Implementation of a Prolog–INGRES Interface

S. Ghosh, C.C. Lin and T. Sellis[†]

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This report describes a working prototype of a Prolog–INGRES interface based on external semantic query simplification. Semantic query simplification employs integrity constraints enforced in a database system for reducing the number of tuple variables and terms in a relational query. This type of query simplifier is useful in providing very high level user interfaces to existing database systems. The system employs a graph theoretic approach to simplify arbitrary conjunctive queries with inequalities. One very interesting feature of the system is to provide meaningful error messages in case of an empty query result resulting from contradiction. In addition to data, rules are stored in the database as well and are retrieved automatically if the Prolog program references them but they are not defined in the Prolog rulebase.

[†] Also with University of Maryland Systems Research Center and Institute for Advanced Computer Studies (UMIACS).

1. Introduction

Large scale knowledge bases require more intelligent processing than current Data Base Management Systems (DBMS) can offer as well as more intelligent access to large scale databases than current expert systems offer. Therefore, it is widely recognized that we need to somehow combine the intelligent query processing part of expert systems with the efficient access techniques of DBMS's [KERS86a,KERS86b]. Integrating logic programming (in particular Prolog) and databases looks very promising since both logic programming and relational databases are related by their common ancestry of mathematical logic [GALL84].

Prolog alone cannot meet the requirement of supporting large databases. However, it constitutes an attractive domain–oriented query language for relational databases, where the specification of joins is easier than in tuple–oriented languages. The power of Prolog as a pure logic programming language (Horn Clauses only) surpasses that of relational calculus, since it is relationally complete. Besides, the power of recursion in Prolog makes it superior to relational calculus in applications with recursive query requirements. However, Prolog is more than just a logic programming language, since it offers features and embodies an execution model that turns it into a general–purpose programming language. This generality adds to Prolog's desirability as a practical database language, since complete applications can now be developed in it. This contrasts with the current approach to the development of database–intensive applications that uses a general purpose programming language with embedded database query statements.

A system which combines a Prolog front–end with a database back–end appears to be a very promising vehicle for developing database and knowledge–based applications and has received a lot of attention in the last few years [BROD86,CERI86,CHAN86, JARK84,JARK86,SCIO86]. An obvious benefit expected is faster execution of Prolog programs, since database predicates can be off–loaded to the database system for more efficient, and possibly parallel, execution. This is particularly useful, when the ability is needed of making inferences over large volumes of data, such as in data–intensive expert systems. The second benefit lies in the enhanced functionality of the database system, due to the newly accrued inferential capability. Also in such a system, programmers will be able to develop the entire application in Prolog (a complete programming language) whereas presently they must resort to a conventional programming language with an embedded query language.

In this paper we report on the implementation of a scheme for integrating Prolog with a relational database system, INGRES, based on the ideas of [JARK84,JARK86]. In this approach there is minimum interaction between the two systems, achieved by an optimizing translation mechanism. Using a variable free subset of Prolog as an intermediate language makes the query simplifier independent of the input and target database language. In addition to this sophisticated optimizer, we have added the capability of storing not only data but rules as well in the database. This adds persistence to Prolog, in the sense that after a session is over and a new session starts, the user needs not re–assert the knowledge asserted in the past.

This report focuses mainly on describing the querying facility of the system. In Section 2, we describe the overall architecture. Section 3 discusses the implementation of the query simplifier while Section 4 presents the implementation of the module for handling rules. Finally we conclude in Section 5 with a summary and current work on the system.

2. System Description

The configuration of the system is shown in Figure 1. In the following sub–sections we discuss the Data and Rule Definition and Query facilities, and the way Prolog communicates with

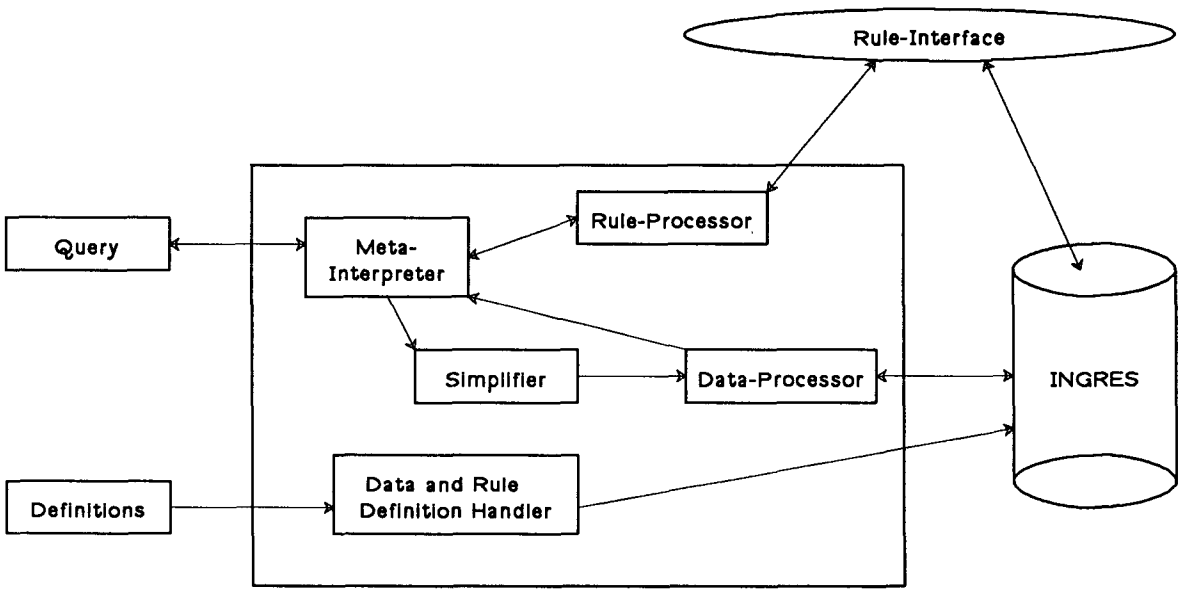

Figure 1: System Architecture

INGRES.

2.1. Data and Rule Definition

This interface allows the user to define a database schema, including relations, indices etc. Moreover, the user can assert key dependencies, more general functional dependencies, referential integrity and value bound constraints. For example, the following define the example database used throughout this presentation [JARK86]

```
define_schema(employee,[eno,i2,ename,c20,salary,i2,dno,i2]).
define_schema(department,[dno,i2,function,c4,mgr,i2]).
define_keydep(employee,[eno]).
define_keydep(department,[dno]).
define_funcdep(employee,[ename],[eno]).
define_funcdep(department,[mgr],[dno]).
define_valuebound(employee,salary,10000,90000).
define_refint(employee,[dno],department,[dno]).
define_refint(department,[mgr],employee,[eno]).
```

Some of these requests are directly translated to INGRES commands, like for example the command for creating a relation. The constraints are stored in relations in the database in a way similar to the one we use to store rules (see section 4). We will not elaborate more on the definition facility. We turn now to describe the more interesting part of the system, that is handling user requests.

2.2. Querying the System

User queries are first processed by the Meta–Interpreter which examines if all rules needed to process the given request are already asserted in Prolog. Rules not currently in the Prolog rule base are passed to Rule–Processor which in turn asks the Rule–Interface to retrieve these rules from the INGRES database. The Simplifier is used to optimize and simplify queries using functional dependencies, referential integrity and value bound constraints. The Data–Processor extracts all database references and generates a QUEL command to retrieve the data from INGRES. The translation from Prolog queries to QUEL commands is achieved by means of an intermediate language used by the Simplifier, called DBCL. Hence, processing can be partitioned into three major components: the translation of a Prolog query to DBCL statements, simplification of the DBCL statements and the translation of the optimized DBCL statements to queries in the target database language. There is also another module not shown in Figure 1, the Attribute–Processor, which is an initialization routine that at start–up time retrieves all the information regarding user relations currently stored in INGRES and needed by the Data–Processor.

2.3. Communication Between Prolog and INGRES

The major problem in interfacing Prolog and INGRES is how to call INGRES from Prolog and get the result back. The first problem can be solved easily by using the `"system"` predicate. For example, `system("ls")` will list the current directory. To call INGRES, `system("ingres DBname")` is enough. However, there is no easy way to pass commands to INGRES from Prolog, therefore, INGRES commands are passed through a file, say, `ingres.command`. Thus, the whole calling sequence is `system("ingres DBname -s < ingres.command")`. The –s option prevents INGRES from printing any message on the screen, which is not needed in this case.

The second and more difficult problem is how to get the result that is retrieved from INGRES back to Prolog. In C–Prolog 1.4, the version of Prolog we use in the current implementation, there is no system function provided for communication with other processes, except the `system` predicate. The only resort is then to ask INGRES to write the result into a file that can be then read by Prolog. Therefore, getting a result from INGRES is really a two–step process: first INGRES retrieves the necessary information specified by the command file, stores it in a temporary relation and finally copies this temporary relation to a plain Unix file. Second, Prolog reads the file and asserts the information in its database. The reason we have to copy the temporary relation to a Unix file is because an INGRES relation file is readable only by INGRES itself.

Invoking INGRES in the way described above is very time–consuming because INGRES has to be setup and terminated for each request from Prolog. A more efficient method is to set up a pipe between Prolog and INGRES which cannot be done in C–Prolog 1.4. However, the Rule–Processor does not use this method because of performance considerations, namely the fact that rules are needed at various places during query processing. Clearly, one does not want to access INGRES every time a rule is needed. For that reason we pre–fetch all rules needed to process a user query. A C program, the Rule–Interface, is provided to retrieve all relevant rules from INGRES. The Rule–Interface is described in more detail in Section 4.

3. Query Simplifier

The query simplification process is based on a graph–theoretic approach similar to that described in [JARK86] and is shown in Figure 2. The simplifier utilizes integrity constraints enforced in the DBMS for transforming queries into a form that can be executed more efficiently.

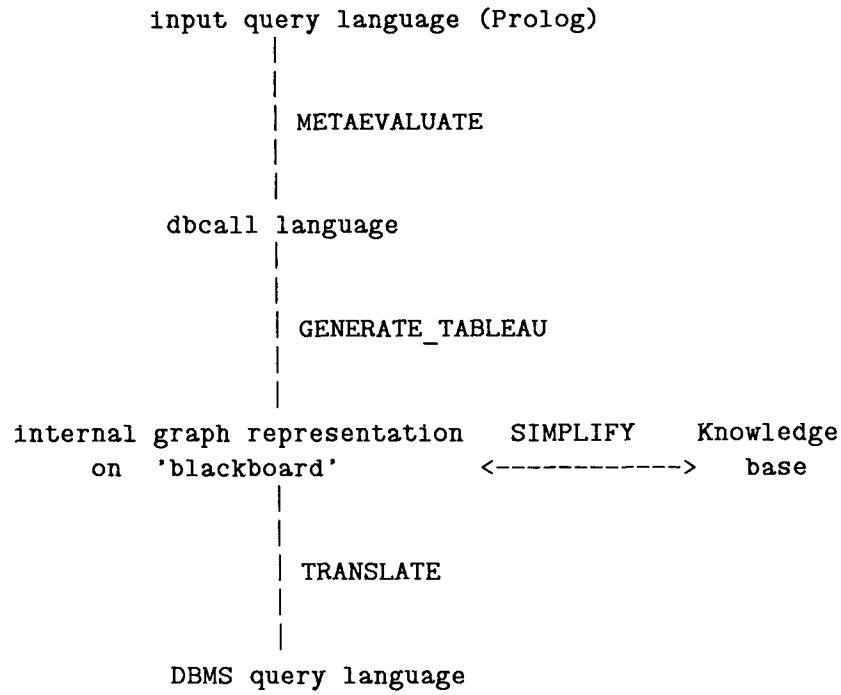

Figure 2: Structure of the semantic query simplifier [JARK86]

We have considered only those integrity constraints which do not depend on the actual database state at any time. In particular we have considered functional/key dependencies, referential constraints and range (or value bounds) constraints on ordered domains. The query simplifier consists of two translation mechanisms and a knowledge base. The first translation is the meta–evaluation of the Prolog query into an intermediate language, DBCL, which is set oriented and uses only base relations. The purpose of this is to collect tuple oriented Prolog requests to form set oriented queries. After this translation, each query is a list of `"dbcall"` predicates of the form:

```
dbcall(Relation_name,List_of_tableau_entries)
```

or

```
dbcall(Operator,Left_operand,Right_operand)
```

where `Operator` can be one of: `equal,notequal,less,lessequal,greater,greaterequal`. The second translation is the translation of the optimized DBCL statements into queries in the target database language, in our case QUEL. The knowledge base contains a schema definition and predicates describing the integrity constraints in the format described in [JARK86]. Examples (taken from [JARK86]) of knowledge base, Prolog view and DBCL query are shown in Figures 3, 4 and 5 respectively.

3.1. A Graph Based Simplification Algorithm

We have implemented in Prolog the simplification algorithm of [JARK86]. The query simplifier uses a graph representation for representing inequalities and dependencies. Inequalities

```
schema(employee,[eno,ename,salary,dno]).
keydep(employee,[eno]).
funcdep(employee,[ename],[eno]).
valuebound(employee,salary,10000,90000).

schema(department,[dno,function,mgr]).
keydep(department,[dno]).
funcdep(department,[mgr],[dno]).

refint(employee,[dno],department,[dno]).
refint(department,[mgr],employee,[eno]).
```

Figure 3: Example of a knowledge base for the query simplifier

```
works_dir_for(X,Y):-
    employee(Eno1,X,Sal1,D),
    department(D,Fct,M),
    employee(M,Y,Sal2,Dno1),
    notequal(X,Y).
```

Figure 4: Example Prolog view

```
[dbcall(employee,[v_eno1,t_X,v_sal1,v_dno1]),
 dbcall(department,[v_dno1,v_fct1,v_mgr1]),
 dbcall(employee,[v_mgr1,smiley,v_sal2,v_dno2]),
 dbcall(employee,[v_eno2,t_X,v_sal3,v_dno3]),
 dbcall(notequal,t_X,smiley),
 dbcall(lessequal,v_sal3,40000)]
```

Figure 5: DBCL equivalent of the query "Who works directly for Smiley and makes less than or equal to 40000" and obtained by meta–evaluating the query :–(works_dir_for(t_X,smiley),employee(_,t_X,S,_),lessequal(S,40000))

---

are represented by explicit edges in the graph and dependencies are represented by implicit edges. The DBCL query is converted into graph form as in [JARK86]. The nodes in the graph correspond to all entries appearing in the DBCL query plus a node *0(d)* for each ordered domain *d*. Edges correspond to the comparisons in the DBCL query. In addition to what is reported in [JARK86], we have implemented limited optimization for queries with notequality constraints.

3.2. Inequality Optimization

Notequal–free Optimization

We have considered notequality separately because generalized inequality optimization is NP–hard [ROSE80]. The graph based algorithm for inequality optimization can be described as follows:

1. Remove all multiple edges between all pairs of nodes.
2. Find the shortest distance between all pairs of nodes [FLOY62].
3. If a negative length cycle is found then stop with an error message and empty query result. The query unsatisfiable.
4. Remove all variables from the graph that occur in zero length cycles with some node *0(d)* and rename variables to a constant corresponding to the length of the path from each node on the cycle to node *0(d)*.
5. For each set of variables which belong to the same zero length cycle, retain in the graph one of the variables and rename the rest of the variables to the name of the variable retained.
6. Delete all redundant edges from the graph. An edge from node x to node y with length c is redundant if there exists a path different from this edge from x to y with length less than or equal to c.

Notequal Optimization

We have avoided generalized notequal optimization because it incurs exponential time. Instead we have considered only special cases. For example if during the simplification process we find that there exists a path from node x to node y of length c (i.e $x \leq y+c$) and there is a notequal comparison ($x \neq y+c$), then the former condition is replaced by a sharper one ($x < y+c$). An example of inequality optimization is shown in Figure 6. The first inequality `(greater,v_sal3,5000)` is removed since it is implied by the value bound on `salary` attribute

```
% a query with redundant comparisons

[dbcall(employee,[v_eno1,t_X,v_sal1,v_dno1]),
 dbcall(department,[v_dno1,v_fct1,v_mgr1]),
 dbcall(employee,[v_mgr1,v_man,v_sal2,v_dno2]),
 dbcall(employee,[v_eno2,t_X,v_sal3,v_dno3]),
 dbcall(greater,v_sal3,5000),
 dbcall(equal,v_man,smiley),
 dbcall(lessequal,v_sal3,40000),
 dbcall(lessequal,v_sal3,60000),
 dbcall(notequal,v_sal3,40000),
 dbcall(notequal,v_sal3,60000)]

% equivalent query after inequality optimization

[dbcall(employee,[v_eno1,t_X,v_sal1,v_dno1]),
 dbcall(department,[v_dno1,v_fct1,v_mgr1]),
 dbcall(employee,[v_mgr1,smiley,v_sal2,v_dno2]),
 dbcall(employee,[v_eno2,t_X,v_sal3,v_dno3]),
 dbcall(lessequal,v_sal3,39999)]
```

Figure 6: Example of inequality optimization

as given in Figure 3. (lessequal,v_sal3,60000) is removed since it is implied by (lessequal,v_sal3,40000). v_man is renamed to smiley at step 5 of the algorithm. (notequal,v_sal3,40000) and (lessequal,v_sal3,40000) are combined to (lessequal,v_sal3,39999). (notequal,v_sal3,60000) is discarded because it is implied by (lessequal,v_sal3,39999).

3.3. FD/KD Optimization

The congruence closure of the FD/KD graph is computed in a breadth first fashion by applying a variation of the first chase algorithm [DOWN80]. The algorithm terminates when no further FD or KD is applicable to an entry in the DBCL query. As an example, consider the simplified DBCL predicate in Figure 6. At level 0, only one functional dependency

```
funcdep(employee,[ename],[eno])
```

is applicable, leading to the renaming of v_eno2 by v_eno1. At level 1,the key dependency for the employee relation becomes applicable, leading to the removal of the fourth row and renaming of v_sal1 to v_sal3 in the first row. The simplified query appears in Figure 7. The application of a functional or key dependency results in adding edges to the graph (because two elements are made equal). Due to this change in the graph, the inequality optimization is examined again and thus, these two optimization steps proceed in an interleaved fashion until nothing changes in the graph.

3.4. Referential Integrity Optimization

The application of referential integrity constraints allows the deletion of certain "dangling" rows from the DBCL query and thus reducing the number of variables and join operations in the resulting query. The algorithm works as follows

1. Check all rows of the DBCL query to see if a row dangles [JARK84]
   if 'yes' then go to step 2
   else stop.
2. Delete the dangling row if it is deletable [JARK84].
   go to step 1.

As an example consider the DBCL query in Figure 8. The third row dangles; it is also deletable since v_mgr1 appears in the second row and there is an applicable referential integrity constraint between mgr in department and eno in employee. After the deletion of the third row, the second row also dangles and is also deletable. The final DBCL query is shown in Figure 9.

```
[dbcall(employee,[v_eno1,t_X,v_sal3,v_dno1]),
 dbcall(department,[v_dno1,v_fct1,v_mgr1]),
 dbcall(employee,[v_mgr1,smiley,v_sal2,v_dno2]),
 dbcall(lessequal,v_sal3,39999)]
```

Figure 7: Simplified version of the DBCL query of Figure 6 after application of FD/KD's

```
[dbcall(employee,[v_eno1,t_X,v_sal1,v_dno1]),
 dbcall(department,[v_dno1,v_fct1,v_mgr1]),
 dbcall(employee,[v_mgr1,v_ename1,v_sal2,v_dno2]),
 dbcall(employee,[v_eno2,smiley,v_sal3,v_dno1]).
```

Figure 8: Example of DBCL query before referential integrity optimization

```
[dbcall(employee,[v_eno1,t_X,v_sal1,v_dno1]),
 dbcall(employee,[v_eno2,smiley,v_sal3,v_dno1]).
```

Figure 9: Simplified version of the DBCL query of Figure 8 after application of referential integrity constraints

---

4. Rule–Processor and Rule–Interface

As mentioned in the introduction, in some applications which contain huge sets of rules, Prolog may not serve very well because it may not be able to store all the rules in the main memory. As expert systems become more and more complicated, this kind of application would be very common [HAYE87]. To make Prolog useful in this case, we need a way to store rules in a traditional database and retrieve them dynamically when required by the user query. To do this, a meta–interpreter (*shell*) as stated in [STER86] is a good candidate for this purpose. However, the shell must examine the existence of rules required by the query before trying to solve the goal(s). Rules mentioned but not defined should be retrieved from the database and asserted in the Prolog database. In the following sub–sections we describe our solution to the problem.

4.1. How to Store Rules in INGRES

Two relations are needed to represent rules in an INGRES database, namely `RULE` and `RULEBODY`. The schema of `RULE` is

```
ruleno    rule number
head      predicate name
numarg    number of arguments
arg       argument list (enclosed by parentheses)
access    access indicator, 0 if not accessed, 1 otherwise
```

and the schema of `RULEBODY` is

```
ruleno    rule number
bodyno    body number
pname     predicate name
numarg    number of arguments
arg       argument list (enclosed by parentheses)
```

For example, the rule

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

may be represented as

RULE Table				
ruleno	head	numarg	arg	access
1	ancestor	2	(X,Y)	0
2	ancestor	2	(X,Y)	0

RULEBODY Table				
ruleno	bodyno	pname	numarg	arg
1	1	parent	2	(X,Y)
2	1	parent	2	(X,Z)
2	2	ancestor	2	(Z,Y)

The original rule can be easily reconstructed by taking the natural join of RULE and RULEBODY on field ruleno. The access bit is used to tell whether the corresponding rule has been retrieved into the Prolog rule base or not. If access="0" then the rule has not been retrieved yet, in other words, it is not defined in Prolog, thus should be accessed if needed.

4.2. The SHELL

The SHELL used in the current implementation is basically the same as Program 12.6 in [STER86] with some modification. However, before trying to solve a goal, it must first search the rulebase to see if that predicate is defined. If the predicate exists, it can be called directly as usual, otherwise the SHELL will first call the Rule–Processor to retrieve the rule and then apply it. For example, if the query is

```
fib(5,X), factorial(X,Y), plus(Y,10,Z).
```

and the predicate factorial is already defined while no definition is given for fib and plus, the SHELL will call the Rule–Processor which will in turn call a C routine (the Rule–Interface) to retrieve the definition of fib with arity 2 and plus with arity 3 from INGRES. The reason that we don't want Prolog to call INGRES directly in this case is merely performance. To get the definition of a rule, the system may have to call INGRES several times before all sub–rules are retrieved. On the other hand, since C can talk to INGRES directly, we only have to call INGRES once to get all the rules if we use C as an intermediate interface between Prolog and INGRES. The flattening process can be done in C quite easily which may be very difficult, if not impossible, to be done in QUEL due to the absence of repeat commands.

4.3. Rule–Interface

The way a C routine can talk to INGRES is by using EQUEL, i.e. by embedding QUEL commands in C routines. The main function of Rule–Interface is to retrieve rules needed by Prolog which are passed as command line arguments. For example, to retrieve rules fib and plus of the above example, the calling method is

```
rule fib 2 plus 3
```

where 2 and 3 are the arities of fib and plus respectively. Only rules with the same arity as

specified will be retrieved.

Once retrieved, all rule definitions needed are written into a file called, say `rule.result`. The format of entries in this file is

```
tuple(ruleno,head,arg,bodyno,pname,barg)
```

where `head` is the name of predicate at the left hand side of the rule and `bodyno` and `pname` are the sequence number and name of predicates at the right hand side of the rule respectively. `arg` and `barg` are arguments enclosed by parentheses for predicates at the LHS and RHS of the rule respectively. For instance, the `ancestor` example will return 3 tuples

```
tuple(1,'ancestor','(X,Y)',1,'parent','(X,Y)').
tuple(2,'ancestor','(X,Y)',1,'parent','(X,Z)').
tuple(2,'ancestor','(X,Y)',2,'ancestor','(Z,Y)').
```

The Rule–Interface also provides a function "*reset_rule*" to reset the access bits of all rules in the relation `RULE`. This function is called every time one starts up a session with the system. Another function of the Rule–Interface, "*define_rule*", accepts rule definitions from the user and inserts them into the `RULE` and `RULEBODY` relations.

4.4. Rule–Processor

The main function of the Rule–Processor is to check if all rules needed to process a query exist in Prolog's rulebase; if not, it calls the Rule–Interface to retrieve them from the database and then asserts them into Prolog's rulebase. To reconstruct rules retrieved from INGRES, the file `rule.result` is read and `tuple()` predicate entries are asserted. Rules can be grouped together by `ruleno` in the `tuple` predicate and then the whole rule is written to another file called `rulebase`. After all rules have been reconstructed, `rulebase` is then consulted and the rules are asserted to Prolog's rulebase.

It may seem that writing to the second file, `rulebase`, is not really needed in the sense that rules can be reconstructed and then asserted directly without much trouble. However, asserting something like

```
fib(0,1).
```

which is assembled by the string `"fib"` and `"(0,1)"`, is interpreted by Prolog as

```
'fib(0,1)'.
```

i.e. quoted predicate, which changes completely the semantics. Writing rules to a file and then consulting it is the easiest method to get around this problem.

5. Summary

We have described one of the successful techniques to integrate a logic based deduction system with a relational database system. This integration approach may not be as efficient as a tightly integrated expert database system which utilizes more sophisticated integrity constraints and has full information about the database state at any given time. However, our prototype was easy to implement (took about three weeks) and may be useful for existing database systems where it is expensive to change the code of the DBMS. Its main advantages are

(1) The user can write the whole application in Prolog and use information already stored in an INGRES database without knowing the INGRES system.

(2) The schema of the database is loaded by the system automatically.

(3) Both rules and data are stored in the database. Rules used in a query but not defined in Prolog, will be retrieved from the database automatically.

We are currently working on extending the system to support more general queries. At present the system handles disjunction and recursion but very inefficiently. It handles disjunction by converting the DBCL query into disjunctive normal form and then generating a query for each of the conjunctions. Since it does not consider the interaction among these disjunctions, it is not very efficient. We are investigating ways to introduce the ideas of [SELL86] in our current prototype. For a recursive query the system will generate a sequence of non–recursive queries to be processed one–at–a–time from INGRES. We are also working on the idea of recognizing recursive queries and sending them to INGRES to be processed by an EQUEL program that will process a recursive query using iteration. Currently the whole system has around 230 clauses.

6. References

[CERI86] Ceri, S., Gottlob, G. and Wiederhold, G., "*Interfacing Relational Databases and Prolog Efficiently*", in [KERS86b].

[CHAN86] Chang, C.L. and Walker, A., "*PROSQL: A Prolog Interface with SQL/DS*", in [KERS86a].

[DOWN80] Downey,P.J., Sethi, R., and Tarjan, R.E., "*Variations on the Common Subexpression problem*", Journal of the ACM (**27**) 4, 1980.

[FLOY62] Floyd, R.W., *Algorithm 97: Shortest Path*, Communications of the ACM (**5**) 6, June 1962.

[GALL84] Gallaire, H., Minker, J. and Nicolas, J.M., "*Logic and Databases: A Deductive Approach*", ACM Computing Surveys, (**16**) 2, June 1984.

[HAYE87] Hayes–Roth, F., Invited Talk, IEEE Compcon, San Francisco, CA, February 1987.

[JARK84] Jarke, M., Clifford, J. and Vassiliou, Y., "*An Optimizing Prolog Front–End to a Relational Query System*", Proceedings of the 1984 ACM–SIGMOD Conference, Boston, MA, June 1984.

[JARK86] Jarke, M., "*External Query Simplification: A Graph Theoretic Approach and its Implementation in Prolog*", in [KERS86a].

[KERS86a] Kerschberg, L., **Expert Database Systems**, Proceedings From the First International Workshop, Benjamin/Cummings, Inc., Menlo Park, CA, 1986.

[KERS86b] Kerschberg, L., Editor, Proceedings of the First International Conference on Expert Database Systems, Charleston, SC, April 1986.

[ROSE80] Rosenkrantz, D.J. and Hunt, M.B., "*Processing Conjunctive Predicates and Queries*", Proceedings of the 6th VLDB Conference, Montreal, 1980.

[SCIO86] Sciore, E. and Warren, D.S., "*Towards an Integrated Database–Prolog System*, in [KERS86a].

[SELL86] Sellis, T., "*Global Query Optimization*", Proceedings of the 1986 ACM–SIGMOD Conference, Washington, DC, May 1986.

[STER86] Sterling, L. and Shapiro, E., **The Art of Prolog**, M.I.T Press, 1986.