A PROLOG-Relational DBMS interface using delayed evaluation [1]

F.Cuppens R.Demolombe

ONERA-CERT
2 Av. E.Belin, BP3025
Toulouse Cedex
France

Abstract

We present a PROLOG-Relational interface based on the loose coupling approach. The interface acts in two steps. In the first one the interface determines the largest sub-sets of the PROLOG program which can be evaluated by the Relational DBMS, the corresponding Relational queries are evaluated, and their answers are inserted in the program, replacing the corresponding program subsets. In the second step the transformed PROLOG program is executed as any standard program, whithout any access to the Relational DBMS.
To determine the program subsets evaluable by the Relational DBMS, we use a modified PROLOG interpreter in which the goal evaluations, for goals expressed by predicates defined in the Data Base, are replaced by Relational query construction.
The paper presents important improvements with regard to similar works. Specially queries are generated which contain sub-queries connected by the operators AND, OR, NOT, or are existentially quantified.

1 Introduction

The most important difference between Deductive Databases and standard Relational Database is the possibility to use, in addition of sets of facts, sets of rules, to derive new facts through a derivation process.

This derivation process can be implemented in two different ways: the first one is to design and to implement it from scratch, the second one is to use an existing inference mechanism. Each one has his advantages and drawbacks, and we have choosen to implement the second one in order to be able to compare it, through an effective implementation, with the other one.

Since PROLOG is one of the most popular languages based on an inference mechanism, and many expert systems are implemented in PROLOG, we have selected this language to express the rules, and a PROLOG interpreter as an inference mechanism.

[1] This work has been supported by the ESPRIT project P316: ESTEAM

2 Approaches for coupling PROLOG and a Relational DBMS

There are, roughly speaking, two different approaches for coupling a PROLOG interpreter and a Relational DBMS, which are usually called "tight coupling" and "loose coupling".[Jarke 84]

In the tight coupling approach the PROLOG interpreter and the Relational DBMS are strongly integrated. For example the PROLOG interpreter can directly use low level functionalities of the DBMS, like relation management in secondary memory, and relation access via indices.

In the loose coupling approach the Relational DBMS is called by the PROLOG interpreter at the top level, that is like a standard user. It sends Relational queries to the DBMS, and the corresponding answers are treated as ground clauses by the interpreter.

The interface which is presented here follows the loose coupling approach. However even if we are in this case we can distinguish two sub-cases corresponding to what we call the "static coupling" approach, and the "dynamic coupling" approach.

In the static approach the queries to be evaluated by the

DBMS are determined and evaluated **before** running the PROLOG interpreter, while in the dynamic approach the queries are evaluated by the DBMS **during** the PROLOG interpreter execution.

For the prototype which is presented in the following we have choosen the static approach, but our interface could also be used, with small modification of the PROLOG interpreter, for the dynamic approach.

- **Loose Coupling**
- **Tight Coupling**
  - **Dynamic coupling**
  - **Static coupling** $\longleftarrow$ **(our approach)**

To design the sofware module, called "Interface", which interfaces the PROLOG interpreter and the Relational DBMS we adopted the following guidelines:

- the Relational queries sent to the DBMS should correspond to the largest sub-sets of the PROLOG program which can be independently evaluated by the DBMS
- the Relational queries are not transformed by the Interface in order to optimise query evaluation; this optimisation is supposed to be done by the Relational DBMS
- neither the Prolog interpreter, nor the Relational DBMS software modules are modified; the consequence of this requirement is that the Interface does not need to know how these two modules are implemented. Moreover there is no need to do any modifications in these modules when they are interfaced. This aspect is very important in practice because usually the companies saling DBMSs don't allow any change in these systems.

3 Architecture of the Interface

The software components which interact with the Interface, and their functionalities are presented below (see figure 1):]

1. the INTERFACE itself analyses the Prolog program, plus a given query, and generates a set of Relational queries according to the method presented in section 5; the subsets of the PROLOG program corresponding to the queries are replaced by calls to new predicates used to represent the answers to these queries; these queries are represented in a language very close to Predicate Calculus, according to a syntax defined in section 6;
2. then the queries are translated into the syntax of the Relational DBMS query language by the TRANSLATOR module;
3. the queries are evaluated by the Relational DBMS and the answers are transmitted to the TRANSLATOR;
4. the answers are transformed into ground clauses relative to the new predicates, according to the PROLOG syntax, by the TRANSLATOR;
5. then the INTERFACE inserts these clauses into the PROLOG program and run the PROLOG interpreter which computes the answer.

In reality, at this moment, the Translator is not implemented, and the Relational DBMS and the Translator are simulated by a Lisp program.

The general idea used for the INTERFACE implementation is to modifiy a PROLOG interpreter in order to replace the **resolution** of some goals, non-recursively defined in function of Data Base predicates, by the **generation** of corresponding queries. In some sense the evaluation of these goals is delayed until the queries are evaluated by the Relational DBMS. That will be explained in the next section. Notice that the PROLOG interpreter which is run in the 5th step is not modified.

4 Methodology used by the Interface

4.1 Definition

We will use the following definitions:

Base Predicate: predicate whose extension is stored in the Relational Data Base; for each Base Predicate P, the PROLOG program must contain a clause of the form:

$$P(\ldots) \longleftarrow edb\text{-}P(\ldots)$$

These clauses allow the INTERFACE to recognize the Base Predicates.

Virtual Predicate: predicate whose definition contains only Base Predicates or Virtual Predicates. A predicate definition is the set of clause bodies, having this predicate in the head.

Recursive Predicate: predicate which appears in a loop in the program connection graph.

Non-recursive virtual predicate: virtual predicate which is not recursive.

Evaluable Predicate: predicate which is evaluated by a LISP function; an evaluable predicate is prefixed by *eval*, like *eval-P*.

A pre-treatement of the PROLOG programs, whose details are not presented here, allows to recognize the type of each predicate.

4.2 Interface Environment

The PROLOG interpreter used by the Interface is PROLISP, an interpreter developed at ONERA-CERT in a LISP environment. The *eval* function allows to evaluate any LISP function, the arguments and the result of which are PROLOG variables.

Moreover the syntax of the clauses is a LISP-like syntax, for example the syntax of the clause:

$$P(*x, *y) \longleftarrow Q(*x, a) \wedge R(*x, *y)$$

is, in the PROLISP syntax:

$$((P *x *y)(Q *x\ a)(R *x *y))$$

However in this presentation we will conserve the usual PROLOG syntax.

4.3 Methodology

The query submitted by the user is considered by the INTERFACE as a set of goals to be solved.

It tries to solve these goals using the standard PROLOG strategy (depth first), and do the same unifications as the PROLOG interpreter. So, if some constants appear in the query, they are "propagated" in all the sub-goals, except in the case where a sub-goal is recursively defined.

The first time the interpreter finds a Base Predicate, instead of trying to solve this sub-goal, it stores it to generate a Relational query, and continues the resolution as this sub-goal would be solved.

If the next sub-goal is also a Base Predicate, it is inserted in the query wich is in generation, and the resolution is continued, else the generation of this query is terminated, the corresponding sub-goals are replaced in the PROLOG program by a new predicate which is used to represent the ground clauses associated to the query answer, and the resolution is continued.

In fact this general methodology is modified when a predicate definition contains:

- an evaluable predicate, or
- a negation, or
- a "cut", or
- a predicate defined by several clauses, or
- a recursively defined predicate.

The methodology will be made more precise in the next section through examples.

5 Query Generation

5.1 Example presentation

Let us consider a very simple example in the financial domain where the Base Predicates are (see also in the Annex):

$Participation(x, y, z)$: the company x owns z percent of the company y.

$Country(x, y)$: the country of the stock, or bond, x is y.

$Stock\text{-}Return(x, y)$: the return of the stock x is y.

$Bond\text{-}Return(x, y)$: the return of the bond x is y.

The other predicates are defined by PROLOG clauses, they are presented with each example.

5.2 Standard case

The standard case, the simplest one, is the case where the goals of the query refer to non-recursive virtual predicates, each predicate being defined only by one clause.

Most of the PROLOG-DBMS presented in the literature reduce the clustering of Base Predicates to this case.

Let us consider, for example, the query:

$$\longleftarrow Equiv\text{-}Stock(*x_1, *x_2)$$

and the program P_1:

$$Equiv\text{-}Stock(*x_1, *x_2) \longleftarrow Same\text{-}Return(*x_1, *x_2) \wedge Same\text{-}Country(*x_1, *x_2)$$

$$Same\text{-}Country(*x_1, *x_2) \longleftarrow Country(*x_1, *y) \wedge Country(*x_2, *y)$$

$$Same\text{-}Return(*x_1, *x_2) \longleftarrow Stock\text{-}Return(*x_1, *y) \wedge Stock\text{-}Return(*x_2, *y)$$

$$Country(*x, *y) \longleftarrow edb\text{-}Country(*x, *y)$$

$$Stock\text{-}Return(*x, *y) \longleftarrow edb\text{-}Stock\text{-}Return(*x, *y)$$

The result of the query generation step is the query:

$$Q_1(*x_1, *x_2) = \exists *y\ (Stock\text{-}Return(*x_1, *y) \wedge Stock\text{-}Return(*x_2, *y)) \wedge \exists *z\ (Country(*x_1, *z) \wedge Country(*x_2, *z))$$

and the transformed program P_1':

$$Equiv\text{-}Stock(*x_1, *x_2) \longleftarrow Q_1(*x_1, *x_2)$$

The rational behind the introduction of the existential quantifier "∃" is explained in sections 5.4 and 5.6 .

If the query evaluation by the DBMS returns the following facts:

$Q_1(a,c)$
$Q_1(b,d)$
$Q_1(a,b)$

They are inserted in P_1' leading to the final program P_1'':

$Equiv\text{-}Stock(*x_1, *x_2) \longleftarrow Q_1(*x_1, *x_2)$
$Q_1(a,c)$
$Q_1(b,d)$
$Q_1(a,b)$

The query generation process is detailed with a similar example, where the predicate arguments are not explicited for simplification.

Let us consider the next query and program:

$$\begin{array}{rcl} & \longleftarrow & Q \\ Q & \longleftarrow & A \wedge B \\ A & \longleftarrow & A_0 \wedge A_1 \\ B & \longleftarrow & B_0 \wedge B_1 \end{array}$$

where A_0, A_1, B_0, and B_1 are Base Predicates.
The Interface begins by trying to solve Q:

begin Q resolution

then it has to solve A:

begin A resolution

then it has to solve A_0; it finds that A_0 is a Base Predicate and starts the generation of a the query q_1:

$$q_1 = A_0$$

then it has to solve A_1; it finds that A_1 is a Base Predicate and inserts A_1 into the query q_1:

$$q_1 = A_0 \wedge A_1$$

then it finds the end of A resolution:

end A resolution

then it has to solve B:

begin B resolution

by a similar process it generates:

$q_1 = A_0 \wedge A_1 \wedge B_0$
and $q_1 = A_0 \wedge A_1 \wedge B_0 \wedge B_1$

end B resolution
end Q resolution

In the case where a variable appears in the body of a clause, but does not appear in the head, this variable is existentially quantified in the generated query.

Let us consider, for example, the clause:

$$A(x,y) \longleftarrow A_0(x,y,z) \wedge A_1(x,t)$$

In the generated query the variables z and t are quantified giving:

$$q_1 = \exists z,t\,(A_0(x,y,z) \wedge A_1(x,t))$$

There are two reasons to do this quantification: a performance reason, since this quantification limits the number of $< x, y >$ tuples in the answer, and a semantic reason when there are OR or NOT operators in the generated query. This last point is detailed latter.

If there are constants in the query, they are propagated by the unification process. For example if we have the sub-goal $A(a,b)$, the new sub-goals are $A_0(a,b,z)$ and $A_1(a,t)$, and the generated query is:

$$q_1 = \exists z,t\,(A_0(a,b,z) \wedge A_1(a,t))$$

5.3 Case of evaluable predicates

We consider now the same sort of query and program as in section 5.2, the only difference being that some predicates can be evaluable predicates.

Let us consider,for example, the query:

$$\longleftarrow Similar\text{-}Stock(*x_1, *x_2)$$

and the program P_2:

$Similar\text{-}Stock(*x_1, *x_2)$
$\quad \longleftarrow Similar\text{-}Return(*x_1, *x_2) \wedge$
$\quad\quad Same\text{-}Country(*x_1, *x_2)$

$Same\text{-}Country(*x_1, *x_2)$
$\quad \longleftarrow Country(*x_1, *y) \wedge Country(*x_2, *y)$

$Similar\text{-}Return(*x_1, *x_2)$

$$\longleftarrow Stock\text{-}Return(*x_1, *y_1) \wedge$$
$$Stock\text{-}Return(*x_2, *y_2) \wedge$$
$$eval\text{-}Similar(*y_1, *y_2)$$

$$Country(*x, *y) \longleftarrow edb\text{-}Country(*x, *y)$$

$$Stock\text{-}Return(*x, *y) \longleftarrow edb\text{-}Stock\text{-}Return(*x, *y)$$

where *eval-Similar* is an evaluable predicate which is satisfied iff the absolute difference between y_1 and y_2 is less than 2.

The result of the query generation step is the queries:

$$Q_2(*x_1, *y_1, *x_2, *y_2) = Stock\text{-}Return(*x_1, *y_1) \wedge Stock\text{-}Return(*x_2, *y_2)$$
$$Q_3(*x_1, *x_2) = \exists * y\ (Country(*x_1, *y) \wedge Country(*x_2, *y))$$

and the transformed program is:

$$Similar\text{-}Stock(*x_1, *x_2) \longleftarrow Q_3(*x_1, *y_1, *x_2, *y_2) \wedge eval\text{-}Similar(*y_1, y_2) \wedge Q_2(*x_1, *x_2)$$

The query generation can be detailed with this simplified example:

$$\begin{array}{ccl} & \longleftarrow & Q \\ Q & \longleftarrow & A \wedge B \\ A & \longleftarrow & A_0 \wedge A_1 \\ B & \longleftarrow & B_0 \wedge eval\text{-}F \wedge B_1 \end{array}$$

At the beginning we have, as in section 5.2, the steps:

begin Q resolution
  begin A resolution
    $q_1 = A_0$
    $q_1 = A_0 \wedge A_1$
  end A resolution
  begin B resolution
    $q_1 = A_0 \wedge A_1 \wedge B_0$

then the interpreter finds the evaluable predicate *eval-F*, stops the generation of q_1, and starts the generation of a new query for B_1:

    $q_2 = B_1$
  end B resolution
end A resolution

The method is to stop the construction of a query since an evaluable predicate is found. In some cases it could be possible to cluster larger queries by permuting the evaluable predicate from left to right in a clause body, but it is difficult to guarantee that this does not change the program semantics, because evaluable predicates can have side effects like input-output.

5.4 Case of negations

Now we consider PROLOG programs containing negations defined, as usual, by failure, by the clauses:

$$\neg(*x) \longleftarrow *x/fail$$
$$\neg(*x)$$

where $*x$ can be unified with a litteral.

Programs containing negations need a particular treatment since we want the negations to be evaluated by the DBMS. Let us consider, for example, the query:

$$\longleftarrow Equiv\text{-}Distributed\text{-}Stock(*x_1, *x_2)$$

and the program P_3:

$$Equiv\text{-}Distributed\text{-}Stock(*x_1, *x_2) \longleftarrow Same\text{-}Return(*x_1, *x_2) \wedge \neg Same\text{-}Country(*x_1, *x_2)$$

$$Same\text{-}Country(*x_1, *x_2) \longleftarrow Country(*x_1, *y) \wedge Country(*x_2, *y)$$

$$Same\text{-}Return(*x_1, *x_2) \longleftarrow Stock\text{-}Return(*x_1, *y) \wedge Stock\text{-}Return(*x_2, *y)$$

$$Country(*x, *y) \longleftarrow edb\text{-}Country(*x, *y)$$

$$Stock\text{-}Return(*x, *y) \longleftarrow edb\text{-}Stock\text{-}Return(*x, *y)$$

The result of the query generation step is:

$$Q_4(*x_1, *x_2) = (\exists * y\ (Stock\text{-}Return(*x_1, *y) \wedge Stock\text{-}Return(*x_2, *y))) \wedge (\neg \exists * z\ (Country(*x_1, *z) \wedge Country(*x_2, *z)))$$

and the transformed program P_3' is:

$$Equiv\text{-}Distributed\text{-}Stock(*x_1, *x_2) \longleftarrow Q_4(*x_1, *x_2)$$

This example shows why it is necessary to introduce existential quantifiers. Indeed if the variable z is not quantified in Q_4 it is impossible to translate the query into a Relational query language because the range of z is not explicited in the query, and the query would not be a Domain Independent formula.[Demolombe 82]

The process of query generation is detailed with the next example:

$$\begin{array}{ccl} & \longleftarrow & Q \\ Q & \longleftarrow & A \wedge \neg B \\ A & \longleftarrow & A_0 \wedge A_1 \\ B & \longleftarrow & B_0 \wedge B_1 \end{array}$$

At the beginning we have, as in section 5.2, the steps:

begin Q resolution
 begin A resolution
  $q_1 = A_0$
  $q_1 = A_0 \wedge A_1$
 end A resolution
 begin $\neg B$ resolution

If B is a non-recursive virtual predicate, and if all the variables appearing in B appear in the query q_1, then the construction of the query q_1 is continued giving:

  $q_1 = A_0 \wedge A_1 \wedge \neg(B_0$
  $q_1 = A_0 \wedge A_1 \wedge \neg(B_0 \wedge B_1)$
 end $\neg B$ resolution
end Q resolution

If one of the previous condition is not satisfied, the construction of the query q_1 is stopped, and a second query is generated:

  $q_2 = B_0$
  $q_2 = B_0 \wedge B_1$
 end $\neg B$ resolution
end Q resolution

So, depending on the conditions we have explicited, the transformed program is:

$Q \longleftarrow q_1$ with $q_1 = A_0 \wedge A_1 \wedge \neg(B_0 \wedge B_1)$

or:

$Q \longleftarrow q_1 \wedge \neg q_2$ with $q_1 = A_0 \wedge A_1$,
and $q_2 = B_0 \wedge B_1$

For instance, if we have the program:

$$Q(x,y) \longleftarrow A(x,y) \wedge \neg B(x,y)$$
$$A(x,y) \longleftarrow A_0(x,y) \wedge eval\text{-}F(x,y) \wedge A_1(x)$$
$$B(x,y) \longleftarrow B_0(x) \wedge B_1(x,y)$$

the query generation process is:

begin $Q(x,y)$ resolution
 begin $A(x,y)$ resolution
  $q_1 = A_0(x,y)$
  $q_2 = A_1(x)$
 end $A(x,y)$ resolution
 begin $\neg B(x,y)$

since the variable y does not appear in q_2, the q_2 generation is stopped and a new query generation is started:

  $q_3 = B_0(x)$
  $q_3 = B_0(x) \wedge B_1(x,y)$
 end $\neg B(x,y)$ resolution
end $Q(x,y)$ resolution

The transformed program, in this case is:

$$Q(x,y) \longleftarrow q_1(x,y) \wedge eval\text{-}F(x,y) \wedge q_2(x) \wedge \neg q_3(x,y)$$

5.5 Case of cut

The predicates having a cut in their definitions need also a special treatment as it will be shown with the next example. Let us assume that we want to obtain italian stocks x_1, having at least one french owner, and we are interested to know the nationality y_2 of each owner x_2. In that case we will have the query:

$$\longleftarrow Italian\text{-}French\text{-}Stock(*x_1, *x_2, *y_2)$$

and the program P_4:

$$Italian\text{-}French\text{-}Stock(*x_1, *x_2, *y_2)$$
$$\longleftarrow Country(*x_1, Italian) \wedge French\text{-}Owner(*x_1, *x_2) \wedge Country(*x_2, *y_2)$$

$$French\text{-}Owner(*x_1, *x_2)$$
$$\longleftarrow Participation(*x_3, *x_1, *z) \wedge Country(*x_3, French) \wedge / \wedge Participation(*x_2, *x_1, *t)$$

$$Participation(*x, *y, *t)$$
$$\longleftarrow edb\text{-}Participation(*x, *y, *t)$$

$$Country(*x, *y) \longleftarrow edb\text{-}Country(*x, *y)$$

where "/" denotes the cut.

The semantics of the cut rule, is to select only one solution for the goals which are in the clause containing the /, and which are evaluated before the /. In the example the cut rule allows to get only one tuple $< x_3, x_1, z >$ satisfying:

$$Participation(*x_3, *x_1, *z) \wedge Country(*x_3, French),$$

for each x_1 satisfying:

$$Country(*x_1, Italian).$$

The generated queries are:

$$Q_5(*x_1) = Country(*x_1, Italian)$$
$$Q_6(*x_1) = \exists *x_3, *z\ (Participation(*x_3, *x_1, *z) \wedge Country(*x_3, French))$$
$$Q_7(*x_2, *x_1) = \exists *t\ (Participation(*x_2, *x_1, *t))$$
$$Q_8(*x_2, *y_2) = Country(*x_2, *y_2)$$

and the transformed program is P_4':

$$
\begin{aligned}
&\mathit{Italian\text{-}French\text{-}Stock}(*x_1, *x_2, *y_2)\\
&\quad\longleftarrow Q_5(*x_1)\wedge\\
&\qquad \mathit{French\text{-}Owner}(*x_1, *x_2)\wedge\\
&\qquad Q_8(*x_2, *y_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{French\text{-}Owner}(*x_1, *x_2)\\
&\quad\longleftarrow Q_6(*x_1) \wedge / \wedge Q_7(*x_2, *x_1)
\end{aligned}
$$

It can be noticed that the queries Q_5 and Q_6 cannot be clustered. Indeed, if they are clustered, like in the following program:

$$
\begin{aligned}
&\mathit{Italian\text{-}French\text{-}Stock}(*x_1, *x_2, *y_2)\\
&\quad\longleftarrow Q_5(*x_1) \wedge Q_6(*x_1) \wedge /\wedge\\
&\qquad Q_7(*x_2, *x_1) \wedge Q_8(*x_2, *y_2)
\end{aligned}
$$

we obtain only one solution for $Q_5(*x_1) \wedge Q_6(*x_1)$, that is only one italian stock having a french owner.

The query generation process can be detailed with the following example:

$$
\begin{array}{lcl}
 & \longleftarrow & Q\\
Q & \longleftarrow & A \wedge B \wedge C\\
A & \longleftarrow & A_0 \wedge A_1\\
B & \longleftarrow & B_0 \wedge B_1 \wedge / \wedge B_2 \wedge B_3\\
C & \longleftarrow & C_0 \wedge C_1
\end{array}
$$

At the beginning we have, as in section 5.2, the steps:

begin Q resolution
  begin A resolution
    $q_1 = A_0$
    $q_1 = A_0 \wedge A_1$
  end A resolution
begin B resolution

at this stage, since the B body contains a cut, we need to stop the q_1 construction and to start the construction of a new query q_2:

$q_2 = B_0$
$q_2 = B_0 \wedge B_1$

when the cut is found the q_2 construction is stopped and a new query construction is started:

$q_3 = B_2$
$q_3 = B_2 \wedge B_3$

when all the goals in the clause containing the cut are solved the q_3 construction is stopped, and a new query construction is started:

  end B resolution
  begin C resolution
    $q_4 = C_0$
    $q_4 = C_0 \wedge C_1$
  end C resolution
end Q resolution

The tranformed program in this case is:

$Q \longleftarrow q_1 \wedge B \wedge q_4$
$B \longleftarrow q_2 \wedge / \wedge q_3$

5.6 Case of disjunctive definitions

We consider now PROLOG program containing predicates defined by several clauses.

Let us consider, for example the query:

$$\longleftarrow \mathit{Equiv\text{-}Asset}(*x_1, *x_2)$$

and the program P_5:

$$
\begin{aligned}
&\mathit{Equiv\text{-}Asset}(*x_1, *x_2)\\
&\quad\longleftarrow \mathit{Same\text{-}Country}(*x_1, *x_2)\wedge\\
&\qquad \mathit{Same\text{-}Asset\text{-}Return}(*x_1, *x_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Same\text{-}Country}(*x_1, *x_2)\\
&\quad\longleftarrow \mathit{Country}(*x_1, *y) \wedge \mathit{Country}(*x_2, *y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Same\text{-}Asset\text{-}Return}(*x_1, *x_2)\\
&\quad\longleftarrow \mathit{Asset\text{-}Return}(*x_1, *y)\wedge\\
&\qquad \mathit{Asset\text{-}Return}(*x_2, *y)
\end{aligned}
$$

$$\mathit{Asset\text{-}Return}(*x, *y) \longleftarrow \mathit{Stock\text{-}Return}(*x, *y)$$

$$\mathit{Asset\text{-}Return}(*x, *y) \longleftarrow \mathit{Bond\text{-}Return}(*x, *y)$$

$$\mathit{Country}(*x, *y) \longleftarrow \mathit{edb\text{-}Country}(*x, *y)$$

$$\mathit{Stock\text{-}Return}(*x, *y) \longleftarrow \mathit{edb\text{-}Stock\text{-}Return}(*x, *y)$$

$$\mathit{Bond\text{-}Return}(*x, *y) \longleftarrow \mathit{edb\text{-}Bond\text{-}Return}(*x, *y)$$

In this case the *Asset-Return* predicate is defined by two clauses, and the queries corresponding to each clauses are connected by an OR operator, denoted by "$\vee$". The generated query is:

$$
\begin{aligned}
&Q_9(*x_1,*x_2) =\\
&\quad (\exists * y\ (\mathit{Country}(*x_1, *y)\wedge\\
&\qquad \mathit{Country}(*x_2, *y)))\wedge\\
&\quad (\exists * z\ ((\mathit{Stock\text{-}Return}(*x_1, *z)\vee\\
&\qquad \mathit{Bond\text{-}Return}(*x_1, *z))\wedge\\
&\qquad (\mathit{Stock\text{-}Return}(*x_2, *z)\vee\\
&\qquad \mathit{Bond\text{-}Return}(*x_2, *z))))
\end{aligned}
$$

and the generated program is P'_5:

$$Equiv\text{-}Asset(*x_1, *x_2) \longleftarrow Q_9(*x_1, *x_2)$$

The process of query generation is detailed with the next example:

$$\begin{array}{rcl} & \longleftarrow & Q \\ Q & \longleftarrow & A \wedge B \wedge C \\ A & \longleftarrow & A_0 \wedge A_1 \\ B & \longleftarrow & B_0 \\ B & \longleftarrow & B_1 \\ C & \longleftarrow & C_0 \\ C & \longleftarrow & C_1 \end{array}$$

At the beginning we have, as in section 5.2, the steps:

begin Q resolution
  begin A resolution
    $q_1 = A_0$
    $q_1 = A_0 \wedge A_1$
  end A resolution
  begin B resolution

At this stage the interface finds that there are several possibilities to solve B, each one corresponding to a clause whose consequence is B; if B is a non-recursive virtual predicate, and if all the clauses defining B have the "same" consequences (changing just variable names), then the standard PROLOG search strategy is changed; the resolution of C is started only when **all** the possibilities to solve B have been explored, and all the sub-queries corresponding to each possibility are connected by an OR operator; so we have:

begin 1st B resolution
  $q_1 = A_0 \wedge A_1 \wedge (B_0$
end 1st B resolution
begin 2nd B resolution
  $q_1 = A_0 \wedge A_1 \wedge (B_0 \vee B_1$
end 2nd resolution
  $q_1 = A_0 \wedge A_1 \wedge (B_0 \vee B_1)$
end B resolution
begin C resolution

with the same process we have for C resolution:

  $q_1 = A_0 \wedge A_1 \wedge (B_0 \vee B_1) \wedge (C_0$
  $q_1 = A_0 \wedge A_1 \wedge (B_0 \vee B_1) \wedge (C_0 \vee C_1)$
end C resolution
end Q resolution

The reason why the PROLOG strategy is changed in the case of disjunctive queries generation is that the standard strategy would lead to generate several queries whose evaluation is much more expensive. For instance, with the previous example we would obtain the four queries:

$q_1 = A_0 \wedge A_1 \wedge B_0 \wedge C_0$
$q_2 = A_0 \wedge A_1 \wedge B_0 \wedge C_1$
$q_3 = A_0 \wedge A_1 \wedge B_1 \wedge C_0$
$q_4 = A_0 \wedge A_1 \wedge B_1 \wedge C_1$

and that would lead to repeat four times the sub-query $A_0 \wedge A_1$.

In the case of disjunctive query generation we need again to introduce existential quantifiers. Indeed, if we have the following B definition:

$B(x) \longleftarrow B_0(x)$
$B(x) \longleftarrow B_1(x, y)$

the sub-query: $B_0(x) \vee B_1(x, y)$, cannot be translated into a Relational query language, because it is not Domain Independent. While: $B_0(x) \vee (\exists y\, B_1(x, y))$, can be translated.

5.7 Case of recursive definitions

We consider now PROLOG program containing predicates which are recursively defined.

Let us assume that we want, for example, to obtain the italian companies which, directly or not directly, owns some percentage of a french company. In that case we will have the query:

$$\longleftarrow Italian\text{-}Owners\text{-}of\text{-}French(*x_1, *x_2)$$

and the program P_6:

$Italian\text{-}Owners\text{-}of\text{-}French(*x_1, *x_2)$
  $\longleftarrow Country(*x_1, Italian) \wedge$
    $Control(*x_1, *x_2) \wedge$
    $Country(*x_2, French)$

$Control(*x_1, *x_2)$
  $\longleftarrow Direct\text{-}Control(*x_1, *x_2)$

$Control(*x_1, *x_2)$
  $\longleftarrow Direct\text{-}Control(*x_1, *x_3) \wedge$
    $Control(*x_3, *x_2)$

$Direct\text{-}Control(*x_1, *x_2)$
  $\longleftarrow Participation(*x_1, *x_2, *y)$

$Country(*x_1, *y) \longleftarrow edb\text{-}Country(*x_1, *y)$

$Participation(*x_1, *x_2, *y)$
  $\longleftarrow edb\text{-}Participation(*x_1, *x_2, *y)$

In that case the sub-goals: $Country(*x_1, Italian)$, $Participation(*x_1, *x_2, *y)$ and $Country(*x_2, French)$, cannot be clustered in the same

query. Furthermore the x_1 instantiations obtained from $Country(*x_1, Italian)$, cannot be propagated into the sub-goal $Participation(*x_1, *x_2, *y)$, since an italian owners can owns, for instance, a belgium company, which owns a french company; therefore the x_1 instantiations, in *Participation*, must not be restricted to italian companies. So the generated queries are:

$$Q_{10}(*x_1) = Country(*x_1, Italian)$$
$$Q_{11}(*x_1, *x_2) = \exists *y\ Participation(*x_1, *x_2, *y)$$
$$Q_{12}(*x_2) = Country(*x_2, French)$$

and the transformed program is P_6':

$$Italian\text{-}Owners\text{-}of\text{-}French(*x_1, *x_2) \longleftarrow Q_{10}(*x_1) \wedge Control(*x_1, *x_2) \wedge Q_{12}(*x_2)$$

$$Control(*x_1, *x_2) \longleftarrow Q_{11}(*x_1, *x_2)$$

$$Control(*x_1, *x_2) \longleftarrow Q_{11}(*x_1, *x_3) \wedge Control(*x_3, *x_2)$$

In general the generation process of a given query is stopped when the interface finds a sub-goal represented with a recursive predicate, as it is for an evaluable predicate, and the sub-goal is considered as the initial goal. To avoid loops in the generation process, the recursive sub-goals which have been solved are stored.

That will be made more precise with the next example:

$$
\begin{array}{lll}
 & \longleftarrow & Q \\
Q & \longleftarrow & A \wedge B \wedge C \\
A & \longleftarrow & A_0 \wedge A_1 \\
B & \longleftarrow & B_0 \\
B & \longleftarrow & B_0 \wedge D \\
C & \longleftarrow & C_0 \wedge C_1 \\
D & \longleftarrow & D_0 \wedge D_1 \\
D & \longleftarrow & D_0 \wedge B
\end{array}
$$

At the beginning we have, like in the previous cases:

begin Q resolution
  begin A resolution
    $q_1 = A_0 \wedge A_1$
  end A resolution
  begin B resolution

At this stage the interface finds that B is recusively defined, and it stores that B resolution is going on.

begin 1st B resolution
  $q_2 = B_0$
end 1st B resolution
begin 2nd B resolution
  begin B_0 resolution
    $q_3 = B_0$
  end B_0 resolution
  begin D resolution

Like for B, it stores that D is recursively defined and that D resolution is going on.

begin 1st D resolution
  $q_4 = D_0 \wedge D_1$
end 1st D resolution
begin 2nd D resolution
  begin D_0 resolution
    $q_5 = D_0$
  end D_0 resolution
  begin B resolution

At this stage the interface finds B is recursively defined, and that its resolution has yet been initiated; in that case, since it is at the end of a loop no resolution is started for B.

      end B resolution
    end 2nd D resolution
  end D resolution
end 2nd B resolution
end B resolution
begin C resolution
  $q_6 = C_0 \wedge C_1$
end C resolution
end Q resolution

It can be noticed that, in this particular example, our method is not optimal since $q_3 = B_0$ and $q_4 = D_0 \wedge D_1$ could be clustered in a unique larger query. However it is not possible to cluster the subqueries appearing in a loop in the general case; for example this is impossible if we have a mutual recursion with two predicates, like:

$$B \longleftarrow B_0 \wedge D \wedge B$$
$$D \longleftarrow D_0 \wedge D_1 \wedge B \wedge D$$

To avoid to have to distinguish too much complex cases, we finally decided to drop this kind of optimisation.

6 Syntax of the generated queries

We adopted a syntax for generated queries which exactly corresponds to the generation process. That is there

is no need to change the form of the queries after their generation. Moreover this syntax allows a straightforward transformation into any Relational query language because each sub-formula of a formula in this form is Domain Independent.

The consequence is that the translation of these formulas into a Relational Calculus-like language, or into a Relational Algebra-like language is a pure syntactical work.

The grammar of the language is defined by the rules:

$$\begin{array}{rcl} < formula > & \longrightarrow & (< formula' >^*) \\ < formula' > & \longrightarrow & (OR < formula >^*) | \\ & & (EXIST(< var >^*) < formula >) | \\ & & (NOT < formula >) | \\ & & < litteral > \\ < litteral > & \longrightarrow & (< predicate >< argument >^*) \end{array}$$

where the "*" denotes n repetitions of some item $(n > 0)$.

This syntax (a bit unusual) is very close to PROLISP syntax style, its semantics can be easily understood with a small example:

$$((L_1)(OR((L_2)(L_3)))(NOT((L_4))))$$

is usually represented with:

$$L_1 \wedge (L_2 \vee L_3) \wedge \neg L_4$$

7 Comparison of the dynamic and static coupling

We have presented an Interface based on the static coupling approach. It is interesting to compare its advantages and drawbacks with regard to the dynamic coupling approach. In the following we do this comparison taking into account two different criteria: the number of interactions between the PROLOG interpreter and the Relational DBMS, and the size of the answers obtained from the DBMS which must be stored in core memory.

The differences are significant only in the case of the queries whose evaluation involves predicates which are recusively defined. Let us consider for example the program P_7, which contains a recursive definition of the Control predicate:

$$\longleftarrow Control(a, *x_2)$$

$$Control(*x_1, *x_2) \longleftarrow Direct\text{-}Control(*x_1, *x_2)$$

$$\begin{array}{l} Control(*x_1, *x_2) \\ \quad \longleftarrow Direct\text{-}Control(*x_1, *x_3) \wedge \\ \quad\quad Control(*x_3, *x_2) \end{array}$$

$$\begin{array}{l} Direct\text{-}Control(*x_1, *x_2) \\ \quad \longleftarrow Participation(*x_1, *x_2, *y) \end{array}$$

$$\begin{array}{l} Participation(*x_1, *x_2, *y) \\ \quad \longleftarrow edb\text{-}Participation(*x_1, *x_2, *y) \end{array}$$

According to the static approach, the query Q_{13} is generated:

$$Q_{13}(*x_1, *x_2) = \exists *y \; Participation(*x_1, *x_2, *y)$$

And the transformed program P_7' is, for instance:

$$\longleftarrow Control(a, *x_2)$$

$$Control(*x_1, *x_2) \longleftarrow Q_{13}(*x_1, *x_2)$$

$$\begin{array}{l} Control(*x_1, *x_2) \longleftarrow Q_{13}(*x_1, *x_3) \wedge \\ \quad\quad Control(*x_3, *x_2) \end{array}$$

$$\begin{array}{l} Q_{13}(a, b) \\ Q_{13}(b, c) \\ Q_{13}(c, d) \end{array}$$

In this case there is only one call to the DBMS, and the size of the answer is the projection of all the *Participation* relation on the two first arguments. The drawback is that the answer contains much more information than what is needed to compute the answer, and can be quite large.

If we adopt the dynamic approach the transformed program contains calls to the DBMS which are evaluated during the PROLOG program execution.

We get the transformed program P_7'':

$$\longleftarrow Control(a, *x_2)$$

$$Control(*x_1, *x_2) \longleftarrow eval\text{-}Q_{13}(*x_1, *x_2)$$

$$\begin{array}{l} Control(*x_1, *x_2) \longleftarrow eval\text{-}Q_{13}(*x_1, *x_3) \wedge \\ \quad\quad Control(*x_3, *x_2) \end{array}$$

where $eval\text{-}Q_{13}(*x_1, *x_3)$ is an evaluable predicate containing a call to the DBMS, like:

$$\begin{array}{l} eval\text{-}Q_{13}(*x_1, *x_3) = \\ \quad Call : \exists *y \; Participation(*x_1, *x_3, *y) \end{array}$$

That leads, at the run time, to a first call corresponding to:

$$eval\text{-}Q_{13}(a, *x_3)$$

The answer, in this example, is just $*x_3 = b$.

Then we have a second call corresponding to:

$$eval\text{-}Q_{13}(b, *x_3)$$

And the answer is $*x_3 = c$. And so on, for each recursive call to the *Control* predicate.

In this approach the number of call to the DBMS can be quite important, leading to an important overhead, because for each call the DBMS has to analyse the query, to optimise it, and so on. But the advantage is that the size of each answer is much more smaller. However in some cases the dynamic approach can find pathological cases where we have only drawbacks, because the size of the answers is as large as for the static approach, and where the same query is evaluated for each recusion step.

The next program P_8 is such an example of pathological cases:

$$\longleftarrow Control(*x_1, d)$$

$$Control(*x_1, *x_2) \longleftarrow Direct\text{-}Control(*x_1, *x_2)$$

$$Control(*x_1, *x_2) \longleftarrow Direct\text{-}Control(*x_1, *x_3) \wedge Control(*x_3, *x_2)$$

$$Direct\text{-}Control(*x_1, *x_2) \longleftarrow Participation(*x_1, *x_2, *y)$$

$$Participation(*x_1, *x_2, *y) \longleftarrow edb\text{-}Participation(*x_1, *x_2, *y)$$

Indeed the transformed program, according to the dynamic approach, is again P_7''. And for each recursion step we have a call of the form:

$$eval\text{-}Q_{13}(*x_1, *x_3)$$

where no variable is instantiated. This means that all the *Participation* relation projection is recomputed for each recursion. These examples show that we cannot conclude that in all cases one approach is better than the other one.

8 Conclusion

We have presented the implementation of a PROLOG-DBMS interface which do not need any modification of the PROLOG interpreter, or of the DBMS evaluator. This interface, even if it is not optimal , allows to cluster many calls to the DBMS since it can generate queries containing: AND, OR, or NOT operators, and existential quantifiers. Of course, though the examples in section 5 concern only one of these operators, the method works for more complex programs containing any combinations of them.

The only restriction on the PROLOG programs is to not contain *Assert*, or *Retract* operations.

A byproduct of the interface is that it could be used, with few changes, for a dynamic coupling approach. For example after the generation of the P_6' program (see section 5.6), instead of evaluating Q_{10}, Q_{11} and Q_{12}, and integrating corresponding answers in P_6', it would be easy to consider Q_{10}, Q_{11} and Q_{12} as evaluable predicates, associated to LISP functions, which are dynamically called during the PROLOG interpreter running.

References

[Bocca 86] Bocca. J. *EDUCE: A marriage of convenience: Prolog and a Relational DBMS.* Symposium on Logic Programming. 1986.

[Ceri 86] Ceri. S., Gottlob. G., Wiederhold. G., *Interfacing Relational Databases and Prolog Efficiently.* Proceedings From the First International Conference Expert Database Systems. Editor L. Kerschberg. 1987.

[Demolombe 82] Demolombe. R. *Syntactical characterization of a sub-set of Domain Independent formulas.* Int. Report ONERA-CERT. 1982.

[Denoel 85] Denoel. E, Roelants. D, Vauclair. M. *Coupling PROLOG with INGRES.* Int. Report Philips Research Laboratory, Brussels. 1985.

[Jarke 84] Jarke. M, Clifford. J, Vassiliou. Y. *An optimizing front-end to a Relational query system.* Proc. ACM SIGMOD. 1984.

[Kunifuji 82] Kunifuji.S, Yokota.H. *Prolog and Relational Databases for fifth generation computer systems.* Proc. Workshop on Logical Bases for Data Bases. 1982.

[Orci 85] Orci.I, Sahlin.D. *Two-mode evaluation for dealing with implicit interactions between Logic programs and Relational Data Bases.* Int. Report SYSLAB. 1985.

[Vanden 84] Vanden Bossche-Marquette. M, Venken. R. *LOKI: a Logic oriented approach to knowledge bases supporting natural user interaction.* First ESPRIT technical week. 1984.

[Venken84] Venken.R. *A PROLOG interpreter for partial evaluation and its application to source to source transformation and query optimisation.* Proc. AICA. 1984.

Annex

```
(defdb financial-db

; Negation
; ********
((NOT *x) *x (slash) (fail))
((NOT *x))

; Asset-Return
; ************
; x: asset    y:return
((Asset-Return *x *y)  (Stock-Return *x *y))

((Asset-Return *x *y)  (Bond-Return *x *y))

; Same-Return
; ***********
; x1: stock   x2: stock
((Same-Return *x1 *x2)
   (Stock-Return *x1 *y)
   (Stock-Return *x2 *y))

; Same-Asset-Return
; *****************
; x1: asset   x2:asset
((Same-Asset-Return *x1 *x2)
   (Asset-Return *x1 *y)
   (Asset-Return *x2 *y))

; Similar-Asset-Return
; ********************
; x1: asset   x2: asset
((Similar-Asset-Return *x1 *x2)
   (Asset-Return *x1 *y1)
   (Asset-Return *x2 *y2)
   (eval (Sim '*y1 '*y2) t))

; Similar-Stock-Return
; ********************
; x1: stock   x2: stock
((Similar-Stock-Return *x1 *x2)
   (Stock-Return *x1 *y1)
   (Stock-Return *x2 *y2)
   (eval (Sim '*y1 '*y2) t))

; Same-Country
; ************
; x1: asset   x2: asset
((Same-Country *x1 *x2)
   (Country *x1 *y) (Country *x2 *y))

; Equiv-Stock
; ***********
; x1: stock   x2: stock
((Equiv-Stock *x1 *x2)
    (Same-Return *x1 *x2)
    (Same-Country *x1 *x2))

; Equiv-Asset
; ***********
; x1: asset   x2: asset
((Equiv-Asset *x1 *x2)
   (Same-Asset-Return *x1 *x2)
   (Same-Country *x1 *x2))

; Similar-Stock
; *************
; x1: stock    x2: stock
((Similar-Stock *x1 *x2)
    (Similar-Stock-Return *x1 *x2)
    (Same-Country *x1 *x2))

; Similar-Asset
; *************
; x1: asset    x2: asset
((Similar-Asset *x1 *x2)
    (Similar-Asset-Return *x1 *x2)
    (Same-Country *x1 *x2))

; Equiv-Distributed-Stock
; ***********************
; x1: stock    x2: stock
((Equiv-Distributed-Stock *x1 *x2)
   (Same-Return *x1 *x2)
   (NOT (Same-Country *x1 *x2)))

; Italian-French-Stock
; ********************
; x1: italian stock having a french owner
; x2: owner of x1     y2: country of x2
((Italian-French-Stock *x1 *x2 *y2)
   (Country *x1 Italian)
   (French-Owner *x1 *x2)
   (Country *x2 *y2))

; French-Owner
; ************
; x1: stock having a french owner  x2: owner of x1
((French-Owner *x1 *x2)
    (Participation *x3 *x1 *z)
    (Country *x3 French)
    (slash)
    (Participation *x2 *x1 *t))

; Direct-Control
; **************
;  x1: owner   x2: owned
((Direct-Control *x1 *x2)
    (Participation *x1 *x2 *y))

; Control
; *******
; x1: owner  x2: owned
```

```
((Control *x1 *x2)   (Direct-Control *x1 *x2))

((Control *x1 *x2)
    (Direct-Control *x1 *x3) (Control *x3 *x2))

; Italian-Owners-of-French
; ************************
; x1: italian owner of x1  x2: french stock
((Italian-Owners-of-French *x1 *x2)
  (Country *x1 Italian)
  (Control *x1 *x2)
  (Country *x2 French))

; Diff-Country
; ************
; x1: asset    x2: asset
((Diff-Country *x1 *x2)
    (Asset-Return *x1 *y1) (eval (> '*y1 '5) t)
    (Asset-Return *x2 *y2) (eval (> '*y2 '5) t)
    (NOT (Same-Country *x1 *x2)) )

; Good-French-and-Similar
; ***********************
; x1: french asset   y1: return of x1
; x2: asset similar to x1    y2: return of x2
((Good-French-and-Similar *x1 *y1 *x2 *y2)
    (Asset-Return *x1 *y1) (eval (> '*y1 '8) t)
    (Country *x1 French)
    (slash)
    (Similar-Asset *x1 *x2)
    (Asset-Return *x2 *y2))

; Set-of-Owners-by-Country
; ************************
;  x: asset  y: country of x
;  l: list of x's owners in the country y
((Set-of-Owners-by-Country *x *y *l)
    (Direct-Control *u *x) (Country *u *y)
    (Group-by *x *y nil *l))

; Group-by
; ********
; x: stock  y: country of some x's owner  l1: working list
; l: list of x's owners in the country y
((Group-by *x *y *l1 *l)
    (Direct-Control *u *x) (Country *u *y)
    (NOT ( eval (Belongs '*u '*l1) t))
    (slash)
    (Group-by *x *y (*u . *l1) *l) )

((Group-by *x *y *l *l))

; List-of-Participation
; *********************
; x: stock  l:list of x's owners  p: list of participations
((List-of-Participation *x nil nil) (slash))

((List-of-Participation *x (*l1 . *l) (*p1 . *p))
    (Participation *l1 *x *p1)
    (List-of-Participation *x *l *p))

; Control-Nationality
; *******************
; x: stock   y: nationality of owners having the majority
((Control-Nationality  *x *y)
    (Set-of-Owners-by-Country *x *y *l)
    (List-of-Participation *x *l *p)
    (eval (Sum '*p) *s)
    (eval (> '*s '50) t) )
; Base predicates definitions
; ***************************

((Participation *x *y *z)
    (edb (Participation *x *y *z)))

((Country *x *y) (edb (Country *x *y)))

((Stock-Return *x *y) (edb (Stock-Return *x *y)))

((Bond-Return *x *y) (edb (Bond-Return *x *y)))

)
```

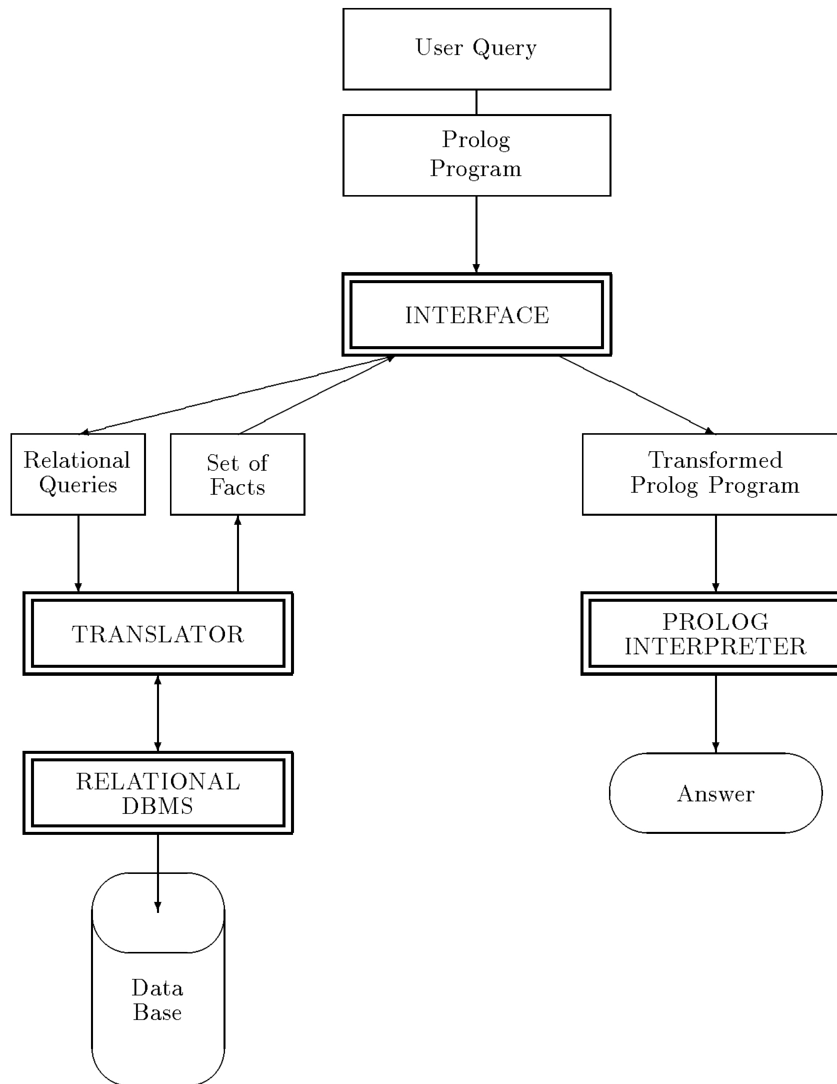

Figure 1: Interface Architecture