HOW DOES AN EXPERT SYSTEM GET ITS DATA?
(Extended Abstract)

Yannis Vassiliou, James Clifford, and Matthias Jarke

Computer Applications and Information Systems
Graduate School of Business Administration
New York University

An Expert System (ES) is a problem-solving computer system that incorporates enough knowledge in some specialized problem domain to reach a level of performance comparable to that of a human expert. In the heart of an expert system lies the program that "reasons" and makes deductions ("inference engine"). To reason, knowledge both of general rules (e.g. if a person works for a company then he/she gets employee benefits) and of specific declarative facts (e.g. john works for nyu) is needed.

With few exceptions, little attention is given in ESs to the handling of very large populations of specific facts. Since early prototype ESs represented specific facts which were characterized by large variety and a very small population, the inefficiency of data handling was not an issue. As ESs increase in sophistication and ambition, they deal with applications requiring a very large population of facts, often in the form of existing databases manipulated by generalized DBMS.

This short paper (see [Vassiliou et al 1983] for more details) investigates the technical issues of enhancing expert systems with database management facilities in four stages, leading to the coupling of the ES with a large DBMS. Our vehicles are first-order logic (with Prolog) and relational database management.

1.0 ELEMENTARY DATABASE MANAGEMENT - STAGE 1

On the simplest level, the whole population of facts can be represented directly in the knowledge base of the expert system. While this approach is feasible for any expert system, using Prolog can take us a step further.

Relational databases can be represented directly in Prolog [1] as a listing of all instantiated predicates corresponding to relation tuples. In addition, Prolog can be used directly as a database query language since it has a very powerful inference engine in place (theorem prover). A query for suppliers of many parts may be defined as:

```
supplies_many(Sno) :- supply(Sno,Pno1,Qty1),
                      supply(Sno,Pno2,Qty2),
                      not(Pno1=Pno2).
```

This example illustrates both the query capabilities of Prolog, and the powerful mechanism for "generalized" views. Such views differ from the traditional DBMS views (database windows) in that with the use of variables they can accept parameters, essentially becoming "moving windows" over the database.

There are two limitations to using Prolog directly as a database system:

(a) Large Databases. Executing Prolog programs requires the assertions representing the database (instantiated predicates) to be in main memory. Even when the database can fit in main memory, and despite the fact that Prolog implementations are very efficient, there are indexing limitations.

(b) Generality. Simple-minded use of Prolog can offer only elementary data management facilities. For instance, there are no data dictionary or directory and no generalized set-oriented relational operations.

2.0 DBMS WITHIN EXPERT SYSTEMS - STAGE 2

A further step towards integrating the deductive capabilities of Prolog with database management capabilities can be taken by implementing a general purpose DBMS directly in Prolog. This provides a means of adding flexible and general data access mechanisms to the inference engine.

The feasibility of this has also been demonstrated by [Kunifuji and Yokota] in [Gallaire et al 1982]. However, their implementation of relational operators requires the user to know the entire scheme of each relation and the order of the attributes in the scheme. Our approach, by contrast, provides a

---

[1] For a clarification of Prolog's approach to relational database management, and in general for the use of logic in databases, [Gallaire and Minker 1978, Gallaire et al 1981, 1983] provide excellent references.

simple way to specify generalized operators acting on any relation and set of attributes [Vassiliou et al 1983].

Large databases are a problem with this approach, but for specific applications, clever indexing schemes that guide decisions about which portions of external files should be read into the internal database can be devised. However, these strategies are not easily generalizable.

The major limitation in this stage is that often an existing very large database may be needed in the expert system application. A generalized commercial DBMS will most probably be in place for this database. and it may be prohibitively costly to maintain a separate copy of the whole database for the ES.

3.0 LOOSE COUPLING WITH EXTERNAL DBMS - STAGE 3

Conceptually the simplest solution to the problem of using existing databases is to extract a snapshot of the required data from the DBMS when the ES begins to work on a set of related problems. The snapshot is stored in the internal database of the ES as described in the previous section.

This stage still may not be general enough. Since decisions for extraction must be made statically, loose coupling is not suitable if the portion of the database to be extracted is not known in advance.

4.0 TIGHT COUPLING WITH EXTERNAL DBMS - STAGE 4

An online communication channel between the ES and the DBMS is assumed at this stage. Queries can be generated and transmitted to the DBMS dynamically, and answers can be received and transformed into the internal ES knowledge representation. Thus in tight coupling the ES must know when and how to consult the DBMS, and must be able to understand the answers.

The naive use of the communication channel assumes the redirection of all ES queries, on predicates representing relations, to the DBMS for stored database relations. Any such approach faces at least two major difficulties:

A.- Number of Database Calls

Since the ES normally operates with one piece of information at a time (tuple), there may be a large number of calls to a database for each ES goal. This approach will generate a particularly inefficient version of a "nested iteration" query evaluation algorithm and will not make use of query optimization procedures of the DBMS to reduce the number of secondary storage accesses.

B.- Complexity of Database Queries

Prolog goals considered as queries can be substantially more complex than queries expressed in a database query language such as SQL (i.e. contains generalized views and recursion).

These difficulties can be overcome by collecting and jointly executing database calls rather than executing them separately whenever required by the ES. In practice, we use an amalgamation of the ES language with its own meta-language. This allows for a deferred evaluation of predicates requiring database calls, while at the same time the inference engine (theorem prover) of the ES is working. Since all inferences are performed at the meta-level (simulation of object-level proofs), we are able to bring the complex ES queries to a form where some optimization and direct translation to a set of DBMS queries is feasible.

The queries are directed to the DBMS, answers are obtained and transformed to the format accepted by the ES for internal databases. The ES can then continue its reasoning at the object-level. Each invocation of predicates corresponding to database relations will now amount to an internal ES database goal, rather than a call to an external DBMS.

This implementation is based on the high level description of the DEMO predicate presented in [Bowen and Kowalski 1982], and is similar to that of [Kunifuji and Yokota] in [Gallaire et al 1982]. Our work extends the above approaches by providing a more general treatment of evaluable predicates (e.g., finite negation (not) and disjunction (or) are treated with no restrictions). In addition, we also address the issue of of the general ES-DBMS coupling mechanism. The full implementation, together with examples, is given in [Vassiliou et al 1983].

We now describe the overall mechanism (Figure 1) that allows for deferred database calls. Linking Prolog and meta-Prolog is accomplished with the introduction of a binary predicate called "meta". The reflect program produces "meta" predicates from a set of Prolog statements. It also groups the meta predicates by providing a unique name for the program. The reflect function is invoked once before the start of a session. In the heart of the mechanism is the proof of metaevaluate(Pgm,Meta_Goals,Ctrl,New_Goals) in the meta-language, where Pgm and Meta_Goals are the meta-language names of object assumptions, goals, respectively. Ctrl is a parameter which specifies either a bound in the proof of metaevaluate or an action to be taken later (e.g. optimization, translation to relational algebra or SQL). The result, New_Goals, is a series of Prolog predicates in a deferred evaluation state (a series of dbcalls and other non-evaluable predicates). The generate program is activated by metaevaluate and creates an internal (Prolog) database relation. In doing so, it uses and controls the execution of the sub-programs "optimize", "sql_translate", "sql_call", and "format_db". Optimize performs some optimization to the goals generated in metaevaluate. One optimization is the removal of redundant goals. Another optimization identifies cases where a series of DBMS queries is required (e.g., in recursion). By imposing an ordering on the goals, "optimize" makes it

possible that a query result can be used for answering the next query more efficiently. Sql-translate generates SQL queries from optimized goals. First, the procedure identifies the database relations involved from the predicate names in Optimized_Goals and its knowledge about the database schema (SQL's FROM clause). Next, it identifies target attributes (SQL's SELECT clause) from the universally quantified variables of the original goals, and ignores all other variables in the goals unless they serve as join fields (e.g., rel1.field1 = rel2.field2). All constant values are translated to restrictions on field values (e,g., fieldname = constant). Finally, sql-call invokes the existing DBMS by sending an SQL query, with the result formatted to an internal Prolog database by format-database.

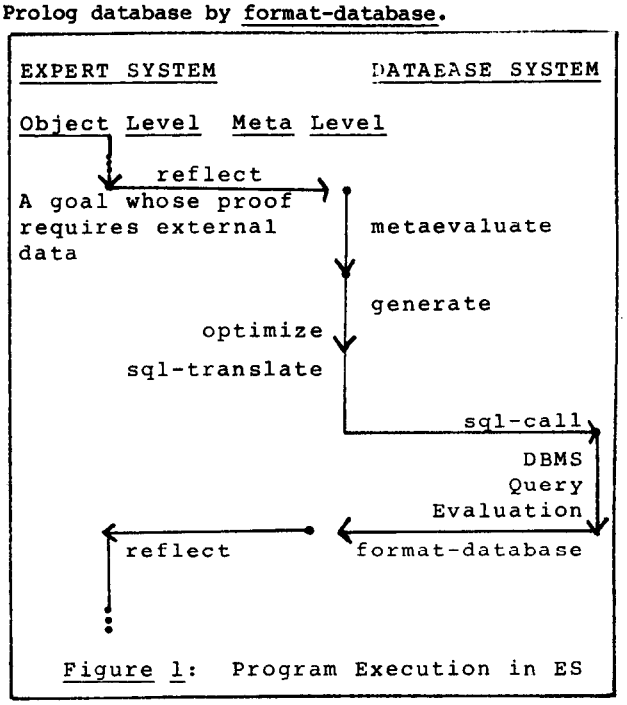

Figure 1: Program Execution in ES

As an illustration of the process outlined above, assume a statement in a hypothetical expert system involving "good_supplier", which is defined with the Prolog statements (supply, supplier are the relations in the database):

```
good_supplier(Sno,Pno) :- north_european(Sno),
                          major_supplier(Sno,Pno).
north_european(Sno) :-
                    or(supplier(Sno,N,St,london),
                       supplier(Sno,N,St,paris)).
major_supplier(Sno,Pno) :- supply(Sno,Pno,Qty),
                           greater(Qty, 300).
```

Since an instantiation of "good_supplier" would require database calls, we use "metaevaluate" as the first subgoal:

```
  metaevaluate(pr1,
       [good_supplier(v_Sno,v_Pno)],5, NG)
```

An intermediate result from "metaevaluate" is :

```
NG = [or(dbcall(supplier,v_Sno,v_N,v_St,london),
         dbcall(supplier,v_Sno,v_N,v_St,paris)),
      dbcall(supply,v_Sno,v_Pno,v_Qty),
      dbcall(greater,v_Qty,300)]
```

Given the specific value for the Control parameter of "metaevaluate", the program "generate" is invoked. First, its sub-programs "optimize" and "sql_translate" transforms the new goals to the SQL_query:

```
SELECT  Sno, Pno
FROM    supplier, supply
WHERE   ((supplier.City = 'london') OR
          (supplier.City = 'paris'))
        AND (supply.Qty > 300)
        AND (supply.Sno = supplier.Sno);
```

The call is made to the external DBMS (sql_call), and the answer retrieved from Answer_location (format_db). Finally, a new internal database is generated with the description:

good_supplier(sno, pno)

After this process, the next statements in the expert system clause can use "good_supplier". No additional database calls are needed.

5.0 CONCLUDING REMARKS - FURTHER RESEARCH

In this paper we have outlined a number of strategies for establishing a cooperative communication between the deductive and data components of an expert system. We have shown that the spectrum of possible mechanisms to link these two components is effectively a continuum from, at one extreme, a single logic-based system that implements both components, to, at the other extreme, two completely separate systems with a strong channel of communication.

Finally, a research question of particular interest to the database community is the use of an ES as a DBMS "interface" [Jarke and Vassiliou 1983]. Could an ES be used as a sophisticated access mechanism (e.g. high-level optimization, understanding of user intent)? How could ES technology be used for integrity checking and improved locking mechanisms? Such a "DBMS-expert" may require a tight-coupling mechanism like the one described in this paper.

1. Bowen, K.A., and Kowalski, R.A., "Amalgamating Language and Metalanguage in Logic Programming", Logic Programming, K. Clark and S.A. Tarnlund, eds., Academic Press, 1982.

2. Gallaire, H., and Minker, J., Logic and Databases, Plenum, 1978.

3. Gallaire, H., Minker, J., and Nicolas, J., Advances in Database Theory, Vol.1, Plenum Press, 1981.

4. Gallaire, H., Minker, J., and Nicolas, J., Proc. Workshop of Logical Bases for Databases, Toulouse, December 1982.

5. Jarke, M., and Vassiliou, Y., "Coupling Expert Systems with Database Management Systems", NYU Symposium on Artificial Intelligence Applications for Business, New York, 1983.

6. Vassiliou, Y., Clifford, J., and Jarke, M., "How does an Expert System get its Data?", CAIS Working Paper Series, no.50, GBA,83-26(CR), New York University, 1983.