BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

Interface between Prolog and a General Database Server

by
Jose COTTA *
Raf VENKEN *

Internal Report
BIM-prolog IR8

November 1984

* BIM
Kwikstraat 4
B-3Ø78 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 2ØØA
B-3Ø3Ø Heverlee Belgium
tel. +32 16 2Ø Ø6 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

INTERFACE BETWEEN PROLOG AND A GENERAL DATABASE SERVER

by

Jose Cotta
Raf Venken
Belgian Institute of Management
Kwikstraat 4
3078 Everberg
Belgium

Contents

1. Introduction

2. Syntax of database questions and answers

    2.1 Request
    2.2 Reply

3. Description of the several database questions

    3.1 Opendb
    3.2 Closedb
    3.3 Retrieve
    3.4 Createrelation
    3.5 Insert
    3.6 Delete
    3.7 Deleterelation
    3.8 FurthertupleR
    3.9 FurthertupleD

4. Bibliography

1. Introduction

In this report we specify the interface between the **Prolog** system and a general **Relational Database Server**.

As it is stated in <Ven84> there are three different levels of interface, namely an higher level where the database system has to answer to joins of database calls, a middle level where it answers to individual relation calls, and a lower level where Prolog accesses the database tuple by tuple by means of "seek", "getnext" and similar procedures.

As far as this report is concerned we shall only specify the middle level of interface. The higher and the lower levels will soon have their own specification reports.

This middle level interface is based on the following principles:

1) There exists a **pipe-line** communication channel between Prolog and the Database Server, ie between the processes running Prolog and the Database Server:

```
 --------------                          ---------------
|             |   database questions   |               |
|             |   --------------->     |               |
|             |      (requests)        |               |
|             |                        |               |
|   Prolog    |   database answers     |   DB Server   |
|             |   <---------------     |               |
|             |       (replies)        |               |
|             |                        |               |
|             |                        |               |
 --------------                          ---------------
```

The Database Server will manage the database in a completely transparent way from the Prolog point of vue.

2) To the Prolog system, it only matters that it sends a **database question** through the pipe-line and that it gets back a **database answer** through the same pipe-line.

There are nine different database questions: opendb, closedb, retrieve, createrelation, insert, delete, deleterelation, furthertupleR and furthertupleD that will be described in this report.

3) The filosophy of the interface is based on an uniform type of communication through the pipe-line, ie, Prolog sends a

database question accordingly to a well defined syntax, we will call it the **request** from now on, and receives a database answer also accordingly to a well defined syntax, we will call it **reply** from now on.

Therefore, the aim of this paper is, on one hand, to specify the syntax of the request and reply and, on the other hand, to describe the actions that must be performed for each of the database questions.

2. Syntax of database questions and answers

2.1 Request

All the database questions will be sent by the Prolog system accordingly to the following syntax, stated in BNF form, where the terminals are bold-printed:

<request> ::= "<operation name> **[** [<arguments>] **]** <nl>

<operation name> ::= **opendb** |
closedb |
retrieve |
createrelation |
insert |
delete |
deleterelation |
furthertupleR |
furthertupleD

<arguments> ::= <argument> [**,** <arguments>]

<argument> ::= <predicate definition> |
<identifier> |
integer

<predicate definition> ::= <identifier>
[<predicate arguments> **]**

<predicate arguments> ::= <void variable> |
<variable> |
<string> |
integer

<variable> ::= <identifier>

<void variable> ::= **"_**

<identifier> ::= **"string of char**

<string> ::= **'string of char'**

<nl> ::= **<CR> <LF>**

This syntax suggests the following comments:

1) All the identifiers must be preceeded by a " with the exceptions of strings of characters that must be single quoted and integers that don't need any kind of quotation.

2) For example:

```
"retrieve["address['Raf',"_]]<CR> <LF>
```

is a valid request and represents the Prolog query:

```
?-address(Raf,_).
```

which can be read as: 'Has Raf an address?'.

In section 3 we will give examples of the requests generated by the several database questions.

2.2 Reply

All the database answers will be sent by the Database Server to the Prolog system, through the pipe-line, accordingly to the following syntax, stated also in BNF form with the terminals bold-printed:

<reply> ::= **"** <operation name> **[** <tuple> **,**
<pointer list> **,**
<return code> **]** <nl>

<operation name> ::= **opendb** |
closedb |
retrieve |
createrelation |
insert |
delete |
deleterelation |
furthertupleR |
furthertupleD

<tuple> ::= **[** [<arguments>] **]**

<pointer list> ::= **[** [<pointers>] **]**

<return code> ::= **0** | **negative integer**

<nl> ::= **<CR> <LF>**

<arguments> ::= <argument> [**,** <arguments>]

<pointers> ::= <pointer> [**,** <pointers>]

<argument> ::= **'string of char'** | **integer**

<pointer> ::= **"string of char**

This syntax suggests the following comments:

1) <return code> can be 0 meaning successful operation or an error code represented by a negative integer in case of unsuccessful operation.

2) Note that the <tuple> may be empty and the operation successful if there is no need of passing information back for the Prolog system. In the same way the <pointer list> may also be empty and the operation successful if there are no further tuples admissible as solutions to that particular query. We will go back to this subject when describing the different database questions in detail.

In the next section we will give examples of the replies expected by the Prolog system as answers to the several database questions.

3. Description of the several database questions

Each of the database questions generates a request with different arguments. The replies to these requests have also different arguments, accordingly to the information that is sent back to the Prolog system. Therefore, in this section we will describe in detail all the database questions and the operations that must be executed by the database management system for each of them. We will also give a lot of examples to illustrate the description.

3.1 Opendb

This question opens a specified database for further use. Therefore, it must be called always in the beginning of the session, either if we want to create a new database or to use an already existing one.

Its actual **request** has the following format:

```
"opendb['<name>',<options>]<CR><LF>
```

where <name> is a character string specifying the name of the database to be opened and <options> is a list of options for that operation.

Its actual **reply** has the following format:

```
"opendb[[],[],<return>]<CR><LF>
```

where <return> is 0 if the operation is sucessful and a negative integer otherwise. The first two arguments are [] because there is no <tuple> and no <pointer list> to send to Prolog.

Let's see how this operation is called through some examples.

Example 3.1.1: (New database)

Suppose that we want to create a new database called "mydata". The request for this operation would be:

```
"opendb['mydata',[...]]<CR<LF>
```

Assuming that the operation was sucessful the reply sent to Prolog would be:

```
"opendb[[],[],0]<CR><LF>
```

Example 3.1.2: (Already existing database)

Let's now suppose that the database "mydata" already existed. The request to call such an opening would be:

```
"opendb['mydata',[...]]<CR><LF>
```

Assuming that the operation was sucessful the reply sent to Prolog would be:

```
"opendb[[],[],0]<CR><LF>
```

Notes about the opendb operation:

1) This operation must preceed any of the other operations.

2) When the operation is sucessful the database is opened accordingly to the list of options.

3) The operation is not sucessful, and therefore the database is not opened, when it is already opened or when the list of options can not be satisfied. In this case the error code will be transmited in <return>.

3.2 Closedb

This question closes the currently opened database. Therefore, it must be the last call of the session. If a "closedb" is called before the end of the session, the other existing questions can't be executed, with the exception of "opendb".

Example 3.2.1:

If the Prolog system wants to close the currently opened database it will send the following request:

```
"closedb[]<CR><LF>
```

Assuming that the operation was sucessful, the reply of the Database Server would be:

```
"closedb[[],[],0]<CR><LF>
```

Notes about the closedb operation:

1) This operation must be called in the end of the session.

2) This operation is not sucessful if there is no database

currently opened. Otherwise it is always sucessful.

3.3 Retrieve

This operation is used either to validate a tuple over the database or to fetch information in order to complete a certain tuple. In the first case there will be no variables in the request and in the second case the fields to be retrieved will be represented by variables. The actual format of its **request** is the following:

```
^retrieve[^name[arg1,...,argn]]<CR><LF>
```

The **reply** to this query must specify, when needed, the tuple that was found, the list of pointers to the other solutions, when they exist, and the return code, therefore its format will be the following:

```
^retrieve[[tuple],[pointers],return]<CR><LF>
```

Example 3.3.1: (Retrieve without variables)

Let us suppose that we have a database with facts about persons and their addresses, thus the following table:

ADDRESS

NAME	PLACE
Raf	Leuven
Jose	Heverlee

Which is represented, in Prolog, by:

```
address(Raf,Leuven).
address(Jose,Heverlee).
```

The Prolog query: ?-address(Raf,Leuven). (is Leuven the address of Raf?) would generate a database question, and the following request would be sent to the Database Server, through the pipe-line:

```
^retrieve[^address['Raf','Leuven']]<CR><LF>
```

This request is succesful because in fact the address of Raf is stated in the table of the relation "address" as beeing Leuven so the following reply would be sent to Prolog:

```
"retrieve[[],[],0]<CR><LF>
```

where the two []'s state that there is no need of sending information back to Prolog (the first), and that there are no further solutions to this query (the second).

Example 3.3.2: (Retrieve with one variable)

In the same database, the Prolog query: "?-address(Jose,_X)." (where does Jose live?) would generate the following request to be sent to the Database Server:

```
"retrieve["address['Jose',"-1]]<CR><LF>
```

The last argument of the relation address represents the variable X of the Prolog query. It's value is thus the aim of the query. The reply for this query would be:

```
"retrieve[['Jose','Heverlee'],[],0]<CR><LF>
```

where the [] specifies that there are no further solutions to this query.

Example 3.3.3: (retrieve with a void variable)

Let us now suppose that in the former query Prolog was not interested in the value of the variable X. This means that the query would be: has Jose an address? and it could be written in Prolog syntax: "?-address(Jose,_)." .

This difference would generate a request just like the previous one but with a difference in the representation of the variable which is now void:

```
"retrieve["address['Jose',"_]]<CR><LF>
```

As the variable is now void Prolog just needs an yes-no answer, therefore the reply will have two []'s, in the tuple and in the pointer list. The return will be 0 because the operation is sucessful.

```
"retrieve[[],[],0]<CR><LF>
```

Example 3.3.4: (Retrieve with backtracking points)

Let us now suppose that our previous database table for the relation "address" was the following:

ADDRESS

NAME	PLACE
Raf	Leuven
Jose	Heverlee
Yves	Heverlee
Bart	Heverlee

The Prolog query: "?-address(_X,Heverlee)." (who lives in Heverlee?) would generate the following request:

```
"retrieve["address["-1,'Heverlee']]<CR><LF>
```

In this case, the first solution is the tuple ['Jose','Heverlee'] and there are 2 other solutions. Let's suppose that they have pointers "PNT1 and "PNT2. The reply would be:

```
"retrieve[['Jose','Heverlee'],["PNT1,"PNT2],0]<CR><LF>
```

If the Prolog system needs to backtrack on this query then it will send a "furthertupleR" query with "PNT1 or "PNT2. We will come back to this subject in paragraph 3.8.

Notes about the retrieve operation:

1) The contents of the database remains unchanged after a retrieve operation. This operation just accesses the information stored in the database.

2) The reply only contains explicitly the first solution to the database query. The other solutions are sent in the form of pointers directly to the tuples and will eventually be used with the question "furthertupleR" when the Prolog system backtracks. We will discuss this subject in paragraph 3.8.

3) When a query only contains void variables the database system doesn't have to retrieve their values. Therefore the reply will have []'s in the tuple and in the pointer list.

3.4 Createrelation

This question adds the specification of a database relation to the schema of the database.

This operation sends to the Database Server the description of the relation in terms of its name, its arity and some characteristics of its arguments.

For the moment, the Prolog system will only send to the Database Server, as characteristics of the relation arguments, its uniqueness types.

Those uniqueness types can be used by the Database Server to determine which argument or combination of arguments constitutes the primary key and what type of indexing, if any, is needed for the other arguments. On the other hand the combination of the uniqueness types of the several arguments can be used to determine if there is the possibility of having duplicate records in that relation, or not.

Other characterics like the field names, field synonyms and field types must, for the moment, be settled by default by the Database Server.

The format of this **request** is the following:

```
"createrelation["name['UniqType1',...,'UniqTypen']]<CR><LF>
```

where UniqTypei = NonKey, Key, KeyPart or OptionalKey.

Example 3.4.1:

Suppose that we want to create a relation with name "person", with four fields, with the key composed by the first and the second fields, an optional key on the fourth field and no key on the third field. The request for such a creation would be:

```
"createrelation["person['KeyPart','KeyPart',
                        'NonKey','OptionalKey']]<CR><LF>
```

Assuming that the operation was sucessful the reply would be:

```
"createrelation[[],[],0]<CR><LF>
```

Notes about the createrelation operation:

1) The effect of this operation is to create, in the data ditionary, a definition of a relation with the specified characteristics.

2) This operation is unsucessful when there already exists a definition for that relation in the data ditionary or when an internal error occurs. In these cases the error code is transmited to the Prolog system in <return>.

3) Remark that this operation changes the data ditionary by adding a new definition to it. We can only insert information in the table of a relation after creating its entry in the data ditionary.

4) In the future, characteristics like the argument types and argument names will also be provided by the Prolog system to the Database Server.

3.5 Insert

This is one of the simplest database queries because it has no variables. The format of its **request** is the following:

```
"insert["name[tuple]]<CR><LF>
```

Note that this operation will only add records to previous existing tables. The role of creating new relations belongs to the operation "createrelation" (described in the last paragraph) and not to "insert".

Example 3.5.1:

Suppose that, in the context of the previous examples, we want to add to the database relation "address" the Prolog unit ground clause: "address(Pol,Brussels)." which can be read as: "the address of Pol is Brussels".

The request that the Prolog system would sent to the Database Server would be:

```
"insert["address['Pol','Brussels']]<CR><LF>
```

Assuming that the operation was successful the reply that Prolog would get as answer would be:

```
"insert[[],[],0]<CR><LF>
```

because there is no need for sending the tuple back to Prolog and because there are no other solutions to this query.

Notes about the insert operation:

1) The contents of the database is changed if the insertion is

sucessful. Depending on the definition of the relation this operation is successful or not if the record already exists in the database. If an internal error occurs then its code will be returned to Prolog.

2) In the request for insertion there are never variables because in Prolog the external database can only contain unit ground clauses.

3) The arguments of the relations are, for the moment, always atoms, ie, there can be no functors with arity greater than 0 inside those arguments. Later this restriction will be removed and the arguments of the relations will have the possibility to be any Prolog term.

3.6 Delete

This operation is used to delete a record in a database table. When it is called the record may or may not be completely specified. In the later case the values of the fields that were not specified must be returned to the Prolog system. In both cases the pointer to the record must be returned to Prolog together with the pointers for the, eventually existing, other solutions. This pointers will be used with the operation "furthertupleD" (see 3.9) when the Prolog system backtracks. The format of the **request** for this operation is the following:

```
"delete["name[tuple]]<CR><LF>
```

Note that as far as Prolog is concerned it makes no difference if we are deleting the last record of the relation, and its table will be empty from then on, or if we are deleting one of its records and there will remain more others.

In fact, when the last record of a table is deleted, the table remains existing. The role of deleting a table is performed by "deleterelation" as we will see in the next paragraph.

Example 3.6.1:

Suppose that, in the context of the previous examples, we want to delete the information about the address of Raf but we don't know what it is.

The request that would be sent to the Database Server would be:

```
"delete["address['Raf',"-1]]<CR><LF>
```

Assuming that the operation was sucessful and that only one solution was found, the reply that Prolog would get as answer would be:

```
"delete[['Raf','Leuven'],["PNT1],0]<CR><LF>
```

If the Prolog system was not interested in the value of variable -1, that means, if the Prolog query was: delete the address of Raf whatever it is, then the request for this query would be very similar to the one presented above with the only exception that the representation of the variable would change:

```
"delete["address['Raf',"_]]<CR><LF>
```

The reason for this is that the variable is now void so it doesn't matter what value it has. Therefore the reply doesn't need to mention the tuple, however the pointer has extrem importance:

```
"delete[[],["PNT1],0]<CR><LF>
```

Notes about the deletion operation:

1) The contents of the database is changed if the deletion is sucessful. This operation is not sucessful if the record doesn't exist in the database. If an internal error occurs its code will be returned to Prolog.

2) In the reply of this operation there may exist tuples depending on the existance of variables in the request transmitted before.

3) Remark also that, although **n** pointers are returned to the Prolog system, only the tuple corresponding to the first one is **in fact** deleted.

3.7 Deleterelation

This operation is used to delete completely a database relation (table and definition). In the answer of this query only the return is important because the tuple and the pointer list are []'s. When this query is performed Prolog has the warranty that it can be done in order to prevent user's mistakes. The format of its **request** is the following:

```
"deleterelation['name']<CR><LF>
```

Example 3.7.1:

Let's suppose that we have the previous database with the table for the relation "ADDRESS" and that we want to delete it. The request for this question would be the following:

```
"deleterelation['address']<CR><LF>
```

Assuming that the operation was sucessful, the answer would be the following reply:

```
"deleterelation[[],[],0]<CR><LF>
```

And from then on neither the table nor the definition of "address" will exist.

Notes about the deleterelation operation:

1) The contents of the database is deeply changed if the deleterelation is sucessful. This operation is not sucessful if the relation doesn't exist in the database or if the user has unsufficient priviledge (in Unix sense) for the operation. If an error occurs its code will be returned to Prolog.

3) Remark the difference between "delete" and "deleterelation". In the former case just a record is deleted and in the later case the whole relation is deleted (both the table and the entry in the data ditionary). Even when we delete the last record of a certain relation with "delete", the table continues to exist, ie, the relation is not deleted from the database schema. However, when we use "deleterelation" the table and the name of the relation no longer exist, ie, the relation is deleted from the database schema.

3.8 FurthertupleR

As we have seen in the retrieve operation, its reply only contains explicitly the first solution to the query. However, when there exists more solutions, the Database Server sends, in the pointer list, all the pointers that point directly to them. When Prolog backtracks this information is all what it needs in order to get the further solutions.

This primitive "furthertupleR" is the one that sends to the Database Server a certain pointer and gets back the tuple to which it points. So when Prolog wants another solution for a previous "retrieve" query it sends to the Database Server a request of "furthertupleR" with the pointer as argument and it gets back the tuple.

The format of its **request** is the following:

```
^furthertupleR[^pointer}<Ck><LF>
```

Let's see how it works through an example.

Example 3.8.1:

Let's now go back to example 3.3.4 and supose that the Prolog system needs to backtrack the query (?-address(_X,Heverlee).). Remember that, as answer to the first "retrieve" it got a pointer list consisting of the pointers ^PNT1 and ^PNT2. In order to get the other solutions the Prolog system just has to request:

```
^furthertupleR[^PNT1]<CR><LF>
```

and it will get as reply:

```
^furthertupleR[['Yves','Heverlee'],[],0]<CR><LF>
```

whose tuple is another solution to the initial query. If it needs to backtrack again, it will send to the Database Server:

```
^furthertupleR[^PNT2]<CR><LF>
```

and it will receive back:

```
^furthertupleR[['Bart','Heverlee'],[],0]<CR><LF>
```

which is the last solution to the initial query.

Notes about the furthertupleR operation:

1) The only operation which is backtrackable using "furthertupleR" is the "retrieve". The operation "delete" is backtracked using "furthertupleD" that will be discussed in the next paragraph.

2) Prolog stores the pointer list it receives in the reply of the initial "retrieve" in a stack that will be consumed pointer by pointer in backtracking. Each pointer is used to call "furthertupleR".

3) The pointer list of this operation is always empty because Prolog already has the pointers to the other possible solutions.

3.9 FurthertupleD

This operation is the equivalent for "delete" of "furthertupleR". As we have seen in paragraph 3.6, the reply of a "delete" request contains a pointer list with pointers to all its possible solutions.

When the Prolog system needs to backtrack that "delete" operation it calls "furthertupleD" and sends the pointer for the tuple that will be deleted.

The format of its **request** is the following:

```
"furthertupleD["pointer]<CR><LF>
```

and the format of the **reply** is:

```
"furthertupleD[[tuple],[],return]<CR><LF>
```

Let's see an example of how it works.

Example 3.9.1:

Suppose we have the relation "address" with the table of example 3.3.4 and that we want to delete the Prolog clause:

```
address(_X,Heverlee).
```

The request for this "delete" would be:

```
"delete["address["-1,'Heverlee']]<CR><LF>
```

As answer to this delete the Prolog system would receive the reply:

```
"delete[['Jose','Heverlee'],["PNT1,"PNT2,"PNT3],0]<CR><LF>
```

With this reply Prolog knows that the further solutions of this delete are pointed by "PNT2 and "PNT3. Therefore if it needs to backtrack the initial "delete" it will send the request:

```
"furthertupleD["PNT2]<CR><LF>
```

The reply to this request would be:

```
"furthertupleD[['Yves','Heverlee'],[],0]<CR><LF>
```

If the Prolog system needs to backtrack again the initial "delete", it will call "furthertupleD" again, but for the pointer "PNT3.

Notes about the furthertupleD operation:

1) This operation is used to backtrack the "delete" operation. Therefore there are two different operations "furthertupleR" and "furthertupleD" to backtrack the only two operations that it is possible to backtrack, "retrieve" and "delete".

2) Prolog stores in a stack the pointer list it receives in the reply of the initial "delete" and consumes this stack in bascktracking.

3) The pointer list of this operation is always empty because Prolog already has the pointers to the other possible solutions.

4. Bibliography

<Ven84> Venken, R. and Adler, H. D., Report DB1 and DB2: the interaction between Prolog and a relational database system,
internal LOKI report (ESPRIT pp 107), feb 1984.