into this language are currently developed.

At the heart of the system lies the Query Processor, which serves a double purpose: On the one hand it controls the execution of the entire system. On the other hand it performs some post-processing on the generated models which allows us to perform standard cautious reasoning and brave reasoning, as well as simply generating all or a given number of stable models of a program.

Upon startup, the Query Processor reads the – possibly non-ground – input program and hands it over to the *Rules and Graphs Handler* (RGH), 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* (IGM). The IGM efficiently generates a subset of the grounded input program which has exactly the same stable models. In general, this subset is much smaller.

The Query Processor then again invokes the RGH module, which generates partitionings of two copies of the program output by the IGM. They are used by the Model Generator (MG) and the Model Checker (MC), respectively, and enable a modular evaluation of the program, which often yields a tremendous speedup.

Finally, the Model Generator is started, which generates one candidate for a stable model at a time and calls the Model Checker whether it is indeed a stable model. Upon success, control is returned to the Query Processor, which performs post-processing and possibly invokes the Model Generator to look for further models. More details on MG and MC can be found in [11].

4 Main Modules of the System

We omit the description of the Query Processor, as it is mainly controlling the flow of execution, which has been outlined above.

4.1 Rules and Graphs Handler

The RGH builds and handles some graphs representing different relations between the atoms resp. predicates of the given program $\mathcal{P}$, as well as to single out and analyze syntactical modules of $\mathcal{P}$.

We define relations $\prec^+$, $\prec^-$, and $\prec^h$ on the Herbrand base $B_\mathcal{P}$ as follows: For any $p, q \in B_\mathcal{P}$, $p \prec^+ q$ (resp. $p \prec^- q$, $p \prec^h q$) holds if there exists a rule $r \in \mathit{ground}(\mathcal{P})$ such that $p \in H(r)$ and $q \in B^+(r)$ (resp. $q \in \neg.B^-(r)$, $q \in H(r)$).

Ground Dependency Graph (GG_Π). Given the ground program Π computed by the IGM, one important task performed by the RGH is detecting its syntactical modules, which can be evaluated separately [12, 8]. This can lead to an exponential gain in efficiency. Indeed, even if Π is not in an efficiently evaluable class of programs, many "components" of Π may be. They often fit into known tractable syntactical classes like, e.g., stratified normal logic programs.

We associate to a ground program Π a digraph GG_Π whose set of nodes is B_Π and whose set of arcs E is the union of three differently labeled sets of arcs E^+, E^-, and $E^\vee$. For each pair of nodes q, p, the arc $q \to p$ is in E^+ (resp., E^-, $E^\vee$) iff $p \prec^+ q$ (resp., $p \prec^- q$, $p \prec^h q$) holds. The arc labels are useful to single out syntactical features of subprograms, e.g. limited use of negation or disjunction.

The graph GG_Π naturally induces a partitioning of Π into subprograms called modules, which can be evaluated in a modular fashion. More formally, we say a rule $r \in \Pi$ *defines* an atom p if $p \in H(r)$. A *module* of Π is the set of rules defining all the atoms contained in a particular maximal strongly connected component (SCC) of GG_Π. Intuitively, a module includes (among others) all rules defining mutually dependent atoms.

Ground Collapsed Positive Dependency Graph ($\overline{GG^+_\Pi}$). Given a graph G, we denote by $\overline{G}$ the *collapsed* graph G, which results from G by collapsing each SCC into a single node. Thus, every node of $\overline{G}$ is a set of nodes of G.

The RGH computes also the collapsed graph $\overline{GG^+_\Pi}$, where GG^+_Π is the subgraph of GG_Π whose set of arcs is E^+. This graph is exploited by the Model Checker.

Example 4.1 Consider the following ground program Π:

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

The graph GG_Π is depicted in Figure 2(a). (We use "$\neg$" and "$\vee$" for labeling arcs in E^- and $E^\vee$, respectively, while arcs in E^+ remain unlabeled).

The SCCs of GG_Π are: $\{p(2,3)\}$, $\{t(2)\}$, $\{q(1), q(3)\}$ and $\{t(3)\}$. They correspond to the modules $\{p(2,3) \leftarrow\}$, $\{t(2) \leftarrow\}$, $\{q(1) \vee q(3) \leftarrow p(2,3), \neg t(2)\}$, and $\{t(3) \leftarrow q(3), p(2,3)\}$, respectively.

The collapsed graph $\overline{GG^+_\Pi}$ for GG^+_Π is depicted in Figure 2(b). $\square$

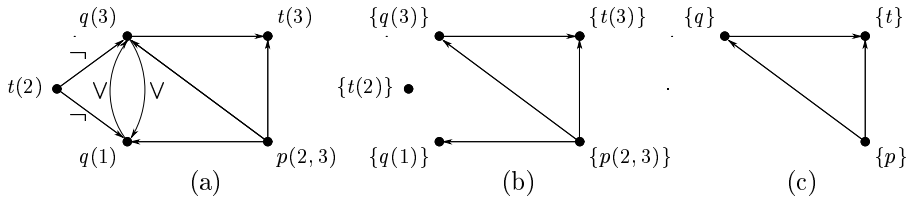

Figure 2: Dependency graphs (a) GG_Π, (b) $\overline{GG^+_\Pi}$, and (c) $\overline{G^+_\mathcal{P}}$

Non-ground Collapsed Positive Dependency Graph ($\overline{G^+_\mathcal{P}}$). The IGM needs information about (positive) dependencies among the IDB predicates of the input program $\mathcal{P}$. By detecting mutually recursive predicates and by imposing a suitable order on the rules defining them, a ground program Π "equivalent" to $\mathcal{P}$ can be generated which avoids a lot of useless work.

Based on the natural adaptation of $\prec^+$ to the set of predicates instead of atoms, we associate to $\mathcal{P}$ a digraph $G^+_\mathcal{P}$ as follows. The nodes of $G^+_\mathcal{P}$ are the IDB predicates of $\mathcal{P}$, and $G^+_\mathcal{P}$ has an arc from a node q to a node p iff $p \prec^+ q$.

The RGH computes the respective collapsed graph $\overline{G^+_\mathcal{P}}$.

Example 4.2 Consider the following program $\mathcal{P}$, where a is an EDB predicate:

$$\begin{array}{rcl} p(1,2) \vee p(2,3) & \leftarrow & \\ t(X) & \leftarrow & a(X) \end{array} \qquad \begin{array}{rcl} q(X) \vee q(Z) & \leftarrow & p(X,Y), p(Y,Z), \neg t(Y) \\ t(X) & \leftarrow & q(X), p(Y,X) \end{array}$$

The graph $\overline{G^+_\mathcal{P}}$ is depicted in Figure 2(c); here, it is isomorphic $G^+_\mathcal{P}$. □

Summarizing, the operations performed by the RGH are:

- given the "input" program $\mathcal{P}$, it computes the graph $\overline{G^+_\mathcal{P}}$ for the IGM;
- given the ground program Π from the IGM, it computes GG_Π.
- from Π and GG_Π, it sets apart the modules of Π for evaluation by the Model Generator;
- it analyzes the syntactical properties of each module of Π (e.g. stratification) for use by the Model Generator;
- it computes the graph $\overline{GG^+_\Pi}$, which is exploited by the Model Checker in order to reduce the search space;
- for each component of $\overline{GG^+_\Pi}$ it analyzes the syntactical properties (e.g. headcycle-freeness [2]) of the corresponding subprogram of Π, again for use by the Model Checker.

4.2 Intelligent Grounding

Since the kernel modules of our system work on the ground instantiation of the input program, an efficient instantiation procedure is important.

The main reason of large groundings even for small input programs is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in $B_\mathcal{P}$, which leads to combinatorial

explosion. However, in a reasonable 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 $\mathcal{P}$.

Let $\mathcal{P}$ be a non-ground program and C a node of the graph $\overline{G^+_\mathcal{P}}$. We denote by $recursive\_rules_\mathcal{P}(C)$ the set of the rules r from $\mathcal{P}$ 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 in $\mathcal{P}$ with a predicate from C in $H(r)$.

Recall 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 extends to r. We define the grounding of r w.r.t. a set of ground atoms $NF \subseteq B_\mathcal{P}$, denoted by $ground(r, NF)$, as the set of ground instances r' of r s.t. $B^+(r') \subseteq NF$.

In order to compute such an r', one can proceed by matching the atoms in $B^+(r)$ one by one with atoms in NF and binding the free variables accordingly in each step. An efficient heuristics is to start with positive literals whose predicate occurs infrequent in NF.

A simplified version of our algorithm *Instantiate*, which computes a ground program Π that has the same stable models as $\mathcal{P}$, is outlined in Figure 3. There, $EDB_\mathcal{P}$ and $IDB_\mathcal{P}$ denote the database and intensional part of $\mathcal{P}$, respectively.

Initially, it sets $NF = EDB_\mathcal{P}$ and $\Pi = \emptyset$. Then, it removes a node C from $\overline{G^+_\mathcal{P}}$ which has no incoming arc (i.e., a source), and generates all instances r' of rules r defining predicates in C which can possibly derive new atoms, given that the atoms in NF are possibly derivable. This is done by calls to *Instantiate_rule*.

These r' are the rules in $ground(r, NF)$ such that every negative EDB literal in $B^-(r')$ is true w.r.t. $EDB_\mathcal{P}$. The set $H(r')$ is included in NF and r' is added to Π after simplifications (here, removal of EDB literals).

For efficiency reasons, first non-recursive rules are instantiated once and for all. Then, the recursive rules are repeatedly instantiated until NF is unchanged.

After that, the next source is processed until $\overline{G^+_\mathcal{P}}$ becomes empty.

Example 4.3 Reconsider $\mathcal{P}$ in Example 4.2, and suppose $EDB_\mathcal{P} = \{a(2)\}$. Then, *Instantiate* computes for $\mathcal{P}$ the following ground program Π:

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

Evaluation of node {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 4.4 *Let $\mathcal{P}$ be a safe disj. datalog program, and Π be the ground program generated by* Instantiate*($\mathcal{P}$). Then, $\mathcal{P}$ and Π have the same stable models.*

4.3 Model Generator

The *Model Generator* produces a set of interpretations that are "candidates" for stable models, which are submitted to the Model Checker for verification. Since the number

```
Function Instantiate(P: Safe_Program) : GroundProgram
var Π: GroundProgram; C: SetOfPredicates; NF: SetOfAtoms;
begin
    NF := EDB_P; Π := ∅;
    G+_P := non-ground collapsed positive dependency graph of P;
    while G+_P ≠ ∅ do
        Remove a node C of G+_P without incoming edges;
        for each r ∈ exit_rules_P(C) do
            Instantiate_rule(P, r, NF, Π);
        repeat
            NF1 := NF;
            for each r ∈ recursive_rules_P(C) do
                Instantiate_rule(P, r, NF, Π);
        until NF1 = NF
    end_while
    return Π
end_function;


Procedure Instantiate_rule(P: Safe_Program; r: Rule;
                           var NF: SetOfAtoms;
                           var Π: GroundProgram)
var H : SetOfAtoms; B+, B−: SetOfLiterals;
begin
    for each instance H ← B+, B− of r in ground(r, NF) do
        if ¬.B− ∩ EDB_P = ∅ then
            NF := NF ∪ H;
            remove EDB literals of P from B+, B−;
            Π := Π ∪ {H ← B+, B−}
        end_if
end_procedure;
```

Figure 3: Basic algorithm for computation of the (simplified) instantiated program

of candidates can be exponential, this module is very important for our system, as it is crucial for performance.

The Model Generator employes a number of techniques for pruning the search space. It exploits recent results on *modularity properties* of the stable model semantics [8, 12], to compute the stable models of one module (subprograms as described in Section 4.1) at a time, following the order suggested by the ground dependency graph GG_Π. Before evaluating a module Λ, the Model Generator asks the RGH to know the syntactic features of Λ. Accordingly, it selects the appropriate computational strategy (e.g., a straightforward polynomial-time algorithm for normal stratified programs, a special algorithm for headcycle-free disjunctive programs [3], etc.). In this way, the algorithm selects the best strategy for each module and limits the inefficient (combinatorial) part of the computation to the modules that are intrinsically hard.

The evaluation of the "hard" modules (i.e., non-headcycle-free, unstratified modules) follows the efficient algorithm for computing disjunctive stable models proposed in [11]. The algorithm is based on a backtracking technique which spans the search space for computing all stable models of the program.

The computation of $STM(\mathcal{P})$ relies on a monotonic operator $\mathcal{W}_\mathcal{P}$ [11] that extends the well-founded operator in [23]. It is defined in terms of a proper 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 cannot be derived from $\mathcal{P}$ assuming the facts in I [11].

Briefly, the algorithm works as follows. $\mathcal{W}^\omega_\mathcal{P}(\emptyset)$ is first computed, which is contained in every stable model. If $\mathcal{W}^\omega_\mathcal{P}(\emptyset)$ is a stable model, it is returned as the unique stable model. Otherwise, moving from $\mathcal{W}^\omega_\mathcal{P}(\emptyset)$ towards the stable models of $\mathcal{P}$, a conjunction of literals (called *possibly-true conjunction* in [11]), whose truth allows to infer new atoms, is assumed true. Then, the computation proceeds by iteratively applying the inflationary version of the immediate consequence operator, until either a stable model of $\mathcal{P}$ is reached or a contradiction arises (i.e., two contradictory literals are generated); in the latter case, backtracking is performed. See [11] for the details.

It is worth noting that a number of further techniques are employed for pruning the search space and improving the efficiency, which cannot be described here for space reasons.

4.4 Model Checker

The *Model Checker* (MC) is invoked by the Model Generator to verify whether a given interpretation is a stable model. Thus, Model Checker has to perform a very hard task in general, because checking the stability of a model is well known to be a co-NP-complete problem [8]. However, recent studies [2, 3, 11] unveiled that minimal model checking — the hardest part of stable model checking — can be efficiently performed for the relevant class of *head-cycle-free* (*HCF*) programs [2]. The MC satisfies the above complexity bounds. Indeed: (a) it terminates in polynomial time on every HCF program; (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. Following the algorithm presented in [11], the MC exploits the information provided by the Rules and Graphs Handler and stored in the graph $\overline{GG^+_\Pi}$ (see Section 4.1). In particular, it proceeds by processing one component of $\overline{GG^+_\Pi}$ at a time, as a model M is stable exactly if, for each component C of $\overline{GG^+_\Pi}$, the set $C \cap M$ contains no

non-empty unfounded set [11]. First of all, the MC computes the set $\mathcal{R}^{\infty}_{\Pi,M}(C \cap M)$ using the monotonic operator $\mathcal{R}_{\Pi,M}(X)$. Intuitively, $\mathcal{R}_{\Pi,M}(X)$ discards from X only elements which are "supported" and cannot belong to any unfounded set. Then, three cases may arise:

(1) $\mathcal{R}^{\infty}_{\Pi,M}(C \cap M) = \emptyset$,

(2) $\mathcal{R}^{\infty}_{\Pi,M}(C \cap M) \neq \emptyset$, and C is HCF,

(3) $\mathcal{R}^{\infty}_{\Pi,M}(C \cap M) \neq \emptyset$, and C is not HCF.

In case 1, the $C \cap M$ contains no non-empty unfounded set, and the Model Generator can proceed to the next component of $\overline{GG^{+}_{\Pi}}$. Case 2 implies that M is not a stable model, and the MC returns this information. In case 3, the subsets of $\mathcal{R}^{\infty}_{\Pi,M}(C \cap M)$ must be considered. If some of them is unfounded, the MC returns that M is not a stable model; otherwise, it proceeds with the analysis of the next component of $\overline{GG^{+}_{\Pi}}$. Note that case 3 is the most expensive case in general.

If all the components are succesfully analyzed, the MC terminates succesfully and returns that M is a stable model.

5 Conclusion

We presented the basic architecture of an advanced disjunctive deductive database system based on some of our previous foundational studies.

Based on this general architecture, we are currently building a system written in C++, utilizing compiler construction tools (flex and bison) and the Standard Template Libraries (STL), which will be part of the new ANSI C++ standard.

So far, we have implemented a fully operational prototype system. Current work includes optimizations on internal data structures and algorithms, as well as research on a more effective pruning of the search space, e.g. by exploiting various properties of (sub)programs. In the near future, we plan to add frontends and translators for SQL3 and diagnosis, as well as a better user interface.

For up-to-date information and a download of our system, please visit the URL `http://www.dbai.tuwien.ac.at/proj/dlv/`.

References

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

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

[3] R. Ben-Eliyahu and L. Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proc. KR-94*, pp. 39–50, 1994.

[4] S. Brass and J. Dix. Disjunctive Semantics based upon Partial and Bottom-Up Evaluation. In *Proc. ICLP '95*, pp. 199–213, Tokyo, June 1995.

[5] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In *Proc. LPNMR '97*, Dagstuhl, Germany, July 1997.

[6] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information*, pp. 241–329. DeGruyter, 1995.

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

[8] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM TODS*, September 1997. To appear.

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

[10] S. Greco, N. Leone, and P. Rullo. COMPLEX: An Object-Oriented Logic Programming System. *IEEE TKDE*, 4(4), August 1992.

[11] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 1997. (Forthcoming).

[12] V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proc. ICLP '94*, pp. 23–38, 1994.

[13] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, 1992.

[14] J. Minker. On Indefinite Data Bases and the Closed World Assumption. In *Proc. CADE '82*, LNCS 138, pp. 292–308, 1982.

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

[16] G. Phipps, M. A. Derr, and K. Ross. Glue-NAIL!: A Deductive Database System. In *Proc. ACM-SIGMOD*, pp. 308–317, 1991.

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

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

[19] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL – Control, Relations and Logic. In *Proc. VLDB '92*, Vancouver (BC), Canada, 1992.

[20] C. Sakama. Possible Model Semantics for Disjunctive Databases. In *Proc. DOOD-89*, pp. 337–351, 1989.

[21] C. Sakama and K. Inoue. Embedding Circumscriptive Theories in General Disjunctive Programs. In *Proc. LPNMR '95*, pp. 344–357, 1995.

[22] D. Seipel. Non-Monotonic Reasoning Based on Minimal Models and its Efficient Implementation. In [24], pp. 53–60.

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

[24] B. Wolfinger, editor. GI Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Springer, 1994.