The Knowledge Engineering Review, Vol. 4: 1, 1989, 31–51

Interfacing a knowledge-based system to a large database

P. M. D. GRAY
Computing Science Dept., Univ. of Aberdeen, Aberdeen AB9 2UB, Scotland, UK

I. G. ARCHIBALD and K. LUNN
Shell Research Ltd, Thornton Research Centre, PO Box 1, Chester CH1 3SH, UK

Abstract

This paper describes the interfacing problem that arose in a Product Formulation expert system written in LISP that had to be interfaced to data in a relational database running on a separate mainframe computer. It surveys the different forms of coupling that are possible and emphasizes the advantages of tight navigational coupling over the more popular set-based coupling. It describes how Prolog was used to overcome the interfacing problems and to provide a customized front end to an end user, based on a navigational interface. It reviews the techniques of using Prolog and the likely obstacles, together with a look forward to databases using Frames or Objects.

1 Introduction

This report is based on experience gained during collaborative work on the PFES demonstrator project (Product Formulation Expert System) during 1985–87, under the British national Alvey research programme in IKBS. The Shell personnel were working on a Luboil expert system demonstrator the purpose of which was to propose a blend of selected components and additives for a lubricating oil, so that it should meet a given performance specification. In order to do this the system needed access to a database containing data both on basic components and also on results from tests of past formulations from various blends. The initial plans called for a coupling from the expert system, written in LISP and running on an AI workstation, to a database held on a mainframe under the RAPPORT DBMS. Meanwhile, work at Aberdeen on a related Alvey project (IKBS 073) was exploring an alternative software architecture based on Prolog. This encouraged the researchers at Shell to explore ways of using Prolog to provide access to their database, as described below.

The LUBOIL expert system

The formulation of lubricants is a complex activity requiring high technical skills, access to large quantities of experimental data, and the ability to make complex decisions, often when available information is incomplete. Lubricants are a mixture of oils and chemical additives, and they have to function for an extended period in a hostile environment, such as a car engine. Inevitably the additives influence each other in their primary functions, and they also have secondary effects which may be desirable or undesirable.

The testing of lubricants is a lengthy and expensive business. It involves placing them in an engine, running the engine under controlled conditions, analyzing the oil after the test, and dismantling the engine to ascertain effects such as wear. Thus an engine test will result in many measurements and a lubricant may require many different tests. This data is obtained at

considerable cost and is recorded in a complex database structure but the retrieval of relevant information in a form convenient to the expert system presents many problems.

The project experience, both at Shell and Aberdeen, is discussed below. It is used to categorize the various interfacing problems and their possible solutions. In particular there is the *human interfacing problem* (how can one provide informal languages which make it easy for the average technical user to get what they want without learning a lot of database theory?) and the *program interface problem* (how can one pass data of unknown and various types, possibly in very large sets, between machines running different programs?). One conclusion is that a subset of Prolog is very well suited to the interfacing task. The human interfacing problem for Luboil users was solved in Prolog in remarkably straightforward fashion by implementing a customized end-user language called TREQL (Lunn and Archibald, 1988). The program interfacing problem was solved by using Prolog's natural navigational style and dynamic type-checking approach to suck across relevant tuples of data as required, using backtracking to give the effect of lazy evaluation. Thus, although the report is meant to impart general information about the database interfacing problem to a variety of users, it will be of most interest to those considering the adoption of Prolog as an intermediate language.

Plan of Report

We start by considering the reasons why expert systems need to use databases, and the basic architectural combinations. We then address the fundamental issue of *set-based* versus *navigational* coupling, which concerns how much control the expert system should have over the database access strategy. We consider the advantages and disadvantages of navigational coupling through Prolog over set-based coupling through SQL. We show the special advantages of Prolog, both in combining data access with computation and in providing a suitable control structure through which an expert system can control the database search. We then describe in more detail the solution adopted for the Luboil demonstrator so as to illustrate the principles discussed.

The Luboil solution was to hold the data in compiled Prolog form using a workstation with many megabytes of memory. A navigational coupling was implemented between the Prolog database and LISP. Nowadays it is not even necessary to hold the data in virtual memory; the data can be on a relational database on a file server coupled to the workstation, but accessed through Prolog as if it were in main memory (Lucas, 1988; Gray and Lucas, 1988).

We go on to demonstrate the remarkable ease with which one can construct customised intelligent front-ends for databases in Prolog. We then address various performance issues and practical limitations found in current software, with a check list of things to watch out for. Finally we briefly survey some ongoing research in the USA into the use of Prolog with AI tools, Frame-based knowledge Bases, advanced compiling techniques and Object-Oriented Databases. Our conclusion is that Prolog provides the right level of abstraction for the interface between Expert System and Database. There is enough control for the system whilst keeping the Expert System structure free from implementation details of the database. In particular, we suggest that all databases ought to provide the necessary "hooks" for a Prolog interface besides providing an SQL interface.

2 Use of databases by expert systems

Early reviews of databases and expert systems appear in Gallaire (1983) and Jarke and Vassilliou (1984). The first workshop on Expert Database Systems also took place in 1984, organized by Kerschberg. The reasons for coupling the two systems can be briefly summarized:

1 The database plays the part of a user at a terminal in *ask the user*. The advantage is that it remembers precisely and, better still, it may already contain large quantities of validated data built up from a variety of sources.
2 There is a central database (e.g. in a hospital) to which multiple users need shared access; either it

is almost static but too large to be copied across into each workstation, or else it is being used by expert systems as a global location for shared update and thus for communication.

3 The amount of data to be accessed is so great that virtual memory paging schemes severely degrade performance and thus it is necessary instead to use database storage structures (such as B-trees or Grid Files (later).

4 The data has a complex structure of types and integrity constraints, which are best expressed declaratively in a database schema, and which is best maintained by a number of knowledge acquisition programs kept separate from the expert system.

Expert Database System Architectures

The simplest forms of architecture are shown in Figure 1, which shows two forms of coupling which are discussed below in greater detail. Either the expert system has an attached query generator which composes an SQL query which is then sent to the database system for processing, returning a set of results, or else it uses its own query optimizer to plan how to search the database and it takes decisions dynamically based on the values returned.

Both architectures in Figure 1 only support queries, since the main need is to access the accumulated expertise in a database and to use it for diagnostic, or advisory or "help desk" expert systems. However, a database can also be used to store plans and designs and new facts. Thus, in future, we may well use databases to contain rules or procedures, besides complex structured data and the integrity constraints needed to validate it all. Thus much of what is now in expert system applications may move into the database, or into a suite of application programs sharing stored procedures and data.

One such architecture is shown in Figures 2 and 3 which are taken from Paton and Gray (1988). This shows an Object-Oriented Database (OODB) which is loaded initially with primitive system code, and then with Prolog procedures, which are themselves generated from a Schema describing fact types. These in turn are accessed from the OODB by a load utility which is used to populate the database. This process is then repeated incrementally as new types of facts are added. Figure 3 shows view definitions being compiled and stored in the database for use by a query language (DAPLEX) or expert systems. Also new procedures for computing with new types of data are defined, compiled and stored. In this case the system is built using Prolog and the Persistent Programming Language PS-Algol, which supports the long-term storage on disc of facts and procedures. Interestingly, a similar diagram is also given by Kerry (1988), but using a Knowledge Acquisition Sub-System, a 4GL and a Maintenance Interface in place of the Load Utility, DAPLEX Parser/Interpreter and PS-Algol Compiler modules respectively; also it is based on SQL and relational databases instead of an

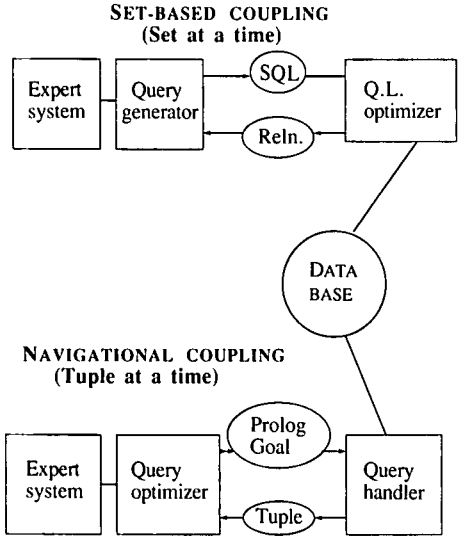

Figure 1 Forms of coupling from Expert System to Database

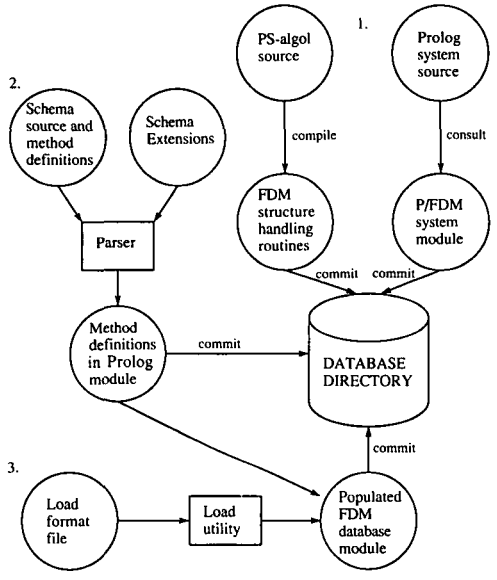

Figure 2 Database architecture—stages in the construction of a database. 1. P/FDM system source in PS-algol and Prolog is compiled. 2. Schema definitions are parsed, populating a new module with method definitions. 3. The load utility parses a file of data for the database and stores it using the methods defined in 2.

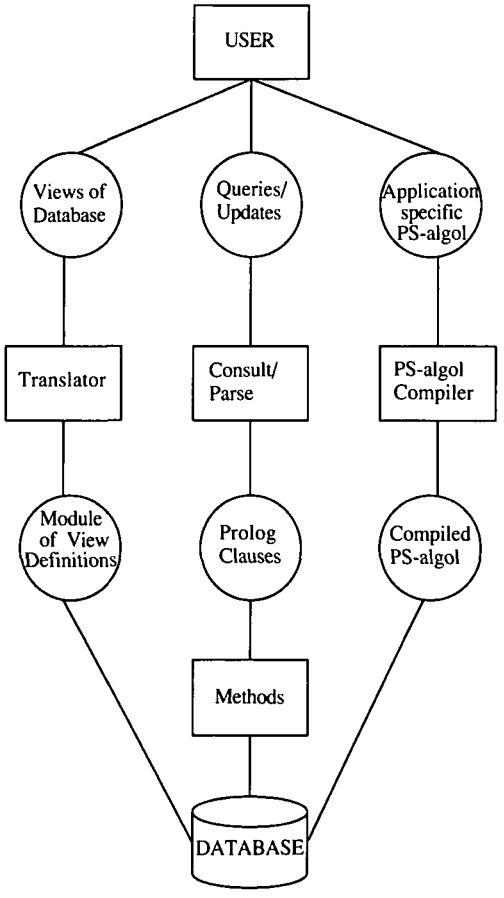

Figure 3 Accessing an OODB—Queries and view definitions written in high level languages are converted to Prolog, which in turn invokes methods written in Prolog or PS-algol which are stored in the database and shared among users

OODB. Nevertheless the diagrams are similar and both show features of the expert system that have migrated into the database/knowledge base. Thus we can expect manufacturers to provide some form of "Knowledge Management" layer in the operating system; this will support their expert system tools in much the same way that we have seen the development of operating system modules to support window handling and commercial database management.

3 Programming languages and databases

The problem of interfacing an expert system to a database is a special case of the more general problem of interfacing a programming language to a database. Unfortunately, databases are not that easy to work with, particularly when one wants to combine data access with calculation. The new and popular Relational Databases, based on Codd's relational model, present an awkward interface problem to the applications programmer, which Bancilhon (1988) calls an *impedance mismatch*. This has to do with the fact that the standard query language (SQL) was designed to enable optimization of set-based queries by cleverly batching data items in order to reduce disc transfers. However, this is not a very convenient way for a program to use a database, since the programmer then has to find ways to store these sets of answers and in particular to declare data structures of the right type to hold them. Instead, programmers tend to prefer a navigational view of access and update, where variables are bound to successive tuples in a relation, and one can use iterative constructs and test the contents of tuples for missing values etc. Thus, typically, one wants to maintain tight control of the search strategy within the expert system, instead of handing a request for a large set of answers over to the database query optimizer. It is the classic problem of trying to couple two "smart" systems, each of which thinks it knows best!

Forms of Database Coupling

Two approaches have evolved to database access. The first we shall call *set-based* coupling and the second *navigational* coupling. In the set-based coupling approach, the application program builds up a query in SQL as a text string made up from values of working variables and constants. This string is then sent over a software pipe to a database running on the same machine or else via a serial communications link to a database back-end procesor; in either case the query is run to completion and returns the complete set of results (usually as a series of printed values with separators between the tuples). The advantage is that one can conveniently link into an existing database, without disturbing the collection of application programs that are in regular use. Furthermore, the database back-end may be set up for shared concurrent access. The disadvantages for application programs were explained above; also the query may take a long while to run to completion, or it may compute a very large number of results which overflow available storage. These things are hard to predict at run-time, and set-based coupling gives no control.

If instead one uses navigational coupling, then the expert system takes much closer control of the search. Typically a single Prolog goal is executed, which returns a tuple at a time from a relation, on request. The values from this tuple can then be used to supply parameters for other goals which return related tuples. This suits a recursive, nested-loop control structure. If one is using Prolog, then the system can abort some of the goals and backtrack to others. It is not necessary to run queries to completion. A user who is browsing can control things by a click of the mouse, or by changing a value on a spreadsheet (Lucas, 1989). Also, one can use complex AI search strategies, where the database itself is treated like a gigantic acyclic graph of relationships to be searched in breadth-first fashion, or by using some heuristic.

Cursors and Hooks

The main technical problem with navigational coupling is that one has to get *hooks* in to the database systems software which will allow results to be returned one at a time. Usually one has to

open a *cursor* on a table which will remember the position of the last tuple satisfying some selection criteria and act as a starting place for the next search. One may even need to have more than one cursor open on the same table, if some kind of recursive search is in progress. Fortunately, these facilities are now being supplied by Oracle and other relational suppliers, otherwise one has the enormous problem of copying the data across into a more suitable form of database. In the case to be considered, the data was actually copied across from a mainframe database into a Prolog main memory database, just in order to allow navigational coupling. The copying was acceptable because the data was largely historical and only updated at infrequent intervals. However, a linkage from compiled Prolog to Oracle using cursors (Lucas, 1989) would probably have given acceptable performance, even if slower.

Concurrent Update

The second technical problem is that it is necessary to lock the open relations against concurrent update, and since a browsing expert system user may leave his terminal waiting for input for long periods, this may then stop updates to most of the database! In this case there is much to be said for taking a "snapshot" of the state of the database and copying it across into another database suitable for navigational coupling. The main database is then freed for update, and periodically the second database can be refreshed; it may not be right up to date but at least it is consistent. Furthermore, as the second database is used for a single user, it does not need the enormous software overhead that is required to implement locking and roll-back on the main database, and so can use cheaper and much more responsive database software, like that found on some PCs.

Loose versus Tight Coupling

Jarke and Vassilliou (1984) introduced the term *loose coupling* to describe the solution of working on a snapshot instead of the real database. More precisely, loose coupling implies that a substantial amount of data is extracted from the database and copied into working memory before knowing what the expert system wants; the system then makes no further requests for data while running, but just uses the extracted subset. Any updates are saved up and sent back at the end of a session. One might consider this to be a means of obtaining a database suitable for navigational coupling. Their system actually copied the database into main memory in the form of Prolog clauses.

They also described a second system implemented in Prolog, but which uses tight coupling through goals which call out directly to the DBMS. In this case data is requested dynamically from the database as the expert system runs, and any updates are done immediately. As soon as the transaction locks are released the updates will be seen concurrently by other expert systems.

Use of SQL

The notions of tight and loose coupling are really orthogonal to the issues of whether to use SQL and set-based coupling. It would be possible to start by retrieving a huge joined set of results via one SQL query and then to continue in loose coupling fashion. It is equally possible to use SQL to open cursors on individual relations and then to use these cursors for navigational coupling as explained above.

In this paper we shall concentrate on the Prolog solution. For those wishing to use SQL from an Expert System there is an excellent review by Kerry (1988). This lists twenty two expert system shells and tools, all of which provide database access, mostly through SQL. It also gives useful specification details and documents the usual irritating problems of incompatibility between Expert Systems and databases in their internal representations of strings, integers and reals. All too frequently this leads to loss of precision, or even of data: for example by truncating strings!

Kerry distinguishes between *direct coupling* where the expert system software has built in declarative facilities for database use, and *indirect coupling* where the system simply permits calls out

to foreign language routines and does not know that these are being used to call DBMS routines. Amongst those providing direct coupling to relational databases are Application Expert, ESE, KBMS, IBM Knowledgetool and TODAY-ES. The AI tools ART, KEE, Nexpert Object and XI-Plus all provide indirect coupling through callouts which in turn use SQL.

4 Use of Prolog for database queries

The Prolog language is peculiarly suitable for database work, since it maintains a uniform mechanism both for computation and data access, through a generalized procedure call based on unification. In a Prolog database, facts are made to look like alternative declarations for a procedure. Each of these declarations (or unit clauses) also looks like a tuple in a relation. The difference is in the way the tuples are accessed. Consider the clauses below for the Prolog procedure blnum(*BlendId*, *Measure*, *Value*).

```
blnum(2157, m1, 2.6).
blnum(2164, m1, 3.9).
blnum(2164, m2, aa).
```

These represent the fact that the blended lubricant identified by the number *BlendId*, when subjected to the analysis denoted by *Measure*, gave the result *Value*. Note that several different analyses may be done on the same blend.

If one makes the procedure call blnum(2157, ml, Y), it succeeds with Y bound to value 2.6 (taken from the first clause). If one makes the call blnum(2136, ml, Y) then it fails because there are no matching clauses. However, if one makes the call blnum(Blend, ml, Y) then it succeeds first by binding Blend to 2157 and Y to 2.6; one can then ask Prolog to re-try, when it re-succeeds by binding Blend to 2164 and Y to 3.9.

Thus the same facts can be queried in many different ways, by giving values to some parameters and not to others. In standard programming languages one cannot do this; one must specify which parameter is input and which is output. Furthermore, if a subsequent goal fails, then the program just halts, whereas Prolog will retry the preceding goal(s) in a depth-first fashion. For example consider the procedure calls.

```
blnum(Blend, ml, Y), Y > 3.0, write(Blend), nl.
```

Here Y is first bound to 2.6, but the following goal fails, so a second solution of Blnum is tried with Y bound to 3.9, which satisfies the second goal, and the final goal succeeds with the side effect of writing out the Blend identifier.

This pattern of access is exactly what one uses in databases. A call with free variables for all parameters causes enumeration of all the tuples of the relation in the database, one by one, with each attribute value bound to the value of the corresponding Prolog variable. A call with values fixed for some parameters can use the values in conjunction with an index to locate the desired tuples very quickly. A call with all the values fixed will either succeed at once through indexing and finding a matching tuple, or else fail and backtrack.

The great advantage over SQL is that the results of the database search are returned as values of variables which can then be passed to other Prolog procedures. These procedures are defined with a body which can do arbitrary recursive computations. They can build or take apart lists, with all the power of LISP, and they can call out to procedures in C for numerical computation or else to operating system and ethernet communication procedures. Thus computation and access are smoothly integrated. There is no "impedance mismatch".

Better still, Prolog has the power of LISP to build data structures (called term structures) and to treat these as Prolog goals or procedures. In other words a Prolog procedure can write a new piece of Prolog and run it. This is just as effective as generating SQL text, and uses a more powerful language. The resulting pieces of Prolog can even be sent over communication lines to another

computer. Furthermore, it can be rewritten and optimised, as shown by Warren in the Chat 80 system (Warren, 1981).

For these reasons, we used Prolog in the Luboil project and advocate its more general use. Relational databases should be provided with "hooks" to implement Prolog interfaces as well as SQL interfaces. Prolog might even become a universal query language as discussed later.

Finally, Prolog provides a control structure which is suitable for many expert system applications and also for use in navigational coupling. Its main weakness is with forward-chaining expert systems, where one has to use the meta-level control facilities of Prolog to assert extra clauses into the "working memory", and thus over-ride the basic backtracking strategy.

Currently there are at least five Prolog interfaces to relational databases, which are described in the collection of papers edited by Gray and Lucas (1988).

Query Optimization

Since a navigational coupling solution takes control of the search strategy, it must also consider the classic problems of database query optimization, namely the need to use indexes in order to avoid long searches, and the need to search in clustered fashion, a page at a time, in order to reduce disk transfers. Thus a search path that suits a programmer may not suit the database and may need to be transformed. This is the main argument for expressing database queries in SQL: the database query optimizer can do the transformation. Furthermore, it has access to data on physical storage layout, cardinality of relations, average page changes required for various types of search, and so on. Some people argue that making this information available to other programs goes against the principle of hiding data storage representations from outsiders.

Thus, they reason that the calling program can never do such a good job on query optimization. However, set-based coupling does have a considerable overhead on initializing workspace files, and so on, before running each query (in the SQL used with Rapport this could take minutes). Also optimizers themselves are slow; thus navigational coupling may win over set-based coupling (Bocca et al., 1986). Furthermore, current query optimizers usually fail to deal with more than ten joins, so that it is a positive disadvantage to use them for very complex queries. Such queries have arisen on the Luboil database, and are even more common when trying to use frame-based systems as explained later.

Navigational coupling is used when one is taking close control of the search, and thus one should be able to do full optimization. Problems of data independence can be solved if the physical storage statistics referred to earlier are held as tables which can be passed out to the optimizer and used for the current session only. In fact optimization is a form of planning, and the expert system usually has good facilities for trying out plans, and it knows more about its use of the database than the DBMS can ever do. Thus it seems better to give the expert system all the information and let it perform the optimization. This issue is discussed further under "Performance".

Navigational Coupling in the Luboil demonstrator

For the Luboil demonstration system the decision was taken to move the database across into a compiled Prolog form that is accessible by navigational coupling to LISP. Advantage was taken of the Prolog implementation on Symbolics to load and compile the data as a Prolog program with about 200,000 unit clauses, equivalent to some 10 Mb of data on file. This proved very impressive in terms of retrieval times, improving on the original mainframe times by two orders of magnitude! In part this was because Symbolics Prolog indexes clauses on all attributes; Quintus Prolog does not and the results using it were less spectacular but still very acceptable. Also, as remarked earlier, data in this form can be updated periodically, but not shared.

Another alternative was tried, using indexed sequential files with secondary index information held as Prolog clauses. This proved more suitable for machines with less memory and without good Prolog compilers. Set-based Coupling using Rapport's own message facility proved impossible to

implement, because of a lack of fast communications hardware between Ethernet and a conventional mainframe. The alternative of installing Rapport on the Symbolics was considered, but taken no further once the very superior performance of compiled Prolog on the Symbolics was established. Furthermore, it would not have made any use of the bulk of the Rapport software, which provides for shared access, concurrent update and commitment of transactions. This decision may be re-examined in the future, since the use of Prolog allows one to change fairly easily from navigational to set-based coupling. For example, one can define a view in SQL representing the join of several relations to be computed in set-based fashion, and then use Prolog to send parameters for this view and to get back answer tuples one at a time, in navigational fashion. As argued below, this is a significant advantage.

5 Query languages

Most expert system interaction with DBMS is in the form of queries. The advantage of querying the database is that it remembers great volumes of detail precisely.

Select-Project-Join

The basic form of the query is well established in the database literature and consists of the operations of selection, join and projection. It is illustrated by a Prolog query on Blend Analysis information in the Luboil demonstration system:

```
sulphated_ash (Bid, Bo, A):- blnum (Bid, 'NSA', A),
                             blends (Bid, –, –, ... Bo, –, –).
```

In the syntax of the SQL query language this would be

```
SELECT   BLEND-ID, BASEOIL, ASH
FROM     BLNUM, BLENDS
WHERE    BLNUM. ANALYSIS-TYPE = 'NSA'
AND      BLNUM. BLEND-ID = BLENDS. BID
```

Here we are finding the result of the sulphated-ash by *selecting* tuples with analysis fields containing 'NSA', and *joining* these tuples to those with the same value for blend identifier, and then *projecting* out just the field values for blend identifier, base oil (from blends) and ash content (from blnum).

This type of query is the mainstay of the SQL language, and can easily be expressed in Prolog. Extra joins are done by adding extra terms onto the Prolog clause, and the conjunctions of selections are treated similarly. For example, a test on ash content less than 5 is performed by simply inserting ",A < 5" after "blnum (...)".

Range Selection

Many of the queries will want to retrieve formulations with numeric values lying within a certain range. This can be done in Prolog by using a conjunction e.g.:

```
(5 < X, X < 9.1)
```

How it is done will depend on the indexing facilities in the database (see later). Prolog can also express queries on arithmetic expressions and totals, e.g.:

```
W is (0.5*(X + Y) + Z), 5 < W, W < 9.6
```

Aggregation

It may be desirable to retrieve a quantity which is summed over all the components in a blend, or is the maximum value taken over all components. This is known as a "group-by" query. It is

expressible in Prolog by using the "setof" or "findall" predicates to collect up all the values into a list, and then using a recursive function to sum them. The example below shows the computation of total zinc content from the separate analyses of zinc in each component of a blend. Here the variable S contains a list representing the set of amounts of zinc (*Anal*) weighted by the percentage of component (*Percent*) and lsum is a recursive Prolog procedure to sum the list:

```
zinc_total(BlendId, Tot):-
findall( Percent*Anal,
 (blcomp(BlendId, Component, Percent),
 comp_anal(Component, 'Zinc', Anal)),  S),
lsum(S, Tot).
```

Probably one should reserve certain named functions such as lsum, for doing totals, maxima etc. One could even develop a high-level analogue of "set-of" in Prolog called "group-by" which combined the computation and set formation, as in ASTRID (Gray, 1984), since it is then easier to optimize the query, and it saves forming large intermediate lists.

Unknowns

Prolog allows one to implement a simple extension to handle null values coming from an existing database, by using a particular atom 'unknown' to represent an unmeasured value. This was necessary because the RAPPORT relations used to hold tests in the Luboil demonstration system are deliberately arranged not to hold unknown or "null" values explicitly. Instead they just hold triples of the form ⟨Test-identifier, Measurement-name, Value⟩. Where a particular measurement is absent, there is no triple! In fact, this is a common way for relational databases to avoid the problem of holding null values. However, for the end user who wishes to see a table of blends with columns giving test results, this has to be made explicit. Once again, Prolog can be used to do this. It is only necessary to iterate over a list of measurement names and to return the pseudo value 'unknown' where a measurement is not found. It is necessary to use the 'cut' operator (!) to produce the effect of an if-then-else test.

```
find(Bid, Meas, Val):-(blnum (Bid, Meas,V), !, Val = V; Val = 'unknown').
```

If the goal blnum succeeds in finding a tuple for the given values of Bid and Meas then it returns a value in V which is then unified with the result parameter Val. Otherwise Val is set to the atom 'unknown'. An extended definition is necessary where we wish to enumerate all measure and blend combinations when 'Val' is given as 'unknown':

```
finda(Bid, Meas, Val):- var(Bid), !, blends(Bid,_,_,...),
            finda(Bid, Meas, Val).
finda(Bid, Meas, Val):- var(Meas), !, measures(Meas),
            find(Bid, Meas, Val).
```

In this case we check if the blend identifier Bid is not given and if so, we use the goal blends(Bid,...) to instantiate it with each possible value in turn. Similarly we enumerate possible measurement types. This type of query is surprisingly difficult to do in a database query language such as SQL. In fact one needs the *Outer Join* construct proposed by Codd (1979). This is not popular in a general database system, since operations such as selection on the null values in the resulting relation are not defined. However, in the Luboil demonstration system one can define their meaning and allow them in queries where the unknown value is returned explicitly, for example:

```
lowtype1(Bid,A):- find(Bid, 'Type1', A), (A = unknown,!; A < 5).
```

Here we ask for ash values below 5, but accept the value 'unknown' as well, since it will be passed back as the value of A.

Recursive explosion of constituents

Sometimes it is necessary to find all the basic chemical constituents of a blend. The blend may include other blends, which may themselves contain blends, and so on. The query is similar to the classic "parts explosion" problem for a bill of materials. It is not expressible in the standard database query languages since it needs to perform joins recursively until no more composite components are left. Since Prolog includes recursion it can express such a query. There are interesting problems here for query optimisers. Rosenthal et al. (1986) have suggested a new construct in relational algebra which combines the operations of explosion and computation of aggregate values (e.g. total weight of constituent). Also newer database hardware may allow such queries to be evaluated much faster by parallel processing. Expressing the query in Prolog should allow it to be transformed to utilise newer processors.

Aliases

Users often refer to the same chemical by different names. Also, where results are merged from different databases the measurements may use different names for the same thing. This is easily handled in Prolog. The original query can be processed against a table of clauses giving synonyms and the database equivalent. Where the database has only two equivalents then a selection can be expressed simply as a disjunction, e.g.

```
(A = 'C-1' ; A = 'C-OO1').
```

Where there are many equivalents one can set up a table of clauses giving canonical equivalents to synonyms and then check the equality of the canonical equivalents, thus:

```
alias('C-1',cxx).
alias('C-001',cxx).
etc...
```

6 Prolog as a universal query language

The Proteus project (Atkinson et al., 1984) devised a universal query language based on relational algebra, so that queries in this form could be passed from a host to a remote site running an unknown DBMS. The remote site used a known schema, represented in a universal intermediate schema language, the ACS (Stocker and Cantie, 1983). This allowed any site to use its favourite query language to access any database on an inhomogeneous network. Interestingly, several of the nodes chose to use Prolog as a means of implementing the translation of the query language to their local query language and vice versa. Following this, Howells et al. (1988) have developed a source to source translation system between pairs of database query languages, which is table driven and implemented in Prolog. This makes access to any remote database very much easier. Their system uses a common intermediate query representation based on relational algebra.

Now, since Prolog has the power (like LISP) to treat pieces of Prolog as data structures, and also to generate and execute them dynamically via the *call* statement, it is attractive to express the queries themselves as pieces of Prolog rather than in relational algebra. The equivalence is given in Gray (1984). Furthermore, the optimizations involved in rewriting pieces of relational algebra can be done just as well on pieces of Prolog, as shown for example by Bundy's (1984) use of Prolog for symbolic manipulation in mathematics, or Warren's (1981) Chat 80 system.

The query language could start by using a subset of Prolog for select project and join, as explained above, and as proposed at the Alvey LKBS workshop (Gray, 1984a). All Prolog interpreters can handle this subset. Extensions to the query language would depend on how much was to be done by the database query optimizer, and on whether it knew how to use special hardware in a database engine. If the optimizer could handle recursive queries, or ones involving group-by, then a more complex piece of Prolog could be sent across, to take advantage of this.

The great advantage of Prolog is its conciseness and expressiveness for manipulating lists and sets and for this pattern matching. This, in conjunction with the ability to treat pieces of Prolog as data, makes it much the easiest language to write pre-processors that massage database queries into a desired form, or transform the results into a 'view' desired by the user.

The second great advantage of Prolog is that it does not distinguish between results obtained by database matching, and those obtained by computation. Both are represented simply as goals. The user does not need to know whether the goal was achieved by processing the goal through a complicated view mechanism, with call-outs to a remote database, or just directly by matching against databases held as clauses in memory. Thus it is very easy to adapt a query language based on a subset of Prolog for use either with set-based or navigational coupling, and also to new database hardware as it appears.

7 Views of a database schema

Modern database theory supports the idea of having many views of a single logical database to suit different users, some of whom may be browsing, whilst others may be expert systems programs. The logical database is made up of data associated with entities, and data which describes relationships between entities. The data is usually "normalized" so as to keep each piece of data in one place only, and to associate it with just those key values which it immediately depends on. This makes it much easier to update, and avoids certain classic anomalies. However users browsing at a terminal do not like seeing the data split up into many separate tables. They usually like to view the database as if it were one big table.

It is the job of the view mechanism to create this illusion. Some systems are very inflexible and allow only a limited set of canned, pre-compiled views. The advantage of Prolog is that it is possible to create views dynamically to suit the user. Thus, a search is made for the relevant tables and their columns, depending on the user's query, and a sequence of joins and projections is composed to produce a single table of results. Initially the table may be fairly large, incorporating a lot of data that roughly fits the requirements. A browsing user will want to see what fits best, and compare with other values close by. They will want to refine the set of values by further selection, and may use joins to pick up and inspect other associations not specified in the original query. This demands two things of the mechanism: (a) temporary storage for intermediate tables, which may be quite large (b) the ability to browse these tables as if they were the original database. A Prolog system can provide both facilities. Requirement (b) is easy because of the generality of solving Prolog goals referred to above.

Requirement (a) is easy for small tables, which can be held as unit clauses in memory, since Prolog allows one to assert any new type of clause at run time. However, if the intermediate tables are large and need indexing, then it may be necessary to store them in the database on disc. Unfortunately, current databases are often inflexible, and demand to know the type of the table well before run-time. To get round this it may be necessary to devise a kind of universal table of the form: (table + column name, Row identifier, value), or several such tables, one for integers, one for reals, and one for strings. Fortunately, modern relational databases are becoming more flexible in this respect, by allowing dynamic declaration of temporary relations. Furthermore, workstations and the larger PCs are now available with many megabytes of memory and those Prolog systems which can use all of this (and can reclaim it) will have little need to keep intermediate tables on disc.

Hiding Schema Details

Databases tend to grow so that they contain very large numbers of relations (or record types), many with a substantial number of attributes (or fields). The user finds this baffling and wants to home in on a small set of tables, or one composite table, satisfying his needs.

To allow inexperienced users to query a database, a query language called TREQL (Thornton Research Easy Query Language) has been developed by Lunn and Archibald (1988), which does not

require the user to understand the details of the database schema. TREQL requires the user simply to list the attribute names of the items required from the database. These items may be from a number of relations. TREQL will determine the necessary joins, compose a Prolog query and extract the data.

It is fairly easy to deduce what column names the user is interested in, particularly if there is information on possible aliases being used, since the query must either give them explicitly, or else implicitly in the form of selections. Deducing the relevant tables is not so easy, since there may be several tables with a particular column name, especially if it is part of a key. One approach is to find a minimal set of tables which, when joined, contain all the attributes.

One can even follow Kaplan (1979) and find a sequence of joins that link two data items via intermediate tables that contain no columns mentioned in the query. However, this is dangerous if the user does not understand the functional dependencies involved. For example, one table may relate an attribute C to two attributes of interest A and B, whilst another table relates C to A only. Which one should be used? Thus it is useful to have some kind of clarification dialogue, or else to pay back the query as an expanded paraphrase naming the relationships that have been assumed. However, paraphrases tend to be convoluted and hard to read. A better scheme for the browsing user on a modern raster graphics terminal is to show the relationships between tables in a graphical fashion by showing a set of connected table headings, as in Query-by-Example. The connections can be inferred by highlighting column names, or else by drawing lines between them. With the aid of a mouse one can click on particular column headings and print out comments about their interpretation.

Joins without losing information

A classic problem with inferring joins is that, since a join acts as an intersection, tuples can be lost. For example, if the user wants the results of several tests on blends to be joined together, and one of these tests has not been performed on some blend, then there will be no tuple for this blend in the joined result, even though the other test results were of interest. This does not happen if Codd's *outer join*, mentioned earlier, is available, but it is not in standard SQL.

The approach used in the Luboil demonstrator is to deal with this problem where it is known to occur, namely in the use of the relation which gives the results of test on various samples. The expert system wants to see this as a table with a very large number of column names, one for each test name with many samples having "unknown" values in any given column. The view required for the expert system then projects the table down to the set of column names, given by the query. The table is actually held as a relation giving triples ⟨sample identifier, test name, value⟩. The transformation is similar to those discussed under Query Language, and is easily handled by Prolog.

Denormalization

A relation in normal form may have a composite key consisting of several fields. The fields may have a natural hierarchical sequence, for example a relation may give information on houses keyed by number within street name within city name. When large numbers of tuples have the same city name it may be desirable to print it out once only, possibly in a separate area at the top of a screen. This process can be repeated in nested fashion (Gray, 1982).

It is also possible to take clusters of related tuples which consist of a repeated key value followed by a single attribute taking different values, and to collect the set of values for this one attribute into a list, shown in brackets for display to the user.

All these techniques produce relations which are not in normal form. This is a disadvantage for further querying, but an advantage for screen layout and presentation. It can also be more concise as a method of passing results back via set-based coupling, since it reduces the repetition of data values.

Holding Data in Frames

The discussion so far has assumed that the database has a relational form and that the data is held in tables. However, in an object-oriented database, individual tuples are held in separate instances of frames in memory, each of which has the same set of slots and slot names but is filled by values from a tuple (or group of tuples). Figure 4 shows an example of such a frame containing personal information and references to other frames (or objects) containing data for related persons. Thus, instead of browsing down a column of tuples in a table, the user wants to see pictorially what other frames a given frame is related to. For example, from a frame representing a blend one might move to frames giving the components, or across to a frame giving the refinery making the blend, and so on. This kind of browsing is familiar to Codasyl database users as "navigation". However, it is made much easier and more attractive by the use of a mouse to click on slots, and by graphic displays of ancestor relationships as trees or nets.

If data from a database were to be held this way it would be easier to interface it navigationally to a tool like KnowledgeCraft, since it uses the KnowledgeCraft representation extended to disc. In fact efforts are being made by the distributors (CMU) to migrate frames (schema) to and from disc. In the longer term it will need to use an object-oriented database as described in the final section.

Self-describing data

The other advantage of holding data in frames is that it is possible to use the IS–A hierarchy to provide generic information about the entities being browsed, which is useful for understanding the nature of the data. Further, the functional relationships are mediated through the slot-names, which contain a list of one or more frames holding related information. Effectively the join between the tuple represented by the frame and a set of other tuples has been pre-computed. The user can follow it with the browser and also consult facet information on the slots which describes the relationship further (even as a reference to a "relationship" frame).

In relational systems, data about data is held in a Data Dictionary, which can usually only be updated with difficulty and may involve recompilation of application programs, or re-loading of databases. Sometimes a set of meta-relations are provided which effectively allow one to query the names of relations and their column names and (Fortran) data types. However, frame systems make available a much richer set of metadata. It is possible to put into facets information on precise ranges of values, or predicates to be satisfied by slot values. The maximum, minimum and actual cardinality of a slot value can be given, also precisely what type of relationship it represents, by reference to a frame for a user-defined relation. Through the IS–A hierarchy reference can be made to generic objects representing a class. These objects may be treated as abstract "concepts", and information on

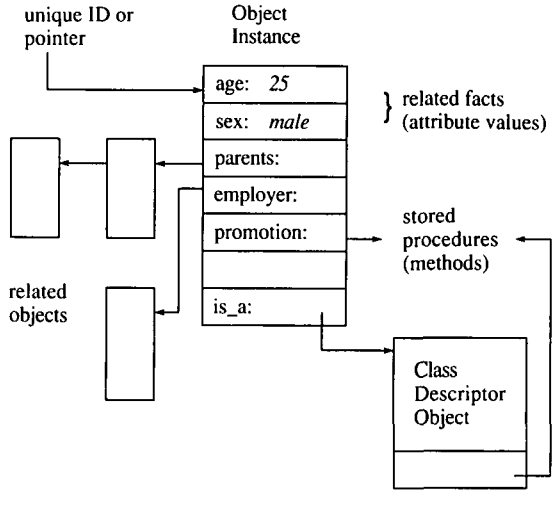

Figure 4 Object Storage

"roles" relating them to other concepts can also be represented in frames, as in the KL-ONE language (Brachmann and Schmolze, 1985), and in the MAKR language (Gray et al., 1988b). There are great possibilities here which are only beginning to be touched on.

8 Performance issues

Query Optimization

If Prolog is used as the intermediate query language then we have to be aware of the left-to-right evaluation strategy. Thus it is important to order the goals so that the first goal is one that can use a database index in preference to one that just enumerates a large relation. Thus it is much better to look for

```
blnum (Bid, 'NSA', A), blnum (Bid, Tst, X), X < A
```

than the other way round, since the first goal uses 'NSA' selectively to find a few values of A, whilst the second enumerates all possible values of Bid and Tst, unless it is given a particular value of Bid by the first goal.

This problem is well known in database query optimization. It is the problem of re-ordering joins and selections which was solved by Selinger (1979). The same problem was solved for Prolog by Warren in 1981, making use of Prolog's ability to reorder a list of goals considered as term structures. The crucial information, which must come from the database (as was discussed earlier under data independence) is the cardinality of the relations, the selectivity (range of possible values) of the various indexes, and what arguments are indexed. Given this information it is relatively easy to write a Prolog optimiser for select-project-join queries.

Database Storage Structures

Various type of storage structure are available, ranging from serial text files, through simple hash tables to full balanced B-Trees and on to pointer networks. We shall discuss them under criteria:

1 *Efficiency of Direct Access*

The fastest methods use hashing, as in Rapport, or extensible hashing, as for example in NU-PROLOG (Ramamohanarao et al., 1986). If the hash tables can be stored in main memory then these techniques can guarantee access in one disk access if a unique key is given. The superimposed codeword technique used in NU-PROLOG can also work with combinations of key values. However, hashing usually leads to randomization which is not good if the values are ordered and one wants all tuples with a value greater than a given value. For this purpose it is better to use B-trees, whose structure partitions the database by the value concerned, so that one can avoid searching large parts of it altogether. A more general alternative is to use Grid Files (Freeston, 1988), which use an in-store directory structure to partition relations according to combinations of argument values. Thus one has only to search a few hyper-squares in the grid instead of scanning the whole relation. This looks very promising for use with Prolog.

With a collection of data that is monotonically growing and infrequently updated, which is typical of scientific measurement data, it pays to index it on all keys or combinations of keys that are used as access paths. These secondary indexes take up space and slow the process of updating, but they save time when searching large collections of data.

2 *Loading and Initialization*

If a database for use by an expert system is just held as serial text files, then one must allow for the time to consult them at the start of a run, and to build auxiliary pointer structures and indexes in

memory to speed up access. The slowest part is often the time taken to read strings and convert them into unique identifers representing atoms, as in LISP, so that each different occurrence of the same string is represented by a pointer to a unique cell. This is a form of Lexical Token conversion. Some Prolog systems allow one to save an image of the consulted files on disc, which can be reloaded very quickly. However this will have to be loaded, modified and resaved every time the files are updated, and it takes a great deal of disc space.

Where a database gets very large and only a small part of it is to be accessed or scanned in any one run, then it pays to hold it on disc and only to fetch in the relevant portion via indexes as required. A form of buffering or cacheing is usually needed to save excessive disk transfers for pieces of data that are accesed again and again.

Another aspect of loading concerns the time to open files and initialize temporary files and workspace, which can take surprizingly long for a database on disk.

3 *Saving Execution Images*

If a complicated search tree or graph has been built up with many frames, and it is desired to preserve it for future sessions, it can take a surprisingly long time to copy it out to disk in textual form for re-consultation. In the case of an early version of KnowledgeCraft used on the Luboil project, the saving of all frames in a context could take ten minutes or more, whilst repeating this for several contexts could take hours. A dump of the virtual memory would be quicker, but it is likely to take an unacceptably large amount of disk space (hundreds of megabytes) in cases where the system has run for some time without a garbage collection to compact the active working memory.

4 *Using Pointer Mechanisms*

Tuples in a relational database are associated by matching the values stored in them. This has to be speeded up by pairing the values with the addresses (Tuple identifiers or TIDs) of tuples that contain them, to form an index. Prolog unit clauses containing atoms or small integers also have to be searched and matched by unification. The search is often speeded up by hashing on the first argument, which it pays to choose carefully. Nevertheless, access via other arguments will be only at the speed of a slow serial scan, unless one builds ones own secondary index in the form of extra unit clauses, containing the argument of interest in first position, followed by the key of the original unit clause.

Semantic net databases and Persistent Heap databases represent frames more like Codasyl databases do, as records with embedded pointers. Pointer following is much quicker than associative matching, which is its big advantage, but it is harder to change the sizes and slot names in records dynamically. A possible solution is to use a scheme like LISP, where frames and records only contain atoms, which point to unique cells, from which the print name of the atom can be recovered, and also, the address of the cell currently representing the frame. This address can be overwritten to point to a new enlarged cell, if the frame has to be expanded. Furthermore, it is relatively easy to show the contents of any frame on the screen, just by listing the print-names, which are somewhat more meaningful than pointer values. The cost of an extra indirection in accessing every frame seems well worth it for the flexibility it provides.

Limitations in Current Software

It is all too common to find that a piece of software which performed well with a small volume of data either crashes on hitting some hidden maximum size limit, or else suffers unacceptable performance degradation, when working with realistic amounts of data. This is distressingly true of Prolog implementations, for reasons given below:

1 *No tail recursion optimization*

Where a sequence of tuples is returned as a long list, the Prolog method for searching the list has to be written recursively. Standard techniques can turn a recursive call at the end of a procedure into a

jump, producing iterative code which saves one from using and then running out of stack space. This optimization is a common feature of compilers, but is less common with interpreters, which all too often allocate stacks of insufficient size.

2 *Lazy evaluation for large sets*
A good Prolog interface to a database will be able to enumerate the tuples, returning them one by one by re-succeeding at the goal, without storing them all up in Prolog workspace. This corresponds to "lazy evaluation" in functional programming. It saves forming large lists and running out of space. Where it does have to form an explicit set, using *findall*, it should be able to use virtual memory or disk files efficiently if the set gets very large.

3 *Interpretation and Compiling*
Warren has shown that compilation, by saving unnecessary run-time tests in unification, and by optimizing tail recursion, amongst other things, can speed up Prolog by a factor of 10 or more. The greatest speed-ups are usually made when the compiler can be given mode information on the expected values for variables. However, one must beware of two restrictions in some of the cheaper compilers, which stop one from using Prolog as its own meta-language for optimization.

The first restriction is to stop the use of *assert*, thus forbidding the addition of new clauses not consulted at compile time.

This makes it impossible to use unit clauses for the storage of small relations and temporary workspace, and may even make it impossible to use *findall* unless it has special workspace of its own. It also makes it impossible to generate and assert several Prolog procedures that call each other.

The second restriction stops the use of *call*, which is used to treat a term structure as a Prolog procedure and to execute it. This makes it necessary to write out the procedure as text and then to recompile all or part of the system and restart. It makes it impossible to build views dynamically to suit a browsing user, which was one of the principal attractions of Prolog listed earlier.

Impact of Large Memories

Some of the problems of performance just discussed are being overcome by improvements in hardware. In particular, the great increases in memory size which allow workstations with 20 Mb to be cost effective have revolutionized data access. Thus one may be able to get all the data into memory and so avoid worrying about clustering of data. At the very least, one can keep all of the indexing structures or hash tables in memory, and techniques mentioned earlier will guarantee to need at most one disc access for an item, instead of a slow sequential search. Alternatively, one may be able to use the entire memory as a gigantic cache, whereas the older database management systems used only fractions of a megabyte for their buffers. This idea has been developed by Harland (1988) in a revolutionary processor the REKURSIV, which combines a large cache with object addressing and user written microcode to greatly speed up data access. Experiments are in process at Aberdeen and Edinburgh to use this with Prolog.

9 Relation to current research

In this final section, we consider the future directions being taken by research in USA and elsewhere. A good source of references are the Expert Database Systems Workshop and the two International Conferences, all edited by Kerschberg (1984, 1986, 1988).

Compiling Prolog Queries

An interesting research project NAIL (Ullman, 1985) applies compiler techniques to transform a navigational style Prolog query into some form of relational algebra or set-based query. Zaniolo (1986) and Tsur (1988) have gone further and designed an extension of Prolog into a pure Logic

Data Language LDL which includes set operations as primitives. They are researching ways to compile queries in this for evaluation on parallel hardware with multiple search engines.

An important aspect of this is that the data retrieved from disc can include nested Prolog term structures, and not just flat normalized tuples built from simple atomic values. Interestingly, the original database engine DELTA, built at ICOT for the Fifth Generation project, was abandoned partly because it did not handle term structures. This shows that any Prolog to handle database queries should have good facilities for constructing sets and handling term structures.

Frame-Based Systems

Large LISP-based tools such as KnowledgeCraft, used on the PFES demonstrator, store most of their knowledge in frames instead of using Production Rules. Frames (see Figure 4) are very different from normalized relational tuples. Thus they do not suit the simple storage provided by a relational database. There are various problems:

1 Some instances of a class of frames may have extra slots which are not present in others, which completely breaks the type structure of a relation.
2 Values in frames may be lists or tree structures and not simply atomic.
3 Frames may have meta-data attached in the form of meta-slots or facets containing descriptive information or constraints.
4 Slots may have attached procedures or demons which fire when the contents are accessed or changed.

An attempt was made by Abarbanel and Williams (1986) at Intellicorp to store KEE frames in a relational database. They found that the frames had to be split up into tiny components, each stored in separate binary relations, with large amounts of concatenated key information required to identify the components. Then these components had to be joined back together, often involving over ten joins, which caused the query optimizer to give up whilst considering all the different ways of doing the joins!

A more practical technique has been implemented by Carnegie Group which allows KnowledgeCraft frames to migrate to disk and back (Fox and McDermott, 1986). This saves turning the frames back into textual form and gives some of the advantages of persistent heap storage. In particular it tends to cluster the frames together on secondary storage which significantly reduces the amount of virtual memory needed to run the tools.

Object-Oriented Databases

A possible solution to the storage of frames is to use an object-oriented database in place of a relational one. These are currently being developed at Oregon (Maier et al., 1986), at Boston (Dayal et al., 1986) and by researchers at DEC (O'Brien et al., 1986) in conjunction with Zdonik and Moss from Brown and Amherst respectively. An early commercial prototype VBASE is available from Ontologic of Boston. These databases allow one to store very general types of pointer structures on disk. They also emphasize the use of abstract data types whose internal representation is known only to the generic procedures or methods which access them. Their inheritance rules are based on classes, rather than on default inheritance as used in frame structures, and thus it has to be adapted.

Their most important feature is their ability to store code in the database along with the data. This is vital for object-oriented programming, where the idea is to store procedures as methods with the data, instead of keeping them locked up in huge application programs. Thus the procedures can be retrieved from the database and shared by many users, which is the basic raison d'etre of databases.

Various programming languages can be used with Object-Oriented Databases. C is commonly used to implement them, and more recently C++. Atkinson and Kulkarni (1986) have used the persistent language PS-Algol. Gray (1988) describes the use of Prolog to implement an Object-Oriented Database for storing many megabytes of data on protein structures, together with Prolog procedures used for calculation and inference. In this case Prolog variables are bound not just to

atomic values, but to object identifiers; generic methods defined in Prolog are then used to access and update the object components (Gray et al., 1988a). Effectively the object identifiers stand for complex term structures held on disc and we have already seen the need to store such term structures. Thus Prolog is able to compute and manipulate objects with the same ease and power that was displayed with relational tuples.

Navigational coupling is necessary when using object-oriented databases, in order to pass references to objects directly. It is also necessary when one wishes to use a database combined with an expert system that enforces integrity constraints expressed in logic. Often in these cases, the most efficient way of updating is to trigger particular checks and actions by the update, as happens with "if-added' rules in frame systems (Stonebraker, 1986).

Expert Systems as Virtual Relations

Larson (1985) at Honeywell Research has an interesting idea about using a distributed system of databases and expert systems which are called up as high-level Prolog goals. Thus the expert system is treated as a kind of 'virtual relation', some of whose arguments are provided and which uses rules to generate answers returned as tuples of input arguments and results. For a futuristic architecture, with cooperating systems available on different machines connected on a network this looks interesting. It certainly shows the generality and potential of using Prolog as a "universal query language".

10 Conclusions

We have drawn on our experience in the ALVEY PFES project in order to discuss the issues involved in interfacing an Intelligent Knowledge-Based System to a large database, and have argued in favour of using a subset of Prolog as an intermediate language. The main advantage is the ease with which Prolog can be used to provide "customized views" of an existing database which suit the calling system. We have seen examples of this in the presentation of measurement data from a Rapport database as columns containing special 'unknown' values for missing tuples.

Yet another advantage of Prolog is that it is very concise, and is its own meta-language. In consequence, one piece of Prolog can be used to construct a sequence of Prolog goals representing, for example, the standard "select–project–join" query. This can then be exported as text to another process running on the same machine, or even over a network. Thus both navigational and set-based coupling are possible, and the method used can be hidden from the calling system, which does not need to know whether the database is on the same machine.

Furthermore, Prolog is a good language for query optimization. We have discussed various forms of query optimization, from simple re-ordering of goals to the total transformation of a query from a tuple-at-a-time form to a set-based form expressed in a language like SQL. Alternatively, Prolog implementations of tables as large numbers of clauses held in main memory are getting more efficient, and as memories get larger these need to be considered. We have surveyed the various issues in performance, and this is an area of rapid development and improvement.

Finally, we have considered some of the newer types of database, including the use of object-oriented databases, which cluster data into objects and hide its internal representation instead of presenting it as printable columns. These databases look suitable for use with frame structures as used in AI tools, and another Alvey project associated with the PFES demonstrator (Campbell and Gray, 1988) has shown how Prolog can be used for this type of database as well. Thus Prolog shows yet another aspect of its surprising adaptability and usefulness for the interfacing task.

Acknowledgements

This research was partly funded by Alvey IKBS grants 052 and 073. The report originally appeared as Appendix 2C to the final report on PFES, and has been edited to update references and to

include extra review material. We are grateful to Shell Research Ltd, Logica UK Ltd, and Schering Agrochemicals Ltd for permission to republish it. Knowledge Craft is a registered trademark of Carnegie Group Inc., Five PPG Place, Pittsburgh, PA 15222. Oracle is a registered trademark of Oracle Corp.

References

Abarbanel, RM and Williams, MD, 1986. "A Relational Representation for Knowledge Bases", Proc. First International Conf on Expert Database Systems, Kerschberg, L ed, Reading, Massachusetts: Addison-Wesley, p 139

Atkinson, MP et al., 1984. "The Proteus Distributed Database System, Proc. Third British National Conf on Databases (BNCOD-3), Longstaff, J, ed, Cambridge: Cambridge University Press, pp 225–245

Atkinson, MP and Kulkarni, KG, 1984. "Experimenting with the Functional Data Model" In: *Databases—Role and Structure*, Stocker, PM, Gray, PMD and Atkinson, MP, eds, Cambridge: Cambridge University Press, pp 311–338

Bancilhon F, 1988. "Object-Oriented Database Systems", Proc. 7th ACM SIGART-SIGMOD-SIGACT Symposium on Principles of Database Systems (Austin), ACM Publications, pp 1–11

Bocca, J et al., 1986. "On the Evaluation Strategy of EDUCE", Proc. ACM SIGMOD86Conf (Washington), Zaniolo, C, ed, ACM, pp 368–378

Brachman, RJ and Schmolze, JG, 1985. "An overview of the KL-ONE knowledge representation system", *Cognitive Science* (9) pp 171–217

Bundy A, 1984. *Computer Modelling of Mathematical Reasoning*, New York: Academic Press

Campbell, CW and Gray, PMD, 1988. "PROOFS: an experimental Frame-Based Programming System implemented in Prolog", Proc. UK IT'88 Conf (Swansea), IEE Publications, PO Box 26, Hitchin, Herts, pp 143–147

Codd, EF, 1979. "Extending the Relational Model to Capture More Meaning", *ACM Trans Database Systems* (December 1979), p 397

Dayal, U and Smith, JM, 1986. "PROBE: A Knowledge-Oriented Database Management System" In: *On Knowledge Base Management Systems*, Brodie, ML and Mylopoulos, J, eds, New York: Springer-Verlag

Fox, MS and McDermott, J, 1986. "The Role of Databases in Knowledge-Based Systems" In: *On Knowledge Base Management Systems*, Brodie, ML and Mylopoulos, J, eds, New York: Springer-Verlag

Freeston, M, 1988. "Grid files for efficient Prolog clause access" In *PROLOG and DATABASES*, Gray and Lucas, eds, New York: Ellis-Horwood (Wiley)

Gallaire, H, 1983. "Logic Data Bases vs Deductive Data Bases", Proc. Logic Programming Workshop 1983, Pereira, LM, ed

Gray, PMD, 1982. "Use of Automatic Programming and Simulation to facilitate operations on Codasyl Databases" In *State of the Art Report DATABASE*, Series 9 No. 8, Atkinson, MP, ed, London: Pergamon Infotech, pp 346–369

Gray, PMD, 1984. *Logic, Algebra and Databases*, Chichester: Ellis-Horwood

Gray, PMD, 1984a. "Interfaces", Proc. 2nd Alvey Workshop on Architectures for Large Knowledge Bases (Manchester), Lavington, S, ed, IEE *Publications, PO Box 26, Hitchin, Herts.*

Gray, PMD, 1986. "Expert Database Systems", *AISB Quarterly*, **59**, pp. 22–23

Gray, PMD, Moffat, DS and Paton, NW, 1988. "A Prolog Interface to a Functional Data Model Database" In: *Advances in Database Technology—EDBT'88*, Schmidt, JW, Ceri, S and Missikoff, M, eds, New York: Springer-Verlag, pp 34–48

Gray, PMD, Storrs, GE and du Boulay, JBH, 1988b. "Knowledge Representations for Database Metadata", *Artificial Intelligence Review* (2), pp 3–29

Gray, PMD, 1988. "Expert Systems and Object-Oriented Databases: Evolving a new Software Architecture" In: *Research and Development in Expert Systems V*, Kelly, B and Rector, A, eds, Cambridge: Cambridge University Press, pp 284–295

Gray, PMD and Lucas, RJ, 1988. *PROLOG and DATABASES: Implementations and New Directions*, New York: Ellis-Horwood (Wiley)

Harland, DM, 1988. *REKURSIV Object-Oriented Computer Architecture*, New York: Ellis Horwood (Wiley)

Howells, DI, Fiddian, NJ and Gray, WA, 1988. "A source-to-source meta-translation system for database query languages—implementation in Prolog" In: *PROLOG and DATABASES*, Gray and Lucas, eds, New York: Ellis-Horwood (Wiley), pp 22–38

Jarke, M and Vassiliou, Y, (1984). "Coupling expert systems and database management systems" In: *Artificial Intelligence Applications for Business'*, Reitman, W, ed, New Jersey: Ablex, pp 65–86.

Kaplan, SJ, 1979. "Cooperative Responses from a Portable Natural Language Data Base Query System", PhD thesis, Moore School, Univ of Pennsylvania, also Stanford Heuristic Programming Report HPP-79-19

Kerschberg, L, ed, 1984. *Expert Database Systems: Proc. First International Workshop (Kiawah Island)*, New York: Addison-Wesley

Kerschberg, L, ed, 1986. *Proc. First International Conf. on Expert Database Systems (Charleston)*, New York: Addison-Wesley

Kerry, R, 1988. "Integration of Expert Systems and Databases", Report 29, Information Systems Engineering Divison, CCTA, Norwich

Larson, JA et al., 1985. "ATOZ: A Prototype intelligent interface to multiple information systems", Proc. IFIP Working Conf. on the Future of Command Languages, Rome: IFIP

Lucas, RJ, 1988. *Database Applications using Prolog*, New York: Ellis-Horwood (Wiley)

Lucas, RJ, 1989. "The Three Dimensional SpreadSheet (Deduc-Table)", Proc. 7th British National Confce on Databases (BNCOD 7), Williams, MH, ed, Cambridge: Cambridge University Press

Lunn, K and Archibald, IG. 1988. "TREQL (Thornton Research Easy Query Language) an Intelligent Front-End to a Relational Database" In: *PROLOG and DATABASES*, Gray, PMD and Lucas, RJ, eds, New York: Ellis-Horwood (Willey, pp 39–51

Maier, D, Stein, J, Otis, A and Purdy, A, 1986. "Development of an Object-Oriented DBMS", Proc. OOPSLA'86 Conf., ACM Publications

O'Brien, P, Bullis, B and Schaffert, C, 1986. "Persistent and Shared Objects in Trellis/Owl", Proc. International Workshop on Object-Oriented Database Systems, Dittrich, K and Dayal, U, eds, IEEE Computer Society, pp 113–123

Paton, NW and Gray, PMD, 1988. "An Object-Oriented database for storage and analysis of protein structure data" In *PROLOG and DATABASES*, Gray, PMD and Lucas, RJ, eds, New York: Ellis-Horwood (Wiley), pp 251–266

Ramamohanarao, K and Shepherd, J, 1986. "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Proc. 3rd International Logic Programming Conf., Shapiro, E, ed, New York: Springer-Verlag, pp 536–543

Rosenthal, A et al., 1986. "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", Proc. ACM SIGMOD86 Conf. (Washington), Zaniolo, C, ed, pp 166–176

Selinger, PG et al., 1979. "Access Path Selection in a Relational Database Management System", Proc. ACM SIGMOD79 Conf., Bernstein, PA, ed, pp 23–34

Stocker, PM and Cantie, R (1983). "A Target Logical Schema: the ACS", Proc. 9th VLDB Conf. (Florence)

Stonebraker, ML, 1986. "Inference in Data Base Systems Using Lazy Triggers" In: *On Knowledge Base Management Systems*, Brodie, ML and Mylopoulos, J, eds, New York: Springer-Verlag

Tsur, S, 1988. "LDL—A Technology for the Realization of Tightly Coupled Expert systems", *IEEE Expert* **3** (Fall 1988), pp 41–51

Tsur, S and Zaniolo C, 1986. "LDL: A Logic-Based Data Language", Proc. 12th International VLDB Conf. (Tokyo), Morgan Kauffman

Ullman, JD, 1985. "Implementation of Logical Query Languages for Databases", *ACM Trans. Database Systems* **10** 289–321

Warren, DHD, 1981. "Efficient Processing of Interactive Relational Database Queries expressed in Logic", Proc. 7th VLDB Conf. (Cannes), pp 272–281

Zaniolo, C, 1986. "Safety and Compilation of Non-Recursive Horn Clauses", Proc. 1st International Conf. on Expert Database Systems, Kerschberg, L, ed, New York: Addison-Wesley