Project Number:	IST-2001-33570
Project Acronym:	INFOMIX
Project Full Name:	Infomix: Boosting the Information Integration
Deliverable Number:	D4.2
Title:	**Algorithms and implementation techniques for the internal integration formalism**
Workpackage:	WP4
Document Type:	Deliverable
Distribution:	Infomix Consortium
Status:	Draft
Document file:	WP4_T1_D4.2_0102.ps
Version:	1.2
Number of pages:	28
Due Date:	September 30, 2003
Delivery Date:	November 11, 2003
Partners contributed:	UNICAL
Partners owning:	UNICAL
Man Months:	10

Short Description:

The report provides algorithms and implementation techniques exploited in the internal integration layer of the INFOMIX system. It mainly presents extensions and optimizations of the DLV system which aim at making it more suitable for data integration contexts. In particular, the report presents: *(i)* implementation techniques for handling aggregate operators, *(ii)* a structure-based backjumping algorithm allowing to improve efficiency of program instantiation, *(iii)* an extension of DLV allowing to improve database interaction and efficient data manipulation.

Document Change Record		
Version	**Date**	**Reason for Change**
v.1.0	15-9-2003	First draft
v.1.1	21-9-2003	Revised Version
v.1.2	2-10-2003	Final Version

Contents

1 Introduction

In this report we describe algorithms and implementation techniques to be exploited in the internal integration layer of the INFOMIX system and allowing to implement and optimize the approaches presented in the previous reports. In particular, since we have chosen DLV as the reference DLP engine, the techniques we present here are basically extensions to the DLV system making it more suitable for data integration application contexts.

As we have already pointed out, even if DLP seems to be the best suited Internal Integration Formalism in the context of the INFOMIX project, classical DLP presents some limitations for its applicability to real data integration settings. Indeed, it is reasonable to assume that users should be able to express most of the SQL2 queries and, consequently, to express data aggregation which is not directly supported by classical DLP. For this reason, in Deliverable D4.1, we have proposed DLP$^{\mathcal{A}}$ an extension of classical DLP allowing to handle aggregation functions. A first contribution of this report is the development of techniques for extending the DLV system so as to implement DLP$^{\mathcal{A}}$.

Recent studies (see, e.g., [1, 12, 2, 5, 3]) showed that query answering in data integration in presence of inconsistent and incomplete data is often complete for the class co-NP, or even for the class Π_2^P. These high computational complexities are mainly caused by the necessity of handling inconsistencies among data stored in different databases. As we have described in previous reports, the basic idea adopted in INFOMIX to solve this problem is to encode the constraints defined on the global schema into a logic program, Π, using unstratified negation or disjunction, such that the stable models of this program yield the repairs of the global database. Answering a user query, $q(x)$, then amounts to cautious reasoning over the logic program Π augmented with the query, cast into rules, and the retrieved facts r.

An attractive feature of this approach is that logic programs serve as executable logical specifications of repair, and thus allow to state repair policies in a declarative manner rather than in a procedural way. However, a drawback of this approach is that with current implementations of DLV, the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability. This calls for suitable optimization methods that help in speeding up the evaluation of queries expressed as logic programs [3]. A possible solution to this problem consists in:

- designing a *localization approach* to reduce complexity, in which irrelevant rules are discarded and the retrieved data is decomposed into two parts: facts which will possibly be touched by a repair and facts which for sure will be not. The idea which is at the heart of the approach is to reduce the usage of the nonmonotonic logic program to the essential part for conflict resolution. This requires some technical conditions to be fulfilled in order to make the part "affected" by a repair small (ideally, as much as possible).
- developing *techniques for recombining* the decomposed parts for query answering, which interleave logic programming and relational database engines. This is driven by the fact that database engines are geared towards efficient processing of large data sets, and thus will help to achieve scalability. In particular, specific parts of

the logic program, especially those not requiring nonmonotonic reasoning, could be instantiated directly on the database engine to improve scalability.

The ideas outlined above are subject of ongoing work and point out two main requirements that the INFOMIX computational logic system must satisfy:

- Optimize as much as possible the instantiation process relative to the logic programs associated with user queries.
- Provide easy and well assessed methods for both an efficient management of data residing in (possibly distributed) databases and the efficient instantiation of input programs directly on database.

As far as the former requirement is concerned, it is possible to devise techniques capable of producing, for each rule of the program, only the relevant subset of all its possible ground instances. One of the main contributions of this report is the description of a structure-based backjumping algorithm we have designed for the instantiation of logic programs that produces "good instantiations" for a given program P. A good instantiation for P is a ground program P' having precisely the same answer sets as P and such that: P' can be computed efficiently from P, and P' does not contain useless rules (P' is as small as possible) and thus can be evaluated efficiently.

In order to satisfy the second requirement we have designed DLV^{DB}, an extension of DLV allowing to solve the database interaction problems mentioned above. As a further contribution of this report we provide a description of the DLV^{DB} architecture and of some associated implementation issues.

As a final remark, all data involved in the integration process must be represented, within the internal integration level, in a uniform way, as expressed by the Internal Integration Data Format (IIDF). However, the specification of the IIDF is out of the scope of this report. This topic is addressed in more detail in Deliverable D6.2.

The plan of the report is the following. The next Section describes the implementation techniques for handling aggregate functions in DLV. Section 3 is devoted to describe the backjumping algorithm implemented in DLV allowing to improve efficiency of program instantiation. Finally, Section 4 describes the extension of DLV for handling database interaction and efficient data manipulation.

2 Implementation techniques for handling aggregate functions

In previous reports we have introduced DLP$^{\mathcal{A}}$, an extension of DLP capable of modelling and reasoning about aggregate data. In this section we focus on the implementation of DLP$^{\mathcal{A}}$ in the DLV system. In order to show how DLV can support aggregate functions, we first give a reminder of the DLP$^{\mathcal{A}}$ syntax and a brief description of the DLV architecture and then we show how this last must be modified for supporting aggregates.

2.1 DLP$^{\mathcal{A}}$ Syntax, a reminder

In this section we give a brief reminder of the DLP$^{\mathcal{A}}$ syntax. For a more complete description of both DLP$^{\mathcal{A}}$ syntax and semantics, the reader is referred to Deliverable D4.1.

A (DLP$^{\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of standard literals. Intuitively, a symbolic set $\{X : a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X : \exists Y s.t.\ a(X,Y), p(Y)\ is\ true\}$. Note that also negative literals may occur in the conjunction *Conj* of a symbolic set.

A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and *Conj* is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name*. Possible function names are #`count`, #`min`, #`max`, #`sum`. An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. Either "$Lg \prec_1$" or "$\prec_2 Rg$" can be omitted. An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol not; if A is an aggregate atom, L is an *aggregate literal*.

A (DLP$^{\mathcal{A}}$) *rule* r is a construct

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

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \ \mathtt{v} \ \cdots \ \mathtt{v} \ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k,$ not $b_{k+1}, ...,$ not b_m is the *body* of r. A DLP$^{\mathcal{A}}$*program* is a set of DLP$^{\mathcal{A}}$ rules.

2.2 The DLV Architecture

The architecture of DLV is illustrated in Figure 1. The general flow in this picture is top-down. The principal User Interface is command-line oriented, but also a Graphical User Interface (GUI) for the core systems and most front-ends is available. Subsequently, front-end transformations might be performed. Input data can be supplied by regular files, and also by Objectivity databases. The DLV kernel (the shaded part in the figure) then produces answer sets one at a time, and each time an answer set is found, "Filtering" is invoked, which performs post-processing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The DLV kernel consists of three major components: The "Intelligent Grounding," "Model Generator," and "Model Checker" modules share a principal data structure, the "Ground Program". It is created by the Intelligent Grounding using differential (and other advanced) database techniques together with suitable data structures, and used by the Model Generator and the Model Checker. The Ground Program is guaranteed to have exactly the same answer sets as the original program. For some syntactically restricted classes of programs (e.g. stratified programs), the Intelligent Grounding module already computes the corresponding answer sets.

For harder problems, most of the computation is performed by the Model Generator and the Model Checker. Roughly, the former produces some "candidate" answer sets

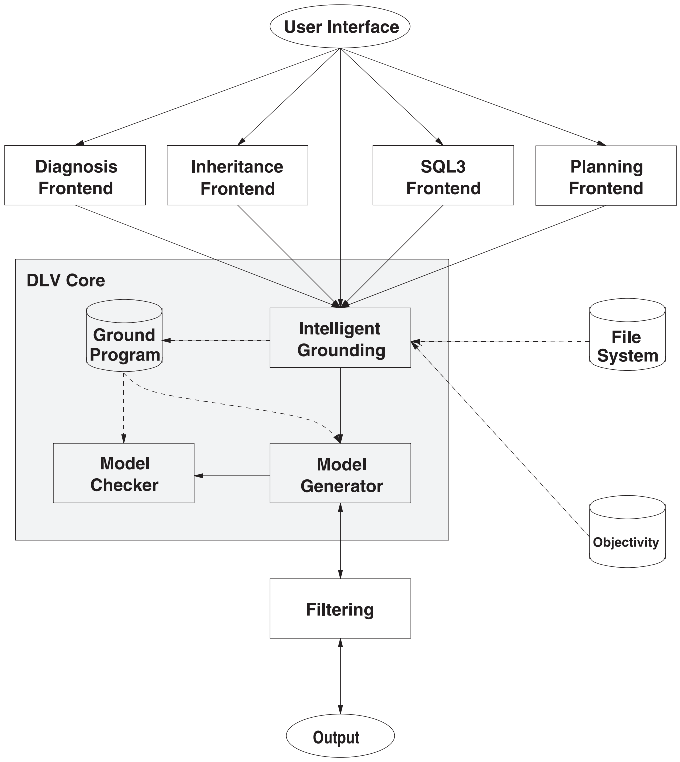

Figure 1: The System Architecture of DLV

(models) [9, 10], the stability and minimality of which are subsequently verified by the latter.

The Model Checker (MC) verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is known to be co-NP-complete. However, MC exploits the fact that minimal model checking — the hardest part — can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs.

2.3 Implementation of aggregates in DLV

The implementation of DLP$^{\mathcal{A}}$ requires changes to all modules of DLV. Apart from a preliminary standardization phase, most of the effort must be concentrated on the Instantiation and Model Generator modules.

Standardization. After parsing, each aggregate A is transformed such that both guards are present and both $\prec_1$ and $\prec_2$ are set to $\leq$. The conjunction *Conj* of the symbolic set of A is replaced by a single, new atom *Aux* and a rule *Aux* :- *Conj* is added to the program (the arguments of *Aux* being the distinct variables of *Conj*).

Instantiation. The goal of the instantiator is to generate a ground program that has precisely the same answer sets as the theoretical instantiation $Ground(\mathcal{P})$, but is sensibly smaller. The instantiation proceeds bottom-up following the dependencies induced by the rules, and, in particular, respecting the ordering imposed by aggregate-stratification where applicable.

For aggregate-stratified components of the input program we proceed as follows. Let "H :- $B, aggr$." be a rule r, where H is the head of the rule, B is the conjunction of the standard body literals in r, and $aggr$ is an aggregate literal over a symbolic set $\{Vars{:}Aux\}$. First we compute an instantiation $\overline{B}$ for the literals in B; this also binds the global variables appearing in Aux. The (partially bound) atom $\overline{Aux}$ is then matched against its extension (which is already available for aggregate-stratified rules since the bottom-up instantiation respects the stratification), all matching facts are computed, and a set of pairs $\{\langle\theta_1(Vars) : \theta_1(\overline{Aux})\rangle,...,\langle\theta_n(Vars) : \theta_n(\overline{Aux})\rangle\}$ is generated, where θ_i is a substitution for the local variables in $\overline{Aux}$ such that $\theta_i(\overline{Aux})$ is an admissible instance of $\overline{Aux}$. (Recall that DLV's instantiator produces only those instances of a predicate which can potentially become true [8, 13], where a ground atom A can potentially become true only if we have generated a ground instance of a rule with A in the head.) Also, we only store those elements of the symbolic set whose truth value cannot be determined yet and process the others dynamically, (partially) evaluating the aggregate already during instantiation. For instance, if Aux is defined by a disjunction-free stratified subprogram (where the truth values of all atoms are already determined during grounding), the aggregate is completely evaluated during the instantiation, its truth valuation is determined, and it is removed from the body of the rule instance. The same process is then repeated for all further instantiations of the literals in B.

Example 2.1 Consider the following rule r:

$$p(X) \text{ :- } q(X), 1 < \#\texttt{count}\{Y : a(X,Y), \text{not} b(Y)\}.$$

The standardization rewrites r to:

$$p(X) \text{ :- } q(X), 2 \leq \#\texttt{count}\{Y : aux(X,Y)\} \leq \infty.$$
$$aux(X,Y) \text{ :- } a(X,Y), \text{not} b(Y).$$

Suppose that the instantiation of the rule for aux generates 3 potentially true facts $aux(1,a)$, $aux(1,b)$, and $aux(2,c)$. If the potentially true facts for q are $q(1)$ and $q(2)$, the following ground instances are generated:

$$p(1) \text{ :- } q(1), 2 \leq \#\texttt{count}\{\langle a : aux(1,a)\rangle, \langle b : aux(1,b)\rangle\} \leq \infty.$$
$$p(2) \text{ :- } q(2), 2 \leq \#\texttt{count}\{\langle c : aux(2,c)\rangle\} \leq \infty.$$

Note that a ground set contains only those aux atoms which are potentially true.

Duplicate Sets Recognition. To optimize the evaluation during instantiation and especially afterwards, we have designed a hashing technique which recognizes multiple occurrences of the same set in the program, even in different rules, and stores them only once. This saves memory (sets may be very large), and also implies a significant performance gain, especially in the model generation where sets are frequently manipulated during the backtracking process.

Example 2.2 Consider the following two constraints:

$$c_1: \quad \texttt{:-} \quad 10 \leq \#\texttt{max}\{V : d(V, X)\}.$$
$$c_2: \quad \texttt{:-} \quad \#\texttt{min}\{Y : d(Y, Z)\} \leq 5.$$

Our technique recognizes that the two sets are equal, and generates only one instance which is shared by c_1 and c_2.

Now assume that both constraints additionally contain a standard literal $p(T)$. In this case, c_1 and c_2 have n instances each, where n is the number of facts for $p(T)$. By means of our technique, each pair of instances of c_1 and c_2 shares a common set, reducing the number of instantiated sets by half.

Model Generation. We have designed an extension of the Deterministic Consequences operator of the DLV system [9] for DLP$^{\mathcal{A}}$ programs. The new operator makes both forward and backward inferences on aggregate atoms, resulting in an effective pruning of the search space. We have then extended the Dowling and Gallier algorithm [7] to compute a fixpoint of this operator in linear time using a multi-linked data structure of pointers. Given a ground set T, say, $\{\langle t_1^1, ..., t_n^1 : Aux^1\rangle, ..., \langle t_1^m, ..., t_n^m : Aux^m\rangle\}$, this structure allows to access T in $O(1)$ whenever some Aux^i changes its truth value (supporting fast forward propagation); on the other hand, it provides direct access from T to each Aux^i atom (supporting fast backward propagation).

3 Improving DLP instantiators for larger data manipulations

The logic-based language supported by the DLV system is quite rich and offers a wide range of syntactical constructs, in order to encode problems from different domains in a very natural and declarative way. Nevertheless, as we have seen in the previous section, the kernel modules of DLV operate on simple ground (i.e., variable-free) programs. Indeed, any given program P first undergoes the so called instantiation process, that computes from P a semantically equivalent ground program P'.

Since this preprocessing phase may be computationally very expensive, having a good instantiation procedure is mandatory. Such a procedure, also called instantiator, should be able to produce a ground program P' with the same answer sets as P such that both: P' can be computed efficiently from P, and P' does not contain useless rules (P' is as small as possible), and thus can be evaluated efficiently by an ASP solver.

The main reason of large grounding programs even for small input programs is that each atom of a rule in a program $\mathcal{P}$ may be instantiated to many atoms in the Herbrand base of $\mathcal{P}$, and this leads to combinatorial explosion. However, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules. A good instantiator should generate ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$.

To this end, the *DLV* instantiator exploits the dependencies among predicates. The instantiation starts by evaluating first the rules defining predicates P_0 that depend on no other predicates (that is, only defined by facts), then the predicates P_1 that only depend on predicates in P_0, and so on. It is worthwhile noting that, if the input program is normal (i.e., `v`-free) and stratified, this instantiator evaluates completely the program and no further module is employed after the grounding; the program has a single answer set, namely the set of the facts and the atoms derived by the instantiation procedure. If the input program is disjunctive or unstratified, the instantiation procedure cannot completely evaluate the program.

Even in this case, at each step of the instantiation process, we have a number of predicates, that we call *solved*, such that the truth values of all their ground instances are already fully determined by the instantiator (each instance of them is already known to be true or to be false, none is undecided - undefined). For instance, all predicates in P_0 are solved, as well as all predicates that only depend on solved predicates. It follows that none of these predicates should occur in the rules (but the facts) of the ground program P' produced by the instantiator. All the predicates occurring in the rules of P' should be unsolved, and will be evaluated by the answer set solver.

Example 3.1 *Consider the following rule*

$$r_1 : a(X,Z) \text{ :- } q_1(X,Z,Y), q_2(W,T,S), q_3(V,T,H), q_4(Z,H), q_5(T,S,V).$$

Suppose we know that predicates q_3, q_4, and q_5 are solved, and consider the following ground instances for r_1:

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_1,t_1,h_1), q_4(z_1,h_1), q_5(t_1,s_1,v_1).$$

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_2,t_1,h_1), q_4(z_1,h_1), q_5(t_1,s_1,v_2).$$

$$\vdots$$

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_{100},t_1,h_{100}), q_4(z_1,h_{100}), q_5(t_1,s_1,v_{100}).$$

Now, assume that all these instances are applicable, that is, all instances of the atoms over the solved predicates q_3, q_4 and q_5 are true, and all instances of the atoms over unsolved predicates (i.e. atoms $q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1)$) could be true (i.e., they are not provably false, at this point). Then, it is easy to see that all these 10000 rules are semantically equivalent to the single instance

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1).$$

Thus, we only need all the (applicable) instantiations of unsolved predicates, while the solved ones are just used to validate such instances. More precisely, we are not interested in finding all the "consistent" substitutions for all variables, but rather their restrictions to the only variables that occur in literals over unsolved predicates. We call such variables the *relevant variables* of a rule r, and any applicable ground instance projected onto the unsolved predicates (as the one shown in the above example) a *relevant instance* for r. These ground rules should be included in any sound instantiation of P.

In this section we present a new kind of structure-based backjumping algorithm that is able to compute efficiently all and only the relevant instances of a rule r. To this end, it exploits both the semantical information on the relevant variables of r, and the structural information on how the literals in the body of r are connected each other through the variables they have in common.

It is worthwhile noting that, unlike other approaches, we do not filter-out useless instances after their computation. Rather, we directly avoid their generation.

3.1 Instantiation of Disjunctive Logic Programs: DLV's Strategy

In this section, we provide a short description of the overall instantiation module of the DLV system, and focus on the "heart" procedure of this module which produces the ground instances of a given rule.

An input program $\mathcal{P}$ is first analyzed from the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see [8]), optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph of $\mathcal{P}'$, its connected components, and a topological ordering of these components. Finally, $\mathcal{P}'$ is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph.

The aim of the instantiator is mainly twofold: (i) to evaluate (`v`-free) stratified program components, and (ii) to generate the instantiation of disjunctive or unstratified components (if the input program is disjunctive or unstratified).

In order to evaluate efficiently stratified programs (components), DLV uses an improved version of the generalized semi-naive technique [15] implemented for the evaluation of linear and non-linear recursive rules.

As previously pointed out, if the input program is normal (i.e., `v`-free) and stratified, the instantiator evaluates completely the program and no further module is employed after the grounding, whereas if the input program is disjunctive or unstratified, the instantiation procedure cannot evaluate completely the program. However, the optimization techniques mentioned above are useful to generate efficiently the instantiation of the non-monotonic part of the program. Two aspects are crucial for the instantiation:

(a) the number of generated ground rules,

(b) the time needed to generate such an instantiation.

The size of the generated instantiation 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 generating a large grounding. However, the time needed by the former can not be ignored otherwise we could not really have a computation time gain.

Rule Instantiation In this section, we describe the process of rule instantiation – the "heart" of the instantiation module – as it is currently implemented in DLV.

```
Algorithm Instantiate
Input  R: Rule, I: Set of instances for the predicates occurring in B(R);
Output S: Set of Total Substitutions;
var  L: Literal, B: List of Atoms, θ: Substitution, MatchFound: Boolean;
begin
    θ = ∅;
    (* returns the ordered list of the body literals (null, L1, ···, Ln, last)  *)
    B := BodyToList(R);        L := L1;
    MatchFound := true;
    S := ∅;
    while L ≠ null
        Match(L, θ, MatchFound);
        if  MatchFound
            if(L ≠ last) ) then
                L := NextLiteral(L);
            else (* θ is a total substitution for the variables of R *)
                S := S ∪ θ;
                L := PreviousLiteral(L);
                MatchFound := false (* look for another solution *)
                θ := θ |PreviousVars(L)
        else
            L := PreviousLiteral(L);
            θ := θ |PreviousVars(L)
    output S;
end;
```

Figure 2: Computing the instantiations of a rule

The algorithm *Instantiate*, shown in Figure 2, generates all the possible instantiations for a rule r of a program $\mathcal{P}$. When this procedure is called, for each predicate p occurring in the body of r we are given its extension, as a set I_p containing all its ground instances. We say that the mapping $\theta : var(r) \rightarrow U_{\mathcal{P}}$ is a valid substitution for r if it is valid for every literal occurring in its body, i.e., if for every positive literal p (resp., negative literal $\neg p$) in $B(r)$, $\theta p \in I_p$ (resp. $\theta p \notin I_p$) holds[1]. *Instantiate* outputs all such valid substitutions for

[1]Actually, for negative literals, we can make this conclusion only in some cases. As an example, if a is a fact and we evaluate $\neg a$, then we can conclude that $\neg a$ is false.

Procedure *Match* (L:Literal, var θ:Substitution, var $MatchFound$: Boolean)
begin
  if $MatchFound$ **then**
    FirstMatch(L, θ, $MatchFound$); (* this is the first try on a new literal *)
  else (* the last match failed, look for another match on a previous literal *)
    NextMatch(L, θ, $MatchFound$);
end;

Procedure *FirstMatch* (L: Literal, var θ: Substitution, var $MatchFound$: Boolean)

(* Look in the extension I_L for the first tuple of values matching θ, and possibly update θ accordingly. The boolean variable MatchFound is assigned True if such a matching tuple has been found; otherwise, it is assigned False. *)

Procedure *NextMatch* (L: Literal, var θ: Substitution, var $MatchFound$: Boolean)

(* Similar to FirstMatch, but finds the next matching tuple. *)

Figure 3: The matching procedures

r, which are in a one-to-one correspondence with the set of all possible ground instances of r.

Note that, since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. For the sake of presentation, we assume that the body is ordered in a way such that any negative literal always follows the positive atoms containing its variables. Actually, DLV has a specialized module that computes a clever ordering of the body (e.g., exploiting the quantitative information on the size of any predicate extension) that satisfies this assumption.

Instantiate first stores the body literals $L_1, \ldots, L_n$ into an ordered list $B = (null, L_1, \cdots, L_n, last)$; *null* and *last* are auxiliary literals exploited in the backtracking procedure to identify the beginning and the end of the body. Then, it starts the computation of the substitutions for r. To this end, it maintains a variable θ, initially set to $\emptyset$, representing, at each step, a partial substitution for $var(r)$.

Now, the computation proceeds as follows: For each literal L_i, we denote by $PreviousVars(L_i)$ the set of variables occurring in any literal that precedes L_i in the list B, and by $FreeVars(L_i)$ the set of variables that occurs for the first time in L_i, i.e., $FreeVars(L_i) = var(L_i) - PreviousVars(L_i)$.

At each iteration of the **while** loop, we try to find a match for a literal L_i with respect to θ. More precisely, if $FreeVars(L_i) \neq \emptyset$, we look for an extension of θ to the variables in $FreeVars(L_i)$; otherwise, we simply check whether θ is a valid substitution for L_i. This is accomplished by the procedure *Match* (Figure 3) that, in turns, calls *FirstMatch* if this is the first attempt to find a match for L_i, or *NextMatch* if we already have a valid substitution for L_i and we have to look for a further one.

If there is no such a substitution, then we backtrack to the previous literal in the list, or else we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise, θ encodes a (total) valid substitution and is thus added to the output set S. Even in this case, we backtrack for finding another solution, since we want to compute *all* instantiations of r.

Note that this kind of classical backtracking procedure works well for rules with a few literals and with a few tuples for each predicate extension. However, DLV has been designed to work even for manipulating complex knowledge on large databases, and for such applications the simple algorithm described above is not satisfactory.

Example 3.2 *Suppose we want to compute all ground instantiations of the rule*

$$r2: \ a(X,Y) \text{ :- } p_1(X,Y), p_2(X,Z), p_3(Z,H,T), p_4(T,W), p_5(X,V,Z), p_6(X,Y,V).$$

and that we have already computed a partial substitution θ for the variables $\{X,Y,Z,H,T,W\}$, but we are not able to find a consistent value for V in the extension of p_5, in order to extend θ. In this case, according to the algorithm in Figure 2, we should backtrack to the previous literal p_4. However, the failure on atom $p_5(X,V,Z)$ is independent of variables $\{H,T,W\}$, and thus we should just find another possible value for Z. It follows that, intuitively, we could safely backtrack directly to atom $p_2(X,Z)$, where this variable has been instantiated. Thus, jumping over both $p_3(Z,H,T)$ and $p_4(T,W)$ can allow us to save a very large amount of time, especially if the extensions of p_3 and p_4 contain many tuples.

To overcome these kinds of troubles, a number of extensions of the backtracking technique have been described in the literature – see, e.g., the *intelligent backtracking* technique developed in the logic programming community [4], or the various backjumping techniques proposed for solving constraint satisfaction problems (CSPs) [14]. Our rule instantiation problem is closer to CSP, however most of these algorithms focused on problems with just binary constraints, and looking for just one solution. On the contrary, in our context, we need a specialized algorithm that should be able to compute efficiently all instantiations of a rule with predicates of arbitrary arity, which corresponds to finding all solutions of general (non-binary) constraint satisfaction problems.

3.2 A Backjumping Technique for DLP-Programs Instantiation

In this section, we describe the Algorithm *BJ_Instantiate*, that given a rule r and a set of relevant variables *OutputVars*, returns a set of substitutions for these variables that are in a one-to-one correspondence with all and only the ground instances of r we are interested in. That is, we do not generate all those ground instances of r that differ only on non-relevant variables. Formally, *BJ_Instantiate* returns the projections on *OutputVars* of all the valid substitutions for r. We call these substitutions the *relevant solutions* of our problem.

The basic schema of this algorithm is no more the classical backtracking paradigm, but rather a structure-based backjumping paradigm, well studied in the constraint satisfaction area (see., e.g., [6, 14]). In these kind of algorithms, when some backtrack step is necessary,

Algorithm *BJ_Instantiate*
Input R: Rule, I: Set of instances for the predicates occurring in $B(R)$,
$OutputVars$: Set of Variables;
Output S: Set of Substitutions;
var L: Literal, B: List of Atoms, θ: Substitution, CSB: Literal, $Status$: MATCH_STATUS;
begin

```
θ = ∅;
(* returns the ordered list of the body literals (null, L_1, ···, L_n, last) *)
B := BodyToList(R);
L := L_1;
Status := SuccessfulMatch;
CSB := null;
S := ∅;
while L ≠ null
    Match(L, θ, Status);
    switch (Status)
        case SuccessfulMatch
            if(L ≠ last) ) then
                L := NextLiteral(L);
            else (* θ is a total substitution for the variables of R *)
                S := S ∪ θ |_OutputVars;
                L := BackFromSolutionFound(L, CSB, Status);
                θ := θ |_PreviousVars(L)
            break;
        case FailureOnFirstMatch
            L := BackFromFailureOnFirstMatch(L, CSB);
            θ := θ |_PreviousVars(L)
            break;
        case FailureOnNextMatch
            L := BackFromFailureOnNextMatch(L, CSB);
            θ := θ |_PreviousVars(L)
            break;
output S;
```

end;

Figure 4: The algorithm BJ_Instantiate

enum MATCH_STATUS = {SuccessfulMatch, FailureOnFirstMatch, FailureOnNextMatch};

Procedure *Match* (L:Literal, var θ:Substitution, var *Status*: MATCH_STATUS)
begin
    if *Status* = SuccessfulMatch **then** (* the last match was successful *)
        FirstMatch(L, θ, *Status*); (* this is the first try on a new literal *)
    else (* the last match failed, look for another match on a previous literal *)
        NextMatch(L, θ, *Status*);
end;

Procedure *FirstMatch* (L: Literal, var θ: Substitution, var *Status*: MATCH_STATUS)

(* Look in the extension I_L for the first tuple of values matching θ, and possibly update θ accordingly. *Status* is assigned SuccessfulMatch if such a matching tuple exists; otherwise, it is assigned FailureOnFirstMatch *)

Procedure *NextMatch* (L: Literal, var θ: Substitution, var *Status*: MATCH_STATUS)

(* Similar to FirstMatch, but finds the next matching tuple. In case of failure, *Status* is set to FailureOnNextMatch *)

Figure 5: Matching procedures for *BJ_Instantiate*

it is possible to jump more than one element, rather than just one, as in the standard chronological algorithm. Of course, such jumps should be designed carefully, in order to avoid that some solution is missed, especially in our case, where we have to compute all solutions.

Let r be a rule and B the ordered list of its body literals $(null, L_1, \cdots, L_n, last)$. We say that L_i $(1 \leq i \leq n)$ is a *binder* for a variable X if there is no literal L_j, with $1 \leq j < i$ such that $X \in var(L_j)$. Moreover, for a set of variables V and a literal L_k, let *ClosestBinder*(L_k,V) denote the greatest literal L_i $(i < k)$ among the binders of the variables in V. A crucial notion in our algorithm is the *Closest Successful Binder* (CSB), which represents, intuitively, the greatest literal that is a binder of some variable X whose current assigned value belongs to the last computed solution. The CSB acts as a barrier for some kind of jumps, as described later in this section.

Another important point is the structure of the relationships among the literals in the body. We say that, for any pair of literals L_i, L_j in B, $L_i \prec_d L_j$ if $i \leq j$ and $var(L_i) \cap var(L_j) \neq \emptyset$. Let $\prec$ denote the transitive closure of the $\prec_d$ relationship and, for any literal L in B, let $dep(L) = \bigcup_{\{L' | L \prec L'\}} var(L')$. Intuitively, this is the set of variables that depends on the instantiation of the literal L, and we refer to it as the *dependency set* of L.

Example 3.3 *As a running example in this section, consider the following rule*

$$r_3 : a(X, Y, Z) \mathrel{:-} q_1(X, T, W), q_2(X, Y), q_3(Z, S), q_4(Z, V), q_5(T, H), q_6(H, T, V).$$

It is easy to check that the dependency set of literal $q_5(T, H)$ is $\{T, H, V\}$, while the dependency set of $q_3(Z, S)$ is $\{Z, S, V, H, T\}$.

In order to instantiate r_3, our algorithm needs the additional information on the relevant variables and the already known instances for the predicates occurring in the body. Then, assume that OutputVars $= \{X, Y, Z, T, W\}$, and that we are given the following extensions for the predicates occurring in $B(r_3)$:

$$\begin{array}{llllll} q_1(x_1, t_1, w_1) & q_2(x_1, y_1) & q_3(z_1, s_1) & q_4(z_1, v_1) & q_5(t_2, h_1) & q_6(h_2, t_2, v_1) \\ q_1(x_1, t_2, w_1) & q_2(x_1, y_2) & & q_4(z_1, v_2) & q_5(t_2, h_2) & q_6(h_2, t_2, v_2) \end{array}$$

Figure 4 shows the algorithm *BJ_Instantiate*. As for Algorithm *Instantiate*, at each iteration of the **while** loop, the procedure *Match* tries to find a match for a literal L_i with respect to the current partial substitution θ. If it succeeds and L_i is not the last literal, then we can proceed with the next literal L_{i+1}. Otherwise, we have to backtrack, and thus we have to decide where to jump and, possibly, update the current CSB. Now, we have a number of different cases to be handled, depending on the outcome *Status* of the procedure *Match*.

1. **Success, and θ encodes a total substitution**. Since also the match on the last literal is successful, θ encodes a valid substitution for the variables in r, and its restriction to *OutputVars* is therefore added to the set of solutions. Then, in order to look for further solutions, we have to backtrack. However, in this algorithm, we are not forced to go back to the previous literal. Rather, we can jump to the closest literal L_j binding a variable of interest, that is, jump to *ClosestBinder*(*last*,*OutputVars*). Moreover, in this case the CSB is set to L_j.

   Example 3.4 *In our running example, the algorithm is able to find the total substitution $\theta(X) = x_1$, $\theta(Y) = y_1$, $\theta(Z) = z_1$, $\theta(T) = t_2$, $\theta(W) = w_1$, $\theta(S) = s_1$, $\theta(V) = v_1$, and $\theta(H) = h_2$. That is, we have a match for all the literals in B and we are at last. Then, the restriction of θ to the set of relevant variables is added to S. In our case, this solution corresponds to the following instance of r_3:*

   $$a(x_1, y_1, z_1) \coloneq q_1(x_1, t_2, w_1), q_2(x_1, y_1), q_3(z_1, s_1).$$

   Now, according to the algorithm, we jump back to $q_3(Z, V)$ for finding other solutions. Note that we do not look for further consistent tuples in the extensions of q_4, q_5, and q_6, because they do not bind any relevant variable. Indeed, possible solutions coming from other instances of these predicates (e.g., the solution with $\theta(V) = v_2$) would just lead to useless rules in the instantiation of the program at hand. Finally, the CSB is set to q_3.

2. **Failure at the first attempt to find a match for a literal L_i**. We jump back to the closest literal L_j binding any of the variables in L_i, that is, jump to *ClosestBinder*(L_i,*var*(L_i)). Indeed, in this case, the only way for finding a match for L_i is to change the assignment for some of its bound variables. Moreover, if L_j precedes CSB, then we can push back CSB to L_j. This will make the next type of jumps less restrictive, see case 3 below.

Function *BackFromFailureOnFirstMatch* (L: literal, var CSB: Literal) : Literal;
begin (* the first match on a new literal failed *)
    L' := *ClosestBinder*(L, *Vars*(L));
    if $L' \prec CSB$ **then**
        $CSB := L'$;
    return L';
end;

Function *BackFromFailureOnNextMatch*(L: Literal, var CSB: Literal) : Literal;
begin (* failure looking for another match for L *)
    if $L = CSB$ **then**
        CSB := *ClosestBinder*(L,OutputVars);
    L' := *ClosestBinder*(L,*DepVars*(L))
    return $\max_{\prec}\{L', CSB\}$;
end;

Function *BackFromSolutionFound*(L: Literal, var CSB: Literal, var *Status*: MATCH_STATUS)
        : Literal;
begin
    Status := FailureOnNextMatch; (* look for another solution *)
    CSB := *ClosestBinder*(L,*OutputVars*);
    return CSB;
end;

Figure 6: Backjumping procedures for *BJ_Instantiate*

Example 3.5 *In our running example, the first time that we try to find a match for q_5, we have computed the partial substitution $\theta(X) = x_1$, $\theta(Y) = y_1$, $\theta(Z) = z_1$, $\theta(T) = t_1$, $\theta(W) = w_1$, and $\theta(S) = s_1$. In this case, we are not able to find any matching instance in the extension of q_5. Indeed, none of its instances has a value t_1 for variable T. Then, we have to change the value assigned to one of the variables occurring in q_5, and thus we can safely jump over q_4, q_3, and q_2, and try to match again $q_1(X, T, W)$. Indeed, q_1 is the closest binder for $var(q_5)$, as it determines the value for variable T.*

3. **Failure while looking for another match for a literal** L_i. In this case, L_i is a binder of some set of variables $\bar{X}$, and we fail in finding a different consistent substitution for these variables. Since we were successful on our first attempt to deal with L_i, this means that, for some reason, we jumped back to L_i from some later item, say L_j, of the list B. Now, we have to decide where to jump after the current failure, and this time the variables occurring in L_i are not the only candidates to be changed. Rather, we have to look at the dependency set of L_i, as shown below.

Example 3.6 *Assume that, in our running example, we are looking for another match for $q_3(Z,S)$ and that the CSB is set to $q_1(X,T,W)$. According to the algorithm, we have to jump to q_1, even if it is not a binder for any variable occurring in q_3. The reason is that q_1 is a binder for T, which belongs to the dependency set of q_3, and changing its value may lead to some new solution (possibly comprising values already considered for the variables occurring in q_3).*

Another important issue concerns the management of the CSB. First, we check whether the current literal L_i coincides with the CSB. If this is the case, we push back the CSB to *ClosestBinder*(L_i,*OutputVars*). In this case, it acts as a barrier and cannot be jumped, otherwise we can miss some relevant solution as the following example shows.

Example 3.7 *Let us continue from the execution step described at point 1 above, where we have found our first solution. Recall that we jumped back to $q_3(Z,S)$ and the CSB is set to this literal. In this case, the CSB is first pushed back to $q_2(X,Y)$, which is the* ClosestBinder*($q_3(Z,S)$,OutputVars). Then, even if according to the dependency set we could jump to q_1, we are forced to stop our jumping back to literal $q_2(X,Y)$, because of the CSB limit. It is worthwhile noting that, if we go directly to $q_1(X,T,W)$, we miss the solution obtainable by assigning y_2 to variable Y and corresponding to the following instance of r_3:*

$$a(x_1, y_2, z_1) \text{ :- } q_1(x_1, t_2, w_1), q_2(x_1, y_2), q_3(z_1, s_1).$$

Theorem 3.8 *Algorithm* BJ_Instantiate *is sound and complete. That is, given a rule r, the ground instances for the predicates occurring in its body, and the set of its relevant variables OutputVars,* BJ_Instantiate *computes the set containing all and only the projections over OutputVars of the valid substitutions for r.*

4 Interfacing DLV with DBMSs

As pointed out in the previous sections, many interesting problems are either "data intensive" or they produce huge ground programs that might not be handled in main-memory data structures. As a consequence, DLV needs efficient and effective data management techniques. Moreover, in data integration contexts, input data is usually stored in (possibly distributed) databases.

These considerations put into evidence the need of combining the expressive power of DLV with the efficient data management features of DBMS*s*. Indeed, DLV provides an expressive power that goes far beyond that of SQL3, whereas good DBMS*s* provide very efficient query optimization mechanisms (like join orderings [11]) and distributed data management techniques.

In this section, we present DLV^{DB} which provides a first, initial contribution in this setting, bridging the gap between ASP systems and DBMS*s*. It is an extension of the DLV system which allows two kinds of program execution: *(i) main-memory execution* in which data residing in (distributed) databases are first loaded in main-memory and then standard DLV is executed, and *(ii) direct database execution*, in which both the instantiation of input programs directly on database and the management of input and output data distributed on several databases are handled.

DLV^{DB} combines the experience in optimizing ASP programs gained within the DLV project with the well assessed data management capabilities of existing DBMS*s*.

The main-memory execution fully supports the DLV language, whereas in *direct database execution* only a small subset of the DLV language, namely that constituted by stratified normal programs with built-in predicates, is supported; however, DLV^{DB} provides a well established infrastructure for the interoperation with databases and allows the application of several optimization techniques already developed or under development in the DLV project.

4.1 Main-Memory Execution

The main-memory execution of DLV^{DB} instantiates the input logic program in main-memory (and, consequently, it exploits the standard DLV instantiator module), but allows input facts to be (possibly complex) views on database tables; moreover, it allows to export (parts of) predicates to database. The interaction between DLV and the database is performed by exploiting ODBC features.

In order to perform these tasks, we introduce two built-in commands in the DLV syntax, namely the #import and the #export command. These are described next.

#import. The syntax of an #import command is the following:

#import(databasename,username,password,"query",predname, "typeConv").

where:

- *databasename* is the name of the database server;
- *username* identifies the user trying to connect to the database;
- *password* is a string defining the password needed by *username* for accessing *databasename*;
- *query* is a string defining the SQL statement to be executed;
- *predname* indicates the name of the relational predicate to import as atom.
- *typeConv* specifies the data conversion rules to be applied for converting database types into DLV data types. The syntax of the typeConv parameter is the following: "type : Conv [, Conv]", where *type* is a string constant, and *Conv* presents one among the following five kinds of conversion:
  - U_INT : the column is converted to an unsigned integer;
  - UT_INT: the column is rounded to an unsigned integer;

- UR_INT: the column is truncated to an unsigned integer;
- CONST: the column is converted to a string without quotes;
- Q_CONST: the column is converted to a string with quotes;

The number of specified conversions must be equal to the number of columns of the selected table.

An #import command retrieves data from a table "row by row" through the SQL query specified by the user and creates one atom for each selected tuple. The name of each imported atom is set to predname, and is considered as part of the EDB program. Further EDB predicates can be added by creating text files to feed into DLV.

Example 4.1 *The following is a program which connects to a database via ODBC, reads all of the data from the table* employee *and stores them as facts (EDB program) with predicate name* emp. *The database name is* postgres*, access is granted to the user* dlvusr *with password* dlv.

#import(postgres,dlvusr,dlv ,"SELECT id, Name, FROM employee",emp,
"type : U_INT, Q_CONST")

#export. The syntax of an #export command is the following:

#export(databasename,username,password,predname,tablename).

where:

- *databasename* is the name of the database server;
- *username* defines the the name of the user trying to connect to the database;
- *password* is a string defining the password needed by *username* for accessing *databasename*;
- *predname* identifies the name of the atom to export into the database;
- *tablename* indicates the table in which data must be copied into;

Each time an answer set M of the associated program is computed, the export command generates a new tuple into *tablename* for each atom in M having name *predname*.

Example 4.2 *The following program is the encoding of the well known 3-coloring problem. Given an undirected graph, represented by facts of the form* edge(X,Y)*, it assigns to each node one among three colors in such a way that no two adjacent nodes have the same color. Assignments computed in the model(s) of the input program are defined by the predicate colored.*

The program shows how to export all of the atoms with predicate name colored *belonging to a stable model, into the relational table named* colors *of the datasource* postgres. *Access is granted to the user* dlvusr *with password* dlv.

$$\#export(postgres, dlvusr, dlv, colored, colors)$$
$$node(X) \text{ :- } edge(X,Y).$$
$$node(Y) \text{ :- } edge(X,Y).$$
$$colored(X, red) \texttt{ v } colored(X, green) \texttt{ v } colored(X, blue) \text{ :- } node(X).$$
$$\text{:- } edge(X,Y), colored(X,C), colored(Y,C).$$

An alternative form of the #export command is the following:

#export(databasename,username,password,predname,tablename,
"REPLACE where <condition>")

which is used to remove from *tablename* the tuples of *predname* for which the "REPLACE where" condition holds. This can be useful for deleting tuples corresponding to violated integrity constraints.

4.2 Direct Database Execution

4.2.1 Supported features

Presently DLV^{DB} allows to handle normal stratified programs; in particular both true negation and negation as failure are supported. Moreover, all built-in predicates allowed in DLV are also supported in DLV^{DB}. Program safety is a necessary condition for the correct execution of the instantiation process; finally, identification of hard constraints violations is also supported even if, at present, the system just warns the user about the existence of these violations.

Since DLV^{DB} has been designed mainly for applications on (distributed) databases, it allows the user to specify some auxiliary directives for handling the interaction between DLV^{DB} and a set of databases. The grammar in which these directives must be expressed is shown in Figure 7.

The auxiliary directives can be subdivided in three sections, namely the *init section*, the *table definition* section and the *final section*.

The *init section* allows to define the working database on which the instantiation process will be carried out.

The *table definition* section allows to specify the mappings between program predicates and database tables. The user can exploit two options in this phase, namely the USE and the CREATE option.

- The USE option must be exploited if the table *TableName* already exists and the data it contains must be exploited as input data (facts) for the instantiation process. Note that this table might reside on a database different from the one exploited for the instantiation (this situation can be specified with the option [FROM DatabaseName]).

```
Auxiliary-Directives ::= Init-section [Table-definition]+ [Final-section]*

Init-Section ::= USEDB DatabaseName;

Table-definition ::=
   [USE TableName ( Attribute [, Attribute]* [, Sql-constraint]? )
   [FROM DatabaseName]?   MAPTO  PredName;]
   |
   [CREATE TableName ( Attribute [, Attribute]* [, Sql-constraint]? )
   [AS ("SQL-Statement")]? [FROM DatabaseName]? [MAPTO  PredName]?
   [KEEP_AFTER_EXECUTION]?; ]

Final-section  ::= [OUTPUT    DatabaseName;]?
   |
   [OUTPUT  [APPEND | OVERWRITE]? TableName [AS AliasName]?
   [IN  DatabaseName]?;]*
```

Figure 7: Grammar of the auxiliary directives.

- The CREATE option must be exploited when the table must be created (if a table with the same name is present in the database it is first removed and then redefined) in the database; in this case the user has the possibility to fill the table with some data by defining an *SQL-Statement* representing a view on other tables. Also in this case tables might reside on a database different from the one exploited for the instantiation. If the user specifies the option KEEP_AFTER_EXECUTION, the table is kept in the database at the end of the instantiation process; otherwise it is removed.

In both cases, the MAPTO option must be exploited to link the table to a predicate. The number of attributes of the table must be equal to the arity of the associated predicate. In the definition of tables, attribute type declaration is needed only for a correct management of built-in predicates.

The *final section* allows either to copy the entire output of the instantiation on a database different from the working database or only some of the tables. If the user does not specify any table name, the entire output is copied; otherwise, only the specified tables are copied. In this latter situation, the user can choose to append the data created during the instantiation to those already present in the target table (by the option APPEND) or to overwrite the (possibly existing) data in the target table (by the option OVERWRITE); if no option is specified, OVERWRITE is the default. The definition of the target table is the same as the source one except for the name which can be changed to *AliasName*.

It is worth pointing out that these directives provide DLV^{DB} with high flexibility in source data location and output management.

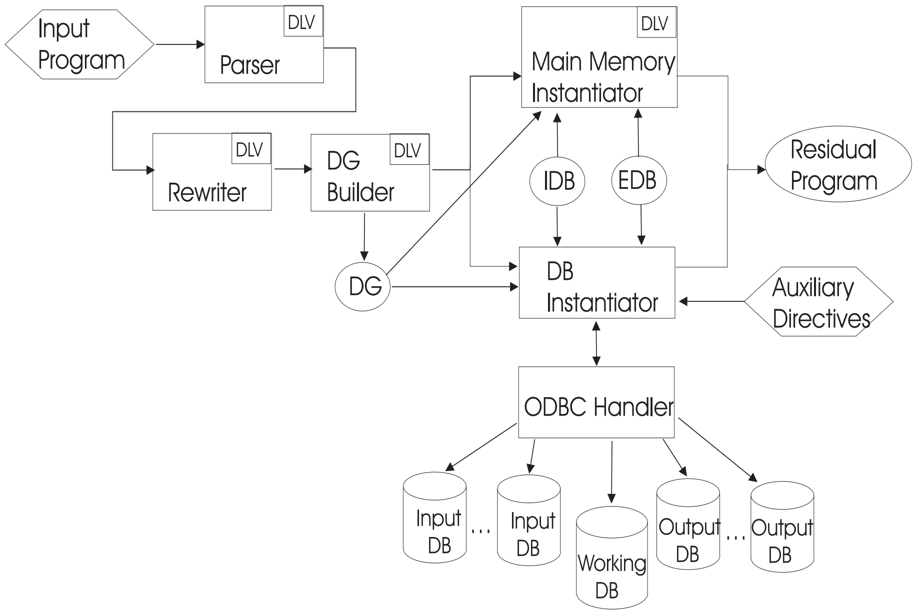

Figure 8: Architecture of the instantiator module of DLV^{DB}.

4.2.2 Architecture

In this section we describe the architecture of DLV^{DB}. It has been developed following the general philosophy of reusing as much as possible existing modules of the DLV system. This allows us to take advantage of optimization techniques, such as program rewritings [8], already developed for DLV.

Figure 8 illustrates the architecture of the instantiator module of DLV^{DB}. In the figure, the boxes marked with DLV are the ones already developed in the DLV system. An input program $\mathcal{P}$ is first analyzed from the parser which encodes the rules in the intensional database (IDB) in a suitable way. Some of the facts can be stored directly in the program; in this case, they are first stored in a main-memory extensional database (EDB) by the parser. Then, a rewriting procedure optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. The dependency graph builder computes the dependency graph of $\mathcal{P}'$, its connected components and a topological ordering of these components. Finally, the DB Instantiator module is activated.

The DB Instantiator module receives: *(i)* the IDB and the EDB (if not empty) generated by the parser, *(ii)* the Dependency Graph (DG) generated by the dependency graph builder and *(iii)* some auxiliary directives specifying the interaction between DLV^{DB} and the databases. Since the input program is supposed to be normal and stratified, the DB Instantiator evaluates completely the program and no further module is employed after the grounding.

All the instantiation steps are performed directly on the working database through the execution of SQL statements and no data is loaded in main-memory in any phase of the instantiation process. This allows DLV^{DB} to be completely independent from the dimension of both the input data and the produced ground program and, consequently, provides high scalability.

Input data (the facts) can reside in several input databases accessible through the internet; however, in order to improve efficiency, they are copied into the local working database before the instantiation phase is started. Finally, the results (or parts thereof) of the computation can be transferred to some output databases.

Communication with databases is performed via ODBC. This allows both to be independent from a particular DBMS and to handle distributed databases.

4.2.3 Implementation Issues

The DB Instantiator first maps all the predicates into the corresponding database tables, as specified by the auxiliary directives; default mappings are applied when needed. Then it stores the facts present in the EDB into the corresponding database tables. Note that DLV^{DB} allows some or all the facts to be already stored in the database; if this happens, they are not loaded in the main-memory EDB by the parser.

After all the mappings and input data have been prepared, the instantiation starts. Presently, DLV^{DB} adopts the simple Naive algorithm [15] for the instantiation. For the first phase of the implementation, we have chosen a simple instantiation algorithm to concentrate our attention on both the translation of DLV rules into SQL statements and the management of the interoperation between DLV^{DB} and the involved databases. However, the extension to more efficient semi-naive approaches will be quite straightforward. The instantiation process proceeds as follows: first the dependendy graph (DG) is computed from the dependency graph builder; DG is exploited for determining the connected components of the program, and a topological ordering of these components. Then, the program is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component according to the dependency graph. This process is iterated until to no new ground instance can be derived from the components.

At each iteration, the instantiation of a rule consists in the execution of the SQL statement representing it. One of the main objectives in the implementation of DLV^{DB} must be that of associating one single SQL statement with each rule of the program, without the support of main-memory data structures for the instantiation. This allows DLV^{DB} to fully exploit optimization mechanisms implemented in the DBMSs and to minimize the “out of memory” problems caused by limited main-memory dimensions.

The main guidelines for the translations are the following: *(i)* variable bindings determine join operations among tables; *(ii)* the SELECT part of the SQL statement is determined by the variables in the head of the rule and the associated bindings, *(iii)* the FROM part of the statement is determined by the predicates composing the body of the rule; *(iv)* variable bindings, negation and built-in predicates are handled in the WHERE part of the statement; *(v)* the INSERT INTO section is determined by the predicate in the head of the rule.

In the following we show the translation from DLV rules to SQL statements by some examples.

Translation of simple positive rules. Consider the rule

$p(X, Y) :- a(X, Z), b(Z, Y).$

It is instantiated by the SQL statement[2]:

```
INSERT INTO p ( SELECT a.att1, b.att2 FROM a, b
                                          WHERE a.att2 = b.att1 )
```

Translation of rules with true negation. True negation is handled as in the previous case, by considering negated atoms as tables in the database.

Translation of rules with negation as failure. Consider the rule

$p(X, Y) :- c(X, Y, Z), not\ d(Z).$

It is instantiated by the SQL statement:

```
INSERT INTO p ( SELECT c.att1, c.att2 FROM c WHERE (c.att3)
                                  NOT IN (SELECT d.att1 FROM d))
```

Translation of rules with built-ins. It is worth pointing out that when running a program containing built-in predicates, the range of admissible numbers must be fixed. We map this necessity in the working database by adding a support table *int* containing the allowed values for integer variables. Moreover, in order to allow mathematical operations among attributes, DLV^{DB} requires the types of attributes to be correctly defined in the database. DLV^{DB} supports all the built-in predicates allowed in DLV, namely $<, >, <=, >=, ==, !=, \#int, \#succ, +, *$.

Consider the rule

$p(X, Y) :- a(X, Z1), a(Y, Z2), \#succ(Z1, Z2)$

it is instantiated by the SQL statement:

```
INSERT INTO p ( SELECT a1.att1, a2.att1 FROM a a1, a a2
                                 WHERE a2.att2 = (a1.att2 + 1) )
```

If the variables specified in the built-ins are not bound to any other variable of the atoms in the rule body, the support table *int* is exploited to bound that variable to its admissible range of values.

[2]Here and in the following we use the notation $t.att_i$ to indicate the i-th attribute of the table t. Actual attribute names are determined from the auxiliary directives.

References

[1] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Specifying and querying database repairs using logic programs with exceptions. In *Proc. of the 4th Int. Conf. on Flexible Query Answering Systems (FQAS 2000)*, pages 27–41. Springer, 2000.

[2] Pablo Barceló and Leopoldo Bertossi. Reapairing databases with annotated predicate logic. In *Proc. the 10th Int. Workshop on Non-Monotonic Reasoning (NMR 2002)*, pages 160–170, 2002.

[3] Loreto Bravo and Leopoldo Bertossi. Logic programming for consistently querying data integration systems. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, 2003. To appear.

[4] M. Bruynooghe and L. Pereira. Deduction revision by intelligent backtracking. *Implementations of Prolog*, pages 194–215, 1984.

[5] Andrea Calì, Domenico Lembo, and Riccardo Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. of the 18th Int. Joint Conf. on Artificial Intelligence (IJCAI 2003)*, 2003. To appear.

[6] R. Dechter. Enhancement schemes for constraint processing:backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41, 1990.

[7] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[8] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[9] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing Goal Derivation in DLP Computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

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

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[12] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *Proc. of the 17th Int. Conf. on Logic Programming (ICLP'01)*, volume 2237 of *Lecture Notes in Artificial Intelligence*, pages 348–364. Springer, 2001.

[13] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI). Springer Verlag, September 2001.

[14] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[15] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.