ACCESS TO SPECIFIC DECLARATIVE KNOWLEDGE BY EXPERT SYSTEMS: THE IMPACT OF LOGIC PROGRAMMING[1]

by

Yannis Vassiliou
James Clifford
and
Matthias Jarke

Information Systems Area
Graduate School of Business Administration
New York University
90 Trinity Place
New York, N.Y. 10006

March 1983

Center for Research on Information Systems
Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #50
GBA #83-26 (CR)

[1]This paper appears in Decision Support Systems 1 (1985) 123-141.

Abstract

As part of the operation of an Expert System, a deductive component accesses a database of facts to help simulate the behavior of a human expert in a particular problem domain. The nature of this access is examined, and four access strategies are identified. Features of each of these strategies are addressed within the framework of a Logic-based deductive component and the relational model of data.

1.0 INTRODUCTION

Decision Support Systems (DSS) require the simultaneous management of data, models, and dialogues [Sprague and Carlson 1982]. DSS research has placed particular emphasis on providing consistent user views of models and data [Bonczek et al 1982], and on supporting access to databases by decision models [Donovan 1976]. The emergence of practically usable Artificial Intelligence (AI) techniques over the last few years impacts these problems in at least two ways. On one hand, the interaction between DSS components, and between DSS and user can be handled more smoothly using AI methods for model management [Bonczek et al 1983; Elam and Henderson 1983] and user interfaces [Blanning 1983]. On the other hand, the addition of knowledge-based decision models, in particular expert systems, to the model base of a DSS presents new challenges for DSS implementation. It is this latter problem that is the focus of this paper.

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. Expert Systems differ from exact or heuristic optimization procedures, as used in conventional DSS, in that they mostly base their recommendations on informal and qualitative decision rules acquired from a human expert, rather than on a complete mathematical formalization of a decision problem [Clifford et al 1983].

In the heart of an ES lies the program that "reasons" and makes deductions, the inference engine. To reason, both general knowledge (rules), e.g. if a person works for a company then he/she gets employee benefits, and specific declarative knowledge (data), e.g. John works for NYU, is needed. The knowledge is usually represented in such formalisms as frames [Minsky 1975], conceptual dependency graphs [Schank 1975], production rules [Waterman and Hayes-Roth 1979], semantic networks [Brachman 1979], or in standard first-order logic. Many of these formalisms can represent both general and specific knowledge. Current Expert Systems differ in sophistication, conceptual complexity, and computational complexity; for instance, the knowledge base may or may not include such concepts as causality, intent, physical principles, and simple empirical associations.

A scenario for consulting an ES using production rules for knowledge representation starts with a presentation of a goal or desired conclusion. The inference engine chains through (forward or backward) a set of production rules to link the conclusion with the assumptions, or known "facts". The system's conclusion is then presented to the user, who can ask for an explanation of the "chain of reasoning" used to arrive to the given result.

This paper is primarily concerned with the organization and access of simple declarative knowledge in the knowledge base of ESs. To organize these data, two dimensions are considered: variety and population. For instance, in a logic-based representation, "variety" refers to the number of different predicates required, and "population" to the number of instances of these predicates.

In early ESs, which are mostly prototypes and are characterized by a large variety and a small population of specific knowledge, the inefficiency of data handling is not a critical issue. Therefore, with very few exceptions, little attention has been given in ES design to the handling of very large populations. The mechanism to retrieve the specific facts does not reach the sophistication and performance of database management systems (DBMS), systems that deal effectively with large volumes of data [Date 1982].

This paper investigates the technical issues of enhancing Expert Systems with database management facilities. The motivation for such enhancements is provided by the rapid advent of ES and the increasingly promising impact that they will have in the business applications sector - an environment that often implies the presence of large databases, usually under the control of a DBMS.

In Section 2, four database access strategies are identified and developed in stages. Tools developed at an earlier stage are often necessary in each subsequent stage. The framework is illustrated with the use of first-order logic and relational database management. In particular, the logic programming language Prolog [Clocksin and Mellish 1981] is presented in Section 3, and its uses as a programming language, a relational database system, and an ES deductive component, are outlined. The way Prolog fits into the proposed framework of access strategies is the topic of Sections 4 and 5. The last section presents a summary and some problems for further research.

2.0 DATABASE ACCESS STRATEGIES BY EXPERT SYSTEMS

Two general architectures are envisioned for the combination of the deductive and the database access components of an expert system. These two components can either be integrated into one system (the

ES), or be independent systems with a defined protocol for communication [Vassiliou, Clifford, and Jarke 1983].

Depending on the level of sophistication for the database access facility, integration suggests two distinct access strategies: elementary database access, and generalized database management. A major distinguishing characteristic between these general strategies is their respective ability to deal with secondary storage management, and therefore, their capability to deal with large populations of specific facts.

With the advent of ESs in the business environment, a strong motivation for coupling an ES with an external DBMS has emerged [Jarke and Vassiliou 1983]. The investment of an enterprise in two different types of systems, both intended to assist decision making and smooth the flow of operations, is greatly justified if the two systems are able to communicate effectively. Thus, the large amounts of data managed by a DBMS can be accessed by the ES in the reasoning process. Moreover, the ES can offer an intelligent interface to a DBMS (in addition to query languages, report generators, etc.). Depending on the nature of communication between the two independent systems (ES and DBMS), two more access strategies are identified: loose, and tight coupling.

Figure 1 illustrates a natural sequence in the development of access strategies. An overview is given in the rest of this section.

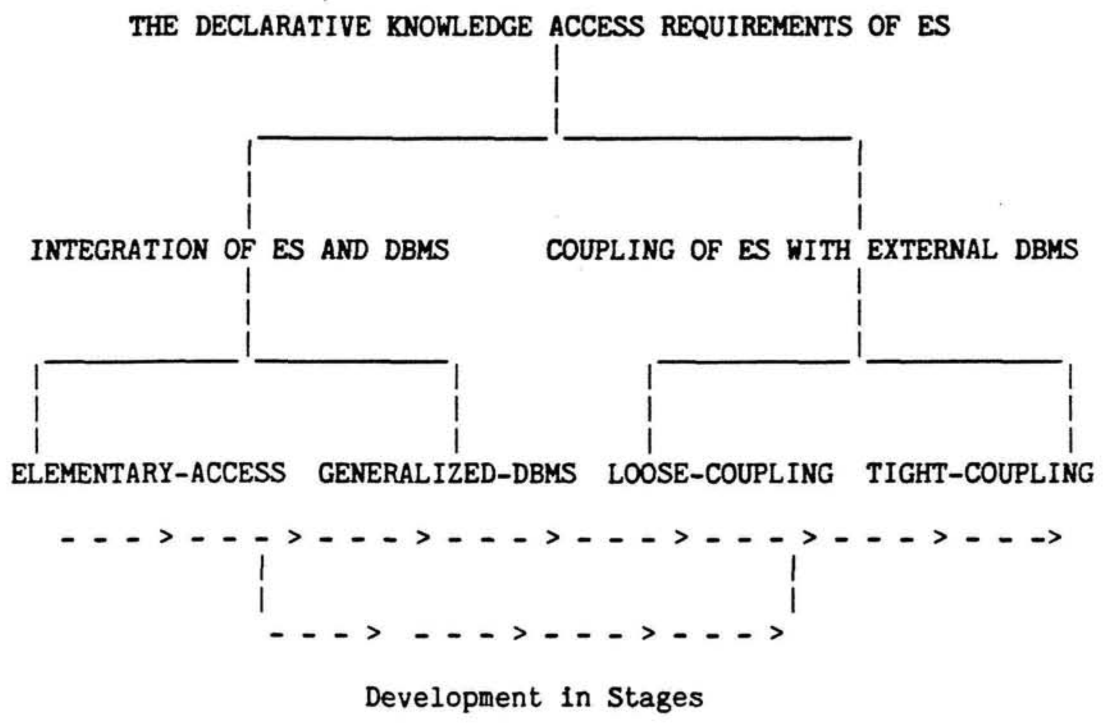
THE DECLARATIVE KNOWLEDGE ACCESS REQUIREMENTS OF ES
INTEGRATION OF ES AND DBMS
COUPLING OF ES WITH EXTERNAL DBMS
ELEMENTARY-ACCESS
GENERALIZED-DBMS
LOOSE-COUPLING
TIGHT-COUPLING
Development in Stages

Figure 1: STRATEGIES FOR DATABASE ACCESS BY AN ES

2.1 Elementary Database Access Within An Expert System - Strategy 1

On the simplest level, the whole population of specific declarative knowledge can be represented directly in the knowledge base formalism provided by the Expert System. Mechanisms such as semantic networks and frames, data structures where all knowledge about an object is collected together, are commonly used in ESs. Furthermore, several languages have been developed to access and manipulate frames and semantic networks, e.g. NETL, KRL, and KLONE [Nau 1983].

The first strategy in the manipulation of these data structures is based on the assumption that during the ES operation they reside in main storage. This simplifies the development of access routines, but presents an obvious limitation on the size of the declarative knowledge population.

2.2 Generalized DBMS Within An Expert System - Strategy 2

As the domains to which ES technology is applied increase, a very large population of specific knowledge is often required. Such Expert Systems have elementary database management facilities as separate processes [Nau 1983]. The minimum requirements for this access strategy are secondary storage management and indexing schemes. This seems to be the norm for current ESs, even though not all such systems exhibit the same level of sophistication.

Moving a step further, a generalized DBMS may be implemented as a sub-process of the ES. The quest for "generalized" database operations in the ES, rather than application-specific database access, may not be cost-effective in many cases. A case where generalization is effectively justified, is when the ES uses it as stepping stone to one of the coupling mechanism described in Sections 2.3 and 2.4.

The major limitation in this stage is that often an existing very large database may be needed in the Expert System application. Assuming a generalized commercial DBMS managing this database, it may be prohibitively costly to maintain a separate copy of the whole database for the ES. As an example, [Olson and Ellis 1982] report experiences with an Expert System used to determine problems with oil wells where data from a very large IMS database was needed but could not be made available.

2.3 Loose Coupling Of The ES With An External DBMS - Strategy 3

Conceptually the simplest solution to the problem of using existing databases managed by an external DBMS is to extract a snapshot of the required data from the DBMS when the ES begins to work on a set of related problems. This portion of the database is stored in the internal database of the ES as described in the previous section. For this scenario to work, the following mechanisms are required:

1. Link to a DBMS with unload facilities;
2. Automatic generation of an ES database from the extracted database;
3. An "intelligence" mechanism to know in advance which portion of the database is required for extraction.

Such a strategy presents several practical advantages and could be used in combination with any of the two previous access strategies. However, loose coupling is not suitable if the portion of the database to be extracted is not known in advance. This refers to the third of the required mechanisms which is clearly the hardest to automate. When this mechanism is not automated, the decisions have to be made "statically" with human intervention. Furthermore, loose coupling is inefficient when different portions of the database are needed for the Expert System at different times. This leads to the need for the final stage: tight coupling of the ES with a DBMS.

2.4 Tight Coupling Of The ES With An External DBMS - Strategy 4

For this access strategy it is assumed that a very large database exists under a generalized DBMS, and the ES needs to consult this database at certain points during its operation. Under this script, an online communication channel between the ES and the DBMS is required. Queries can be generated and transmitted to the DBMS dynamically, and answers can be received and transformed into the internal 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 will assume the redirection of all ES queries to the DBMS. Any such approach is bound to face at least two major difficulties:

A.- Number of Database Calls

Since the ES normally operates with one piece of information at a time (record), a large number of calls to a database may be required for each ES goal. Assuming that the coupling is made at the query language level, rather than an internal DBMS level, such a large number of DBMS calls will result in unacceptable system performance. The number of calls at the query language level could be reduced, if these calls result to a collection (set) of records.

B.- Complexity of Database Calls

Database languages usually have limited coverage. For instance, the majority of query languages do not support recursion. For reasons of transportability and simplicity, it may not be desired to include in the coupling mechanism the "embedding" programming language (e.g. PL/1, COBOL), a language that would solve the discrepancies in power between the ES and DBMS representations and languages.

Therefore, to attain tight-coupling, particular care has to be given to global optimization in using the communication channel, and to the representation and language translation problems. To the authors' knowledge, tight-coupling to an existing DBMS has not yet been implemented in actual systems. It appears that the impact of logic programming and the commercialization of relational database systems will have a profound effect for tight-coupling in future system architectures. Prolog is currently the most widely known programming language; it has been announced as the basis of the 5th Generation Computer Project in Japan [Feigenbaum and McCorduck 1983]. It is becoming clear that logic-based programming languages like Prolog will be highly influential in the ESs of tomorrow. In the remaining sections of this paper, Prolog and a research effort to develop a formalism for coupling a Prolog-based Expert System with a relational DBMS are described.

3.0 A PROLOG INTRODUCTION

3.1 Prolog As A Programming Language

Prolog is a programming language based on a subset of first-order logic, the Horn-clauses. Roughly, this amounts to dropping disjunction from logical consequents, and talking only about definite antecedent-consequent relationships.

Statements.- There are three basic statements in Prolog (the symbol <- denotes implication, and the symbol & denotes the logical AND):

<- P.	means	P is a goal
A.	means	A is an assertion
P <- Q & R & S.	means	Q and R and S imply P

A clause has both a declarative and a procedural interpretation. Thus,

P <- Q & R & S

can be read declaratively:

P is true if Q and R and S are true

or, procedurally (similar to "stepwise refinement" [Wirth 1971]):

To satisfy P first satisfy Q and R and S.

Programs.- A Prolog program is a sequence of clauses whose variables are considered to be universally quantified. Logic predicates are represented with Prolog programs, and since more than one clause may be needed to define a predicate (goal), there is a corresponding AND/OR graph for each predicate. The execution of a program involves a depth-first search with backtracking on these graphs, and uses the unification process based on the resolution principle [Robinson 1965].

As an example of a Prolog program, consider the appending of two lists to form a third. In this Prolog system notation, predicate names are in upper-case, variables are in lower-case, character strings that start with upper-case denote denote constant values, brackets enclose lists, [] is the empty list, and the operator "|" separates the first element of the list from the rest.

```
APPEND([], y, y).
APPEND([x|y], z, [x|w])  <-  APPEND(y, z, w).
```

Clause one asserts that appending the empty list to any list leaves the list unchanged (stopping the recursion). Clause two states that <u>if</u> y appended to z results in w, <u>then</u> a list with first element x and remainder y, when appended to z, results in a list with first element x and remainder w.

Given the goal: "<- APPEND([A], [B,C], new)", Prolog tries to instantiate the variable new to whatever value makes the predicate true. The first clause cannot be used the first time around ([A] is not []). If the instantiation x=[A], y=[], z=[B,C], and new=[A|w] is made then the second clause applies. This requires the evaluation of the right-hand side goal: "<- APPEND([], [B,C], w)". For this goal the first clause applies, w is instantiated to [B,C] and through recursion, new is instantiated to [A,B,C].

An important characteristic of Prolog programs is that there need be no distinction between input and output parameters. Thus, one can also ask for the combination of lists that result in a specific list when appended to each other:

```
<- APPEND(x, y, [A,B,C]).
```

3.2 <u>Prolog And Relational Database Management</u>

To clarify Prolog's approach to relational database management, a short description of two different views of relational databases is required. The traditional view of relational databases [Codd 1970] is that of a collection of tables. Formally, a relational database is a relational structure [Kowalski 1981]. Queries on the database are expressed in languages having the power of first-order logic and are evaluated in the relational structure (evaluational approach). In contrast, a proof-theoretic view would look at a database as a collection of sentences - a theory. A database query is answered by proving it to be a logical consequence of the theory (non-evaluational approach). This distinction is described in detail in [Minker and Gallaire 1978]. Essentially, it amounts to the difference between theories and their interpretations.

[Kowalski 1981] shows that, under certain conditions, this distinction is irrelevant. In particular, it can be shown that all queries in first-order logic evaluate to the same value whether the relational database is interpreted as a structure or as a logic database, provided that:

1. There are finitely many variable-free atoms;
2. The database is described by Horn clauses;
3. The axioms of equality and domain closure are present; and,
4. Negation is interpreted as finite failure.

Relational databases can therefore be represented directly in Prolog as a listing of all instantiated predicates corresponding to relation tuples. For instance, consider the database-oriented view of the world of Suppliers-and-Parts [Date 1982].
The relations (scheme) are:

```
SUPPLIER(sno, sname, status, city)
PART(pno, pname, color, city)
SUPPLY(sno, pno, qty)
```

An instance of the database would be represented in Prolog as:

```
SUPPLIER(S1,SMITH,20,LONDON).
SUPPLIER(S2,JONES,10,PARIS).
...
PART(P1,NUT,RED,12,LONDON).
PART(P3,SCREW,BLUE,ROME).
...
SUPPLY(S1,P1,300).
SUPPLY(S2,P2,200).
...
```

In addition to database representation, Prolog can be used directly as a database query language.

Each query may have the format [Kowalski 1981]:

```
<- QUERY(<target-variables>).
QUERY(<target-variables>) <- GOAL_A & GOAL_B &... & GOAL_N
```

where <target-variables> is a list of variables (corresponding to attribute names). The interpretation is that the user wants to retrieve all instantiations satisfying the goal statements. Thus, <target-variables> corresponds to the target list in conventional query languages. For instance, consider the Prolog statements:

```
/* For all suppliers,
   list the supplier number and the city they live in */

<- LIVES(sno, city).   /* where */
LIVES(sno, city) <- SUPPLIER(sno, any_sname, any_status, city).

/* List the supplier number
   for those suppliers who supply more than one parts */

<- SUPPLIES_MANY(sno).    /* where */
SUPPLIES_MANY(sno) <- SUPPLY(sno,p1,q1) &
                      SUPPLY(sno,p2,q2) & NOT(p1=p2).

/* List the supplier number
   for those suppliers who do not supply more than one parts,
   and live either in London or in Paris */

<- SPECIAL_SUPPLIER(sno).    /* where */
SPECIAL_SUPPLIER(sno) <- NOT(SUPPLIES_MANY(sno)) &
                     OR(LIVES(sno, LONDON), LIVES(sno, PARIS)).
```

These examples can be used to illustrate both the query capabilities of the Prolog formalism, and the powerful mechanism for "generalized" views. Such views differ from the traditional DBMS views in that with the use of variables they can accept parameters. In essence, views allow for a flexible data representation. [Kowalski 1981] also details the use of Prolog for integrity constraints, database updates and historical databases.

3.3 Prolog And Expert Systems

A knowledge base can be represented in first-order logic if the formulas are suitably interpreted. Therefore, Prolog may be used for the knowledge representation. Furthermore, Prolog has the advantage that it already has a very powerful inference engine in place (automatic theorem prover). The unification algorithm used in Prolog is more general than a simple pattern matching algorithm (common in production rule-based systems [Nau 1983]).

As an illustration, a small "toy" Expert System in Prolog is presented. The area of interest is the well-known world of suppliers, parts, and supplies. In this simple example, the "expert" is supposed to recommend where to order by applying the following rules:

1. Order only from suppliers who have supplied the same part and all its subparts before.

2. Suppliers from southern Europe are usually cheaper than those from northern Europe. No suppliers outside Europe should be considered.

3. Display all possible choices within the "optimal" category.

A Prolog expert for this job would obviously need a representation both of the above rules and of the data they require, such as location and previous supply for each part, and the classification of locations into northern and southern Europe. A sketch of a possible Prolog knowledge base follows.

First, the database of specific facts is presented. It is noted that only binary or unary relations are used in this example, but this is not limiting in that there is a simple way to move between binary representations and ternary representations [Kowalski 1979].

```
/* Simple declarative facts (database) */

SUPPLIER(SMITH).
SUPPLIER(JONES).
SUPPLIER(BRAND).

PART(NUT).
PART(WIDGET).
PART(GIZMO).
PART(SCREW).
PART(GADGET).
PART(THINGUM).
PART(SUPERTHINGUM).

SUBPART(NUT, WIDGET).
SUBPART(SCREW, GADGET).
SUBPART(GADGET, GIZMO).
SUBPART(THINGUM, SUPERTHINGUM).

HAS_SUPPLIED(SMITH, NUT).
HAS_SUPPLIED(SMITH, WIDGET).
HAS_SUPPLIED(SMITH, GIZMO).
HAS_SUPPLIED(SMITH, THINGUM).
HAS_SUPPLIED(JONES, SCREW).
HAS_SUPPLIED(JONES, NUT).
HAS_SUPPLIED(JONES, WIDGET).
HAS_SUPPLIED(JONES, GADGET).
HAS_SUPPLIED(JONES, GIZMO).
HAS_SUPPLIED(JONES, SUPERTHINGUM).
HAS_SUPPLIED(BRAND, SCREW).
```

```
LIVES(SMITH, ROME).
LIVES(JONES, LONDON).
LIVES(BRAND, OSLO).

NORTH(LONDON).
NORTH(OSLO).
SOUTH(ROME).
SOUTH(ATHENS).
```

Second, the part of the knowledge base containing the general rules is presented.

```
/* General Rules */

SUGGEST__ORDER(supplier, part) <-
    GOOD__AND__CHEAP(supplier, part).

/* if no good and cheap suppliers exist, then: */

SUGGEST__ORDER(supplier, part) <-
    NOT(GOOD__AND__CHEAP(any__supplier,part)) &
    NORTH__EUROPEAN(supplier) &
    POTENTIAL__SUPPLIER(supplier,part).

GOOD__AND__CHEAP(supplier,part) <-
    POTENTIAL__SUPPLIER(supplier,part) &
    SOUTH__EUROPEAN(supplier).

POTENTIAL__SUPPLIER(supplier,part) <-
    SUPPLIER(supplier) &
    PART(part) &
    NOT(MISSING__SUBPART(supplier,part)).

MISSING__SUBPART(supplier, part) <-
    NOT(HAS__SUPPLIED(supplier, part)).

MISSING__SUBPART(supplier, part) <-
    SUBPART(any__part, part) &
    MISSING__SUBPART(supplier, any__part).

NORTH__EUROPEAN(supplier) <- LIVES(supplier, city), NORTH(city).
SOUTH__EUROPEAN(supplier) <- LIVES(supplier, city), SOUTH(city).
```

To illustrate the use of the above simple Expert System, some examples are given below. It is noted that the user places a goal (desired conclusion) at the "| ?-" prompt and the system returns with a "no" answer if the goal can not be proven, and with an assignment of values to the variables used otherwise. If the goal has more solutions (other variable assignments exist), they are obtained with the typing of a semi-colon until the answer "no" is returned.

```
/*        Example Execution             */

| ?- SUGGEST__ORDER(x,WIDGET).
x = SMITH ;
no

| ?- SUGGEST__ORDER(x,GIZMO).
x = JONES ;
no

| ?- SUGGEST__ORDER(x,SCREW).
x = JONES ;
x = BRAND ;
no
```

4.0 PROLOG AS THE MECHANISM FOR INTEGRATION

4.1 Prolog And Access Strategy 1

As outlined in the previous section, elementary database management corresponds to a direct use of Prolog. The limitations in this strategy are:

(a) Large Databases

Executing Prolog programs in the manner described above requires that the assertions representing the database (instantiated predicates) be in main storage. Even when the database can fit in main storage, and despite the fact that Prolog implementations are very efficient, there are limitations in secondary indexing. For instance, the Prolog DEC-10 compiler, which is considered to be the most efficient implementation, has only one index for the internal database. In short, both external and internal data management are needed for large databases.

(b) Generality

Simple-minded use of Prolog can only offer elementary data management facilities. For instance, there is no data dictionary, no database schema, and no generalized set-oriented relational operations. It may be argued that lack of generality is a matter of convenience rather than an issue of substance. On the other hand, it is closely related to the first limitation, and in the next stage a uniform mechanism to deal with both is used.

4.2 A Generalized Database System In Prolog - Access Strategy 2

Generalized DBMSs gain much of their power by abstracting from specific query predicates to generalized retrieval mechanisms such as the set-oriented relational algebra operators or the SQL nesting mechanisms. One advantage of using these abstractions in Prolog is that they allow generalized selection of predicates instead of forcing the database programmers to define such predicates separately for each class of data.

Therefore, we could take a further step towards integrating the deductive capabilities of Prolog with database management capabilities by implementing a general purpose DBMS directly in Prolog. This can be done quite easily, and provides a means of adding flexible and general data access mechanisms to the inference engine.

In order to effect this stage in ES enhancement with data management facilities, the first requirement is the definition of an internal representation of a relational database. The following Prolog version is a simple and direct strategy for this purpose.

```
DBSCHEMA = [ DB-NAME, [relations], [constraints] ]

REL-i = [ REL-NAME, [scheme], [domains] ]
                                     /* for each REL-i,  1<=i<=n  */
SCHEME = [a-1, ..., a-k]             /* a list of attributes      */
DOMAINS = [d-1, ..., d-k]            /* a list of domains  with   */
                                     /* DOM(a-i) = d-i            */
CONSTRAINTS = [ [list-of-fds],       /* functional dependencies   */
                [list-of-vds],       /* value dependencies        */
                [list-of-sds] ]      /* subset dependencies       */

FD = [ REL-NAME, LHS, RHS ]          /* corresponds:  LHS --> RHS */
LHS = [ a-i1, a-i2, a-il ]           /* a list of attributes      */
RHS = [ b-j1, b-j2, b-jm ]           /* a list of attributes      */

VD =  [ REL-NAME, ATTR-NAME, LOWER-BOUND, UPPER-BOUND ]
          /*  The values for ATTR-NAME must be within the bounds  */

SD =  [ REL-NAME1, ATTR-NAME1, REL-NAME2, ATTR-NAME2 ]
          /*  The values in ATTR-NAME1 must also be in ATTR-NAME2 */

DBINSTANCE = [ DB-NAME, [relation-instances] ]
RELATION-INSTANCE = [ REL-NAME, [tuples] ]
TUPLE-i = [V-1, ..., V-k]            /* V-i is in d-i,   1<=i<=k  */
```

This strategy provides a straightforward implementation of the structure of a relational database as seen by the user (in this case the ES). The Suppliers-and-Parts database of Section 3.2, represented in this format, is given in Appendix 1.

Given such a representation scheme, one can define any number of generalized operations to provide the facilities of a DBMS. The feasibility of this has been demonstrated in [Kunifuji and Yokota 1982]. The basis for the implementation is the predicate "SETOF" built into most Prolog versions. SETOF(x,c,r), returns in the set r all such elements of x that satisfy condition c. For instance, the projection of a relation R on scheme (x1,x2,...,xn) onto the attributes xj1,xj2,...,xjk will have the form:

```
SETOF((xj1,xj2,...,xjk), ( (xi1,xi2,...,xim)^R(x1,x2,..., xn) ), s)
```

where m=n-k, and (xi1,xi2,...,xim)^ denotes the existential quantification of these variables.

As a specific example, the projection of a relation R on scheme (a,b,c) onto the attribute c is "computed" by the following Prolog program:

```
| ?- SETOF(c, ( (a,b)^R(a,b,c) ), s).
```

Note, however, that this view of the projection operator requires the user to know the entire scheme of each relation and the order of the attributes in the scheme; this may be too much to ask in general. The approach taken here (details are given in Appendix 1) by contrast, provides a simple way to specify projection as a generalized operator acting on any relation and set of attributes. Prolog programs map from this simpler, user-oriented view of the operations, to their implementation for the particular database and representation scheme chosen. This provides a degree of logical data independence as in the traditional levelled architecture of DBMSs [Date 1982].

Another feature provided by many DBMSs is the ability to define a "view" of the database for particular applications. These views define only that portion, or rearrangement, of the database of interest to a particular user community, effectively screening out the rest of the database from their sight. For example, users interested only in the set of suppliers without any of their attributes, could define the following view:

```
SUPPLIES(s) <- PROJECT(SUPPLY,[SNAME],result_tuples).
```

The user (typically the ES) has a choice between set-oriented and tuple-at-a-time retrieval operations. This is accomplished with the introduction of an evaluable predicate called "SIMCALL". This predicate simulates Prolog's calls of predicates corresponding to relations (i.e. returns a tuple instantiation). Thus, each call of the predicate SUPPLY, defined below, will return one tuple of the relation SUPPLY (stored in the format described in this Section).

```
SUPPLY(sno,pno,qty) <- SIMCALL(SUPPLY, [sno,pno,qty]).
```

Another issue for the implementation of a generalized DBMS within Prolog is that of efficient secondary storage management. For the latter, it is reasonable to devise a more sophisticated storage strategy (e.g., B-Trees), and perhaps to use auxiliary indexing schemes, hashing, etc. The use of some of these storage structures for implementing a simple business database in Prolog is described in [Pereira and Porto 1982], and some general Prolog data structures and accessing programs are formalized in [Tarnlund 1978].

The work reported in [Pereira and Porto 1982] demonstrates that for specific applications, indexing schemes that guide decisions about which portions of external files should be read into the internal database can be devised. Furthermore, the basic data access predicate (routine) of Prolog can be changed to direct data searches of secondary storage. Such Prolog modifications have been criticized as providing only temporary solutions, while complicating Prolog's basic structure and further divorcing the language system from formal logic.

5.0 PROLOG AND RELATIONAL DBMS AS INDEPENDENT SYSTEMS

5.1 Loose Coupling Of Prolog With A Relational DBMS

Loose coupling can easily be implemented using Prolog and a relational DBMS, under the assumption that a generalized facility as described above exists. A portion of the external database is loaded off-line (before the start of the Expert System session). A superset of the data required by the ES can actually be extracted, but the strategy may prove infeasible if the superset is too large or not known in advance (too many parameters).

5.2 Tight Coupling Of Prolog With A Relational DBMS

5.2.1 Overview -

Tight coupling refers to a dynamic use of the communication channel between the two systems. Essentially, the external database becomes an "extension" of the internal Prolog database. As in the general case, the same two basic problems must be resolved: optimization of database calls, and complexity of queries. Moreover such a coupling system requires dynamic decision-making about the location of the data needed to solve the current problem, and an effective strategy for managing internal storage.

The basic scenario for tightly coupling a Prolog-based ES with an existing relational DBMS is as follows. The user consults the ES with a problem to be solved or a decision to be made; typically this will be expressed in some sort of user-friendly language interface, but for our purposes we can assume that it is expressed directly as a Prolog predicate. Rather than evaluate this user request directly, in a tightly-coupled framework the predicate would be massaged (cf. "REFLECT," Sect. 5.2.3) into a slightly modified form whose evaluation can be delayed while various transformations are performed upon it. This process is analogous to a "pre-processing" stage in language translation. The altered predicate is then "meta-evaluated" (5.2.3). This involves analyzing the request in its Prolog formulation and dynamically determining whatever DBMS queries are required at that state in the ES execution for obtaining the solution. In our case, this involves formulating the queries in the relational language SQL [Astrahan et al 1976], performing certain optimizations upon the original SQL queries so generated, issuing the SQL queries to the DBMS along a communication channel, receiving the result of the query from the DBMS along this same channel, and re-formulating that result within the internal database structure of the ES. At that point, the "meta-evaluation" of the user's request is completed, and the Prolog inference engine can evaluate the request with the required data in its working memory.

As an example of the need for optimization, consider a naive channel use (all Prolog goals are directed to the external DBMS), and the definition of the Prolog clause:

```
SECOND_LEVEL_SUBPART(subpno,pno) <- SUBPART(subpno,pno1) &
                                    SUBPART(pno1,pno).
```

where it is assumed that "SUBPART" is a stored relation for direct (first-level) sub-relationships between parts.

In evaluating this goal, Prolog will call the leftmost "SUBPART" (redirected to the DBMS as an attempt to evaluate it) for a database tuple. subpno and pno1 will be instantiated to some constant values. Then Prolog will call the rightmost "SUBPART" with pno1 already instantiated. Such a 'follow-up' call will be made for each successful instantiation of pno. Moreover, the process is repeated for each tuple of subpart. If no second-level subpart exists in the database, all these 'follow-up' goals will fail. Thus the minimum of 2n+1 number of database calls is required, where n is the number of tuples. (The extra call is the unsuccessful attempt to instantiate the leftmost "SUBPART" when all tuples have been looked at). If there are k second-level subparts, then 2n+k+1 database calls are needed. This naive approach will thus generate a particularly inefficient version of a "nested iteration" query evaluation algorithm and will not make use of any query optimization procedures of the DBMS.

This difficulty can be overcome by collecting and jointly executing database calls rather than executing them separately whenever issued by the ES. In essence, this revised technique replaces the pure depth-first approach of Prolog by a combination of a depth-first reasoning and a breadth-first database call execution

In practice, an amalgamation of the ES language with its meta-language is used, based on the 'reflection principle' [Weyhrauch 1980]. 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), it is feasible 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, and the answers obtained are transformed to the format accepted by the ES for internal databases. Then, the ES can continue its reasoning at the object-level. Each invocation of predicates corresponding to database relations now amount to an ES internal database goal, rather than a call to an external DBMS. The theoretical basis and a detailed description of this approach are presented in Sections 5.2.2 and 5.2.3.

The second difficulty in successfully coupling a Prolog-based ES with a relational DBMS is that Prolog goals, considered as queries, can be substantially more complex than queries expressed in a database query language such as SQL. For example, most DBMS query languages are not able to handle a recursive call such as the Prolog program:

```
ANY_LEVEL_SUBPART(subpno,pno) <- SUBPART(subpno,pno).

ANY_LEVEL_SUBPART(subpno,pno) <- SUBPART(subpno,p1) &
                                 ANY_LEVEL_SUBPART(p1,pno).
```

Much research exists on the issue of recursion in databases. An important distinguishing characteristic is that the depth of recursive calls to databases is usually relatively shallow. For instance, considering again the example of subparts, recursion may only go to a few levels deep before subsequent recursive calls result in "null" answers (no tuples qualifying). This implies an immediate strategy within the framework of language amalgamation discussed above: to translate a recursive Prolog goal to SQL, generate a series of calls that can be translated directly to SQL, execute the SQL calls, and stop when recursion ends (SQL calls return null results). The major problem with this strategy is that it is not possible to know in advance how many such goals must be generated (the translation takes place in the ES). Therefore, it is not feasible to jointly execute these SQL calls. In other terms, little can be done for the translation at compile time, since the end of recursion can only be determined at execution time.

Even under these restrictions, much optimization can be done within the proposed framework. For example, results (tuple values) from initial SQL calls are used for subsequent SQL calls. Other approaches (e.g. [Henschen and Naqvi 1982]) handle recursion elegantly and in a general way at compile time using a method that replaces recursion by iteration. Since Prolog has no iterative statements, and it was not desired to use an embedded query language where iteration can be expressed in the host language, this method is infeasible in the framework proposed here.

5.2.2 The Theoretical Basis For Language Amalgamation -

In order to be able to talk about a language L, the use of a meta-language ML is required. The amalgamation of an object language with its meta-language refers to the ability to move between the two languages whenever it appears more convenient or efficient to use one rather than the other.

Suppose that a goal G is to be proven from a set of assumptions (hypotheses) A in a first-order language L. There are two ways to do this:

(a) Use the proof procedure of L.

(b) Simulate the proof procedure of L in ML as follows: Use a "reflect" relationship that names the assumptions A and goal G of L as Meta-A and Meta-G in ML. The provability of G from A is represented by the provability of the predicate "metaevaluate(Meta-A, Meta-G)" from sentences in ML.

Implementing amalgamation of L and ML requires the definition of the metaevaluate predicate and the naming relationship. In addition, it requires a link (reflection principle) between the two languages.

In this specific case, Horn clause logic (Prolog) is used as the object language. The meta-language is Prolog itself - with the restriction that all sentences are variable-free. This allows to remain in first-order logic. Thus, the naming relationship maps variables to special-form constants which simulate a variable in meta-Prolog.

This implementation of amalgamation is based on the high level description of the DEMO predicate presented in [Bowen and Kowalski 1982], and is similar to the implementation of [Kunifuji and Yokota 1982]. The work reported here extends the above approaches by providing a more general treatment of evaluable predicates. For instance, finite negation (not) and disjunction (or) are treated with no restrictions. In addition, the issue of its use in the context of the general ES-DBMS coupling mechanism is addressed.

Linking Prolog and meta-Prolog is accomplished with the introduction of a binary predicate called "META". For each Prolog clause, a corresponding instantiation of the "META" predicate exists.

The first term of "META" is a list of predicates; the head of the list is the head of the corresponding clause, and the other list elements are the terms in the body of the clause. All variables in these predicates are translated into constants with a special prefix (V_). The second term of "META" allows for the grouping of meta instantiations in a program. For example, the corresponding meta predicate for:

```
     ATHENS_SUPPLIER(sno,sname) <- SUPPLIER(sno,sname,status,ATHENS).
is
     META([ATHENS_SUPPLIER(V_sno,V_sname),
           SUPPLIER(V_sno,V_sname,V_status,ATHENS)], PR1).
```

where PR1 is the name of the program (group of "META" instantiations).

Since the objective of this approach is to defer the evaluation of predicates which correspond to database relations, all such predicates are in a delayed evaluation form. In particular, these predicates are defined in Prolog as follows:

```
SUPPLY(sno,pno,qty) <- DBCALL(SUPPLY, [sno,pno,qty]).
```

using the non-evaluable binary predicate "DBCALL". Other predicates whose evaluation depends on the database values (e.g. equal, not equal) are treated in the same way.

The implementation of the predicate "METAEVALUATE" is described, together with examples of its use, in Appendix 2. Only a high-level description is given here.

Given a set of assumptions A and a set of goals G to be proven in the object language, prove the meta-Prolog predicate:

```
METAEVALUATE(assumptions, meta_goals, control, new_goals).
```

in the meta-language, where "assumptions" is the collection of the original assumptions A in the meta-language, and meta_goals is the meta-language name of the goals G. Control 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).

5.2.3 The Mechanism For Tight-Coupling. -

This section describes the overall mechanism that allows for deferred database calls. The mechanism is presented pictorially in Figure 2. The use of a simple but complete example will illustrate the concepts involved.

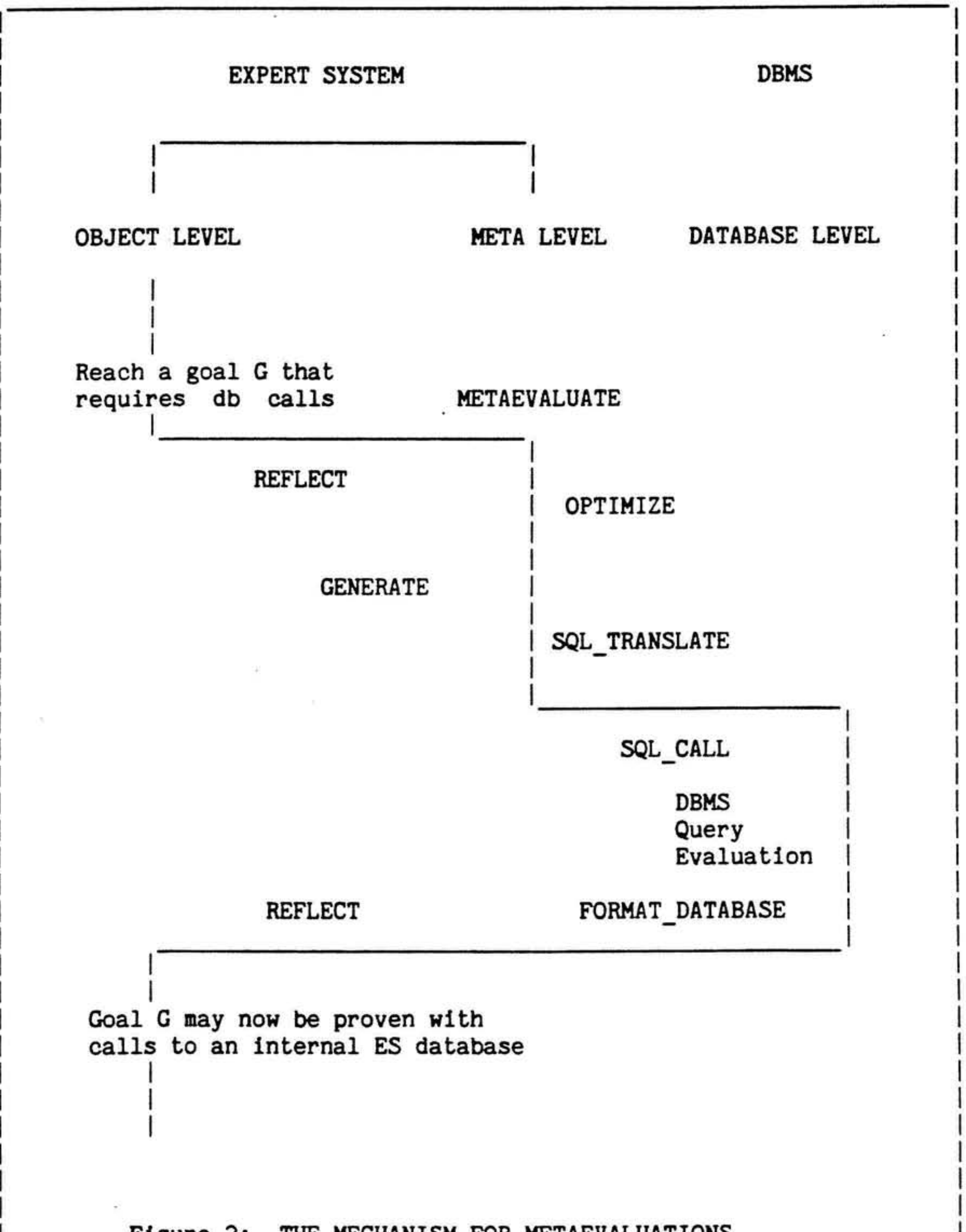

Figure 2: THE MECHANISM FOR METAEVALUATIONS

(A) REFLECT(object_assumptions, meta_assumptions, program_name). This function produces "META" predicates as described in the previous section from a set of Prolog statements. It also groups the meta predicates by providing a unique program_name. The REFLECT function is invoked once before the start of a session.

(B) METAEVALUATE(program_name, meta_goals, control, new_goals). Described in Section 5.2.2.

(B.1) GENERATE(new_goals, results). This program is activated by metaevaluate when the control parameter assumes a particular value. Given a series of new_goals, it creates an internal database relation (result). In doing so, it uses and controls the execution of the sub-programs "OPTIMIZE", "SQL_TRANSLATE", "SQL_CALL", and "FORMAT_DATABASE". Details of the implementation of these procedures will be given in a forthcoming paper.

(B.1.1) OPTIMIZE(new_goals, optimized_goals). This program 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.

(B.1.2) SQL-TRANSLATE(optimized_goals, sql_query). This 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).

(B.1.3) <u>SQL-CALL</u>(sql_query, answer_location).
This is another program activated by "GENERATE". It invokes the existing DBMS by sending an sql_query, with the result redirected to a file identified by answer_location. Each answer to a query contributes to the eventual result of "GENERATE".

(B.1.3.1) <u>FORMAT-DATABASE</u>(answer_location, internal_db).
Since the existing DBMS cannot be expected to deliver the result in the format required by Prolog, this function produces an internal Prolog sub-database from the file identified by sql_calls. Each such database contributes then to the eventual result of the calling function, GENERATE.

As an illustration of the process outlined above consider the following example. The actual Prolog execution and a more detailed description can be found in Appendix 2. Assume an ES that uses a series of informal, heuristic and exact rules, together with a large database of Suppliers-and-Parts managed by an external DBMS. The portion of this external database which is necessary for the example is assumed to contain the stored relations: SUPPLY and SUPPLIER. The hypothetical ES has the schema descriptions of the external database and several rules concerning this database. No actual tuples are stored in the internal ES database. Assume further, the ES rule (goal) "PERFORM_ORDER", which among other predicates involves the predicate (generalized view): "GOOD_BET_SUPPLIER", based on the stored relations and other generalized views.

```
     PERFORM_ORDER(sname, price, delivery)
                 <-  COLLECT_REQUIREMENTS(max_price, latest_del, pno) &
                     GOOD_BET_SUPPLIER(sno,pno) &
                     MAKE_ADJUSTMENTS(sno,new_delivery) &
                     ...
```

where "GOOD_BET_SUPPLIER" is defined as:

```
     GOOD_BET_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_BET_SUPPLIER" would require database calls, "METAEVALUATE" as the subgoal immediately preceding it is used:

```
PERFORM_ORDER(sname, price, delivery)
  <-  COLLECT_REQUIREMENTS(max_price, latest_del, pno) &
      METAEVALUATE(PR1, [GOOD_BET_SUPPLIER(V_sno,V_pno)], 5, newgoals) &
      ! &
      GOOD_BET_SUPPLIER(sno,pno) &
      MAKE_ADJUSTMENTS(sno,new_delivery) &
      ...
```

Note that the "cut" (!) subgoal assures that the metaevaluate predicate will only be executed once.

The first result from "METAEVALUATE" is (see also Appendix 2):

```
newgoals = [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" will be invoked. First, its sub-programs "optimize" and "sql_translate" will transform 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 will be made to the external DBMS (program: SQL_CALL), and the answer will be retrieved from answer_location (program: FORMAT_DATABASE). Finally, a new internal database will be generated with the description:

GOOD_BET_SUPPLIER(sno, pno)

After this process, the next statements in the Expert System clause can use "GOOD_BET_SUPPLIER" in the usual Prolog way. No additional external database calls are needed.

In essence, instead of calling the DBMS each time a tuple is needed, all "qualifying" tuples are brought into the internal database. It should be noted that the above strategy is similar to the "query modification" algorithm [Stonebraker 1975] used in some commercial DBMSs for view processing. Possibly, the single most important advantage in using the theorem prover for query modification is that the whole mechanism is integrated smoothly and naturally into an ES implementation as a generalized tool.

6.0 CONCLUDING REMARKS - FURTHER RESEARCH

In this paper a number of strategies for establishing a cooperative communication between the deductive and data components of an Expert System were outlined. It was 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.

A number of interesting research questions are raised by the spectrum of possible mechanisms for coupling these two essential components of an Expert System. Among the questions examined are: what is a general architecture for the communication channel between these two components? how can the ES DBCALLs be translated into the query language of the DBMS? when and how should these queries be optimized? A research topic under investigation is that of internal ES database space management. How does one manage the amount of free space for storing the results of external database calls? When space has to be freed, how is the decision reached and optimized as to which portion of the internal database need be deleted? A longer range research question concerns the integration of these four access strategies into a single, meta-expert system that combines the expertise of the problem domain with expertise about these four connection types. Given a particular type of problem in the domain of the expert, this meta-expert system would decide which type of coupling is most appropriate.

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 an ES assist in the implementation of language constructs that allow one to formulate arbitrary predicates with relation variables? Such constructs may be used for integrity checking and improved locking mechanisms. A tight-coupling mechanism, like the one described in this paper, may be required by such a "DBMS-expert".

References

1. Blanning, R.W., "Natural Language Query Processing for Model Management", Communications of the ACM, forthcoming.

2. Bonczek, R.H., Holsapple, C.W., Whinston, A.B., "The Evolution from MIS to DSS: Extension of Data Management to Model Management", Decision Support Systems, Ginzberg, M.J., Reitman, W.R., Stohr, E.A. (eds.), North-Holland, 1982, pp. 61-78.

3. Bonczek, R.H., Holsapple, C.W., Whinston, A.B., "Specification of Modelling Knowledge in DSS", Processes and Tools for Decision Support, Sol, H.G. (ed.), North-Holland, 1983.

4. 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.

5. Brachman, R., "On the Epistemological Status of Semantic Networks", Associative Networks: Representation and Use of Knowledge by Computer, N.V. Findler, ed., Academic Press, 1977, pp.3-50.

6. Clifford, J., Jarke, M., and Y. Vassiliou, "A Short Introduction to Expert Systems", IEEE Database Engineering Bulletin, Volume 8, No.4, December 1983 (to appear).

7. Clocksin, W.F., and Mellish, C.S., Programming in Prolog, Springer-Verlag, 1981.

8. Codd, E.F., "A Relational Model for Large Shared Data Bases, CACM, Vol.13, No.6, June 1970, pp.377-387.

9. Date, C.J., An Introduction to Database Systems, (3rd edition), Addison-Wesley, 1982.

10. Donovan, J.J., "Database System Approach to Management Decision Support", ACM Transactions on Database Systems 1, 4 (1976), 344-369.

11. Elam, J.J., Henderson, J.C., "Knowledge Engineering Concepts for Decision Support System Design and Implementation", Information and Management 6 (1983), pp. 109-114.

12. Feigenbaum, E.A., and P. McCorduck, The Fifth Generation Artificial Intelligence and Japan's Computer Challenge To the World, Addison-Wesley, 1983.

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

14. Henschen, L., and S.Naqvi, "On Compiling Queries in Recursive First-Order Databases", Proc. Workshop on Logical Bases for Data Bases, Toulouse, December 1982.

15. Jarke, M., and Vassiliou, Y., "Coupling Expert Systems with Database Management Systems", Artificial Intelligence Applications for Business (W.Reitman, ed.), Ablex, to appear 1983.

16. Kowalski, R.A., Logic for Problem Solving, North-Holland Elsevier, New York, 1979.

17. Kowalski, R.A., "Logic as a Database Language", unpublished, July 1981.

18. Kunifuji, S., Yokota, H., "Prolog and Relational Databases for Fifth Generation Computer Systems", Proc. Workshop on Logical Bases for Data Bases, Toulouse, December 1982.

19. Minsky, M., "A Framework for Representing Knowledge", The Psychology of Computer Vision, P.H. Winston, ed., McGraw-Hill, New York, 1975, pp.211-277.

20. Nau, D., "Expert Computer Systems", Computer, February 1983, pp.63-85.

21. Olson, J.P., and Ellis, S.P., "PROBWELL - An Expert Advisor for Determining Problems with Producing Wells", IBM Scientific/Engineering Conference, Poughkeepsie, New York, November, 1982.

22. Pereira, L.M., and Porto, A., "A Prolog Implementation of a Large System on a Small Machine", Departmento de Informatica, Universidade Nova de Lisboa, 1982.

23. Robinson J.A., "A Machine Oriented Logic Based on the Resolution Principle", JACM, 1965, Vol.1, No.4, pp.23-41.

24. Schank, R.C., Conceptual Information Processing, North-Holland, New York, 1975.

25. Sprague, R.H., Carlson,E.D., Building Effective Decision Support Systems, Prentice Hall, 1982.

26. Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification", Proceedings ACM-SIGMOD Conference, 1975, pp.65-77.

27. Tarnlund, S-A., "Logical Basis for Data Bases", unpublished, 1978.

28. Travis, L., and C. Kellogg, "Deductive Power in Knowledge Management Systems: Ideas and Experiments", Proc. Workshop on Logical Bases for Data Bases, Toulouse, December 1982.

29. Vassiliou, Y., Clifford, J., Jarke, M., "How does an Expert System Get Its Data?", NYU Working Paper CRIS#50, GBA 82-26 (CR), extended abstract in Proc. 9th VLDB Conf., Florence, October 1983.

30. Waterman, D., and Hayes-Roth, F. (eds), Pattern Directed Inference Systems, Academic Press, 1979.

31. Weyhrauch, R., "Prolegomena to a Theory of Mechanical Formal Reasoning", Artificial Intelligence, Vol.13, 1980, pp.133-170.

32. Wirth, N., "Program Development by Stepwise Refinement", CACM, Vol.14, No.4, 1971, pp.221-227.

Appendix 1

```
/***************************************************************************/
/* This is the database used as the example in several sections of the   */
/* paper. It has been copied from Date, and it deals with the world       */
/* of SUPPLIERS_AND_PARTS.                                                 */
/***************************************************************************/

DBSCHEMA( SUPPLIERS_AND_PARTS,

          [ [SUPPLIER,
              [SNO,SNAME,STATUS,CITY],
              [DSNO,DSNAME,DSTATUS,DCITY] ],
            [PART,
              [PNO,PNAME,COLOR,WEIGHT,CITY],
              [DPNO,DPNAME,DCOLOR,DWEIGHT,DCITY] ],
            [SUPPLY,
              [SNO,PNO,QTY],
              [DSNO,DPNO,DQTY] ]  ],

          [   [FD, SUPPLIER, [SNO], [SNAME,STATUS,CITY] ],
              [FD, SUPPLIER, [SNAME], [SNO,STATUS,CITY] ],
              [FD, PART, [PNO], [PNAME,COLOR,WEIGHT,CITY] ],
              [FD, SUPPLY, [PNO, SNO], [QTY] ],
              [FD, SUPPLIER, [CITY], [STATUS] ],
              [FD, PART, [PNAME, COLOR], [CITY] ],
              [VD, SUPPLIER, STATUS, 10, 60 ],
              [SD, SUPPLY, [SNO], SUPPLIER, [SNO] ],
              [SD, SUPPLY, [PNO], PART, [PNO] ]  ] ).
```

The envisioned use of the a database is as follows.
A predicate "open" is used to initiate the database name.

```
        OPEN( database-name )
```

For instance, the Prolog statement

```
| ?- OPEN(SUPPLIERS_AND_PARTS).
```

will instantiate the database-name. No other mention of this name need be made in the sequel. Some small examples of possible queries on this database scheme follow.

```
| ?- RELNAME(rel).
rel = SUPPLIER

| ?- SCHEME(SUPPLIER, scheme).
scheme = [SNO,SNAME,STATUS,CITY]

| ?- SHOWKEY(SUPPLIER, key).
key = [SNO];
key = [SNAME]
```

```
| ?- SHOWSD(relation, sds).

relation = [SUPPLY, SUPPLIER]
sds = [sno, sno] ;

RELATION = [supply, part]
IDS = [pno, pno]


/* An instance of the database */


DBINSTANCE( SUPPLIERS_AND_PARTS,
            [ [ SUPPLIER,
                [ [S1,SMITH,20,LONDON],
                  [S2,JONES,10,PARIS],
                  [S3,BLAKE,30,PARIS],
                  [S4,CLARK,20,LONDON],
                  [S5,ADAMS,30,ATHENS] ] ],
              [ PART,
                [ [P1,NUT,RED,12,LONDON],
                  [P2,BOLT,GREEN,17,PARIS],
                  [P3,SCREW,BLUE,17,ROME],
                  [P4,SCREW,RED,14,LONDON],
                  [P5,CAM,BLUE,12,PARIS],
                  [P6,COG,RED,19,LONDON] ] ],
              [ SUPPLY,
                [ [S1,P1,300],
                  [S1,P2,200],
                  [S1,P3,400],
                  [S1,P4,200],
                  [S1,P5,100],
                  [S1,P6,100],
                  [S2,P1,300],
                  [S2,P2,400],
                  [S3,P2,200],
                  [S4,P2,200],
                  [S4,P4,300],
                  [S4,P5,400] ] ] ] ).
```

```
/*****************************************************************/
/* These examples illustrate a subset of a relational DBMS, built */
/* upon the representation scheme discussed in Section 4. The     */
/* variable and predicate names have been chosen so as            */
/* to make the meaning clear.                                     */
/*****************************************************************/

/* Projection                                                     */
/* General form:      PROJECT(relname,attrlist,projection)        */

| ?- PROJECT(SUPPLIER, [CITY], cities).

cities = [[LONDON],[PARIS],[PARIS],[LONDON],[ATHENS]]

/* Selection.                                                     */
/* General Form:    SELECT(relname,attrs,ops,vals,res)            */

| ?- SELECT(SUPPLIER, [CITY, STATUS], [=,>], [PARIS, 20], result).

result = [[S3,BLAKE,30,PARIS]]


/* Natural Join.                                                  */
/* General Form:       NATJOIN(relname1,relname2,result,scheme)   */

| ?- NATJOIN(SUPPLY, PART, result, scheme).

result = [[S1,P1,300,LONDON,RED,NUT,12],[S2,P1,300,LONDON,RED,NUT,12],
          [S1,P2,200,PARIS,GREEN,BOLT,17],[S2,P2,400,PARIS,GREEN,BOLT,17],
          [S3,P2,200,PARIS,GREEN,BOLT,17],[S4,P2,200,PARIS,GREEN,BOLT,17],
          [S1,P3,400,ROME,BLUE,SCREW,17],[S1,P4,200,LONDON,RED,SCREW,14],
          [S4,P4,300,LONDON,RED,SCREW,14],[S1,P5,100,PARIS,BLUE,CAM,12],
          [S4,P5,400,PARIS,BLUE,CAM,12]],[S1,P6,100,LONDON,RED,COG,19]],

scheme = [SNO,PNO,QTY,CITY,COLOR,PNAME,WEIGHT] ;

/**************************************************************/
/* The following statement simulates the tuple-at-time        */
/* treatment of Prolog for relational databases defined as    */
/* predicates with the form:                                  */
/*           supply(s1,p1,100)                                */
/*           ...                                              */
/**************************************************************/

SUPPLY(sno,pno,qty) <- SIMCALL(SUPPLY, [sno,pno,qty]).

| ?- SUPPLY(sno, pno, qty).

pno = P1,
sno = S1,
qty = 300
```

Appendix 2

```
/* ****************************************************************** */
/*                                                                    */
/*  The predicate metaevaluate has four parameters:                   */
/*                                                                    */
/*      metaevaluate(Assumptions, Goals, Control, NewGoals)           */
/*                                                                    */
/*  Given Assumptions and a set of Goals to be proven, using Control, */
/*  return a new set of goals (NewGoals) - all in a non-evaluable form */
/*                                                                    */
/*  Algorithm:                                                        */
/*              1.- Select a Goal (first one).                        */
/*              2.- Select an appropriate Assumption (clause) with meta */
/*              3.- Rename the variables in the clauses               */
/*              4.- Match the renamed variables                       */
/*              5.- Add the body of the clause to the rest of         */
/*                  the goals producing intermediate goals            */
/*              6.- Apply the variable differences to the above goals */
/*              7.- Use metaevaluate recursively.                     */
/*                                                                    */
/*      Recursion stops when:                                         */
/*              - no goals exist, or                                  */
/*              - all remaining goals are DBCALLs, or                 */
/*              - the arguments of "or" and "not" are all DBCALLs     */
/*                                                                    */
/* ****************************************************************** */

METAEVALUATE(_,goal,control,goal) <- STOPEVALUATING(goal) & !.
METAEVALUATE(meta,[goal|rest],control,newgoals) <-
        META(clause,meta) &
        RENAMEVARS(clause,[goal|rest],[car|cons]) &
        MATCH(goal,car,diff) &
        ADD(meta,cons,rest,intergoals) &
        APPLY(intergoals,diff,othergoals) &
        METAEVALUATE(meta,othergoals,control,newgoals).

/*  Description of an external database (stored relations)      */
/*  This database will be used in the examples that follow      */

SUPPLY(sno,pno,qty) <- DBCALL(SUPPLY, [sno,pno,qty]).

SUPPLIER(sno,sname,status,city) <-
                          DBCALL(SUPPLIER, [sno,sname,status,city]).

PART(pno,pname,color,weight,city) <-
                          DBCALL(PART,[pno,pname,color,weight,city]).

SUBPART(subpno,pno) <- DBCALL(SUBPART, [subpno,pno]).
```

```
/* Some generalized views of the external database (used internally) */

     ANY_LEVEL_SUBPART(spno,pno) <- SUBPART(spno,pno).
     ANY_LEVEL_SUBPART(spno,pno) <- SUBPART(spno,p1) &
                                    ANY_LEVEL_SUBPART(p1,pno).

     SUPPLIES_MANY(sno) <- SUPPLY(sno,pno1,qty1) &
                           SUPPLY(sno,pno2,qty2) &
                           NOTEQUAL(pno1,pno2).

     SPECIAL_SUPPLIER(sno) <- NOT(SUPPLIES_MANY(sno)) &
                              NORTH_EUROPEAN(sno).

     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).

     GOOD_BET_SUPPLIER(sno,pno) <- NORTH_EUROPEAN(sno) &
                                   MAJOR_SUPPLIER(sno,pno).


/*  Meta predicate instantiations corresponding to the stored      */
/*  relations and views. Note the use of PR1 as a program name.    */

META([SUPPLIER(V_sno,V_sname,V_status,V_city),
      DBCALL(supplier, [V_sno,V_sname,V_status,V_city] )],PR1).

META([PART(V_pno,V_pname,V_color,V_weight,V_city),
      DBCALL(PART, [V_pno,V_pname,V_color,V_weight,V_city])],PR1).

META([SUPPLY(V_sno,V_pno,V_qty),
      DBCALL(SUPPLY,[V_sno,V_pno,V_qty])],PR1).

META([SUBPART(V_subpno,V_pno), DBCALL(SUBPART,[V_subpno,V_pno])],PR1).

META([ANY_LEVEL_SUBPART(V_spno,V_pno), SUBPART(V_spno,V_pno)], PR1).
META([ANY_LEVEL_SUBPART(V_spno,V_pno),
          SUBPART(V_spno,V_p1),ANY_LEVEL_SUBPART(V_p1,V_pno)], PR1).

META([SUPPLIES_MANY(V_sno),SUPPLY(V_sno,V_pno1,V_qty1),
                           SUPPLY(V_sno,V_pno2,V_qty2),
                           NOTEQUAL(V_pno1,V_pno2)],PR1).

META([SPECIAL_SUPPLIER(V_sno), NOT(SUPPLIES_MANY(V_sno)),
                               NORTH_EUROPEAN(V_sno)], PR1).

META([NORTH_EUROPEAN(V_sno), OR(SUPPLIER(V_sno,V_N,V_st,LONDON),
                                SUPPLIER(V_sno,V_N,V_st,PARIS))], PR1).

META([MAJOR_SUPPLIER(V_sno,V_pno), SUPPLY(V_sno, V_pno, V_qty),
                                   GREATER(V_qty, 300)], PR1).

META([GOOD_BET_SUPPLIER(V_sno,V_pno), NORTH_EUROPEAN(V_sno)
                                      MAJOR_SUPPLIER(V_sno,
/*  Examples of the execution of "metaevaluate". The Control   */
/*  value is 1, specifying no extra action (e.g. optimization) */

| ?- METAEVALUATE(PR1, [GOOD_BET_SUPPLIER(V_sno, GADGET)], 1, newgoals).

newgoals = [OR(DBCALL(SUPPLIER,[V_sno,V_N,V_st,LONDON]),
               DBCALL(SUPPLIER,[V_sno,V_N,V_st,PARIS])),
            DBCALL(SUPPLY,[V_sno,GADGET,V_qty]),
            DBCALL(GREATER,[V_qty,300])] ;

| ?- METAEVALUATE(PR1, [SPECIAL_SUPPLIER(V_sno)], 1, newgoals).

newgoals = [NOT(DBCALL(SUPPLY,[V_sno,V_pno1,V_qty1]),
                DBCALL(SUPPLY,[V_sno,V_pno2,V_qty2]),
                DBCALL(NOTEQUAL,[V_pno1,V_pno2])),
            OR(DBCALL(SUPPLIER,[V_sno,V_N,V_st,LONDON]),
               DBCALL(SUPPLIER,[V_sno,V_N,V_st,PARIS]))]

/* A Recursive call.  Recursion stops after three levels.    */

| ?- METAEVALUATE(PR1, [ANY_LEVEL_SUBPART(BOLTS, V_sup)], 1, newgoals).

newgoals = [DBCALL(SUBPART,[BOLTS,V_sup])] ;

newgoals = [DBCALL(SUBPART,[BOLTS,V_p1]),
            DBCALL(SUBPART,[V_p1,V_sup])] ;

newgoals = [DBCALL(SUBPART,[BOLTS,V_p1]),
            DBCALL(SUBPART,[V_p1,V_p11]),
            DBCALL(SUBPART,[V_p11,V_sup])]

| ?- METAEVALUATE(PR1, [SUPPLIES_MANY(V_who),NORTH_EUROPEAN(V_who)],1, ng).

ng = [DBCALL(SUPPLY,[V_who,V_pno1,V_qty1]),
      DBCALL(SUPPLY,[V_who,V_pno2,V_qty2]),
      DBCALL(NOTEQUAL,[V_pno1,V_pno2]),
      OR(DBCALL(SUPPLIER,[V_who,V_n,V_st,LONDON]),
         DBCALL(SUPPLIER,[V_who,V_n,V_st,PARIS]))]


| ?- METAEVALUATE(PR1, [OR(SUPPLIES_MANY(SMITH),
                           NOT(NORTH_EUROPEAN(SMITH)))],1,ng).

ng = [OR(DBCALL(SUPPLY,[SMITH,V_pno1,V_qty1]),
         DBCALL(SUPPLY,[SMITH,V_pno2,V_qty2]),
         DBCALL(NOTEQUAL,[V_pno1,V_pno2]),
         NOT(OR(DBCALL(SUPPLIER,[SMITH,V_n,V_st,LONDON]),
                DBCALL(SUPPLIER,[SMITH,V_n,V_st,PARIS]))))]
```