THE IMPLEMENTATION AND USE OF A LOGIC BASED APPROACH TO ASSIST RETRIEVAL FROM A RELATIONAL DATABASE

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy

by Perry Jones

MAY 1988

ABSTRACT

As the use of database systems has become more widespread there have been increasing demands for easier and faster access to the stored information. Enhanced processing power and improved networking capabilities have brought this potential closer to the user than ever before, but many of those who can now benefit from these extensive information sources are not computer experts and may make only infrequent use of computer systems.

A user who is not a computer specialist finds input in a formal language, however well defined, sufficiently repugnant to discourage him from using the system [King 1977]. To enable full advantage to be gained from the opportunities available we need to provide intelligent front ends (IFE) [Bundy 1984] to cater for those who are unfamiliar with the computer system, or who are inexperienced or infrequent users of the database structure.

To facilitate the provision of such IFE we have considered the application of a logic based representation of the domain. By providing the system with a better understanding of the domain we are able to shift more of the burden of retrieval specification from the user onto the system.

ACKNOWLEDGMENTS

I would like to sincerely thank my supervisor Professor Michael Shave for his continued help and encouragement. I am also indebted to the members of staff in the department of Computer Science at Liverpool University for their assistance.

I would like to acknowledge the funding support given to me by the Science and Engineering Research Council of Great Britian.

To my parents,

Marie and Derek

CONTENTS

CHAPTER - 1

INTRODUCTION

1.1 - The demands made on information systems

Computer art is one of the few computing activities for which the output has its own intrinsic value. The value of most other computing activities is derived only from the value which someone or some organisation can gain from their use. Such is the case with the storage of information in a database, where the value to be derived is only that from the application of the data. Thus the raison d'etre for database systems is to provide a service for the benefit of the end user. Enhanced processing power and improved networking capabilities have made this service, and the benefits which can be gained from it, more widely available to the user than ever before. The benefits which can be derived from the use of a database management system are fully described by Date [Date 1986]. We briefly summarise these advantages as:

The data can be shared

Standards can be enforced

Integrity can be maintained

Redundancy can be reduced

Inconsistency can be avoided (to some extent)

Conflicting requirements can be balanced

As individuals and organisations become more aware of the benefits to be gained from the available information sources, so their demands for improved productivity from the information systems increase. The term "productivity" describes the value which can be derived by an enquirer from the reply he receives. The productivity is therefore dependent on several factors

ease of availability

If the costs of obtaining information are greater than the benefits to be derived from having the information, then an individual would choose not to access the database and so the benefits that an organisation can gain from its database system are reduced.

query time

This is measured from the time of the enquirer needing the information through to the time a final reply is received. Query time is an important performance factor in a situation where out of date information is useless

quality and usefulness of the reply

The database may hold the required information but not in a form suitable for the enquirer. For example, a database might respond to an enquiry with a vast mass of statistics about a subject when all the user wanted was a simple interpretation of the data. This problem is referred to as "information overload" [Wiederhold 1986].

1.2 - Naive users

Many of the users who can now benefit from these extensive information sources are not computer experts and may make only infrequent use of computer systems in general or database systems in particular.

Such users form an important proportion of the new class of database end users. They can be described briefly as **naive** or casual database users. The term "naive user" will be used in this thesis as a form of shorthand to include all members of this new category of users. Cuff [Cuff 1984] characterises these users as:

> Casual users do not work regularly and frequently with the system. They tend to forget details about it and to retain only a few simple concepts.
>
> They tend to make errors easily. The more opportunities for error the more errors are made.
>
> They may not know or remember how the database is organised. A casual user should not need to know how to navigate through relations, or to talk about relation or attribute names.
>
> Often casual users will have little or no programming skill.
>
> Casual users tend to forget to fill in all the details of what they want from a database query.
>
> What evidence there is on the matter suggests that casual users do not form complex queries. Although the evidence is hardly conclusive, it lends credence to the view that many casual users will wish to pose only queries that could be stated simply in English. More complicated problems will be tackled, if at all, through consecutive simpler queries.

Enabling the naive user to access an information system directly eliminates the need for a database expert to act as an intermediary between the user and the system. This improves the availability of the system, and reduces the time taken for a user to receive the information he requires. Direct access of database systems by such naive users can significantly improve the overall productivity that an organisation can derive from its stored information. However, to be able to access a system directly a naive user has to be given some form of formal training in the use of that system. The cost of providing such training for naive users is high and may outweigh the available benefits.

In an extreme case a "naive user" may be totally inexperienced in the use of computers, and therefore require training in all aspects of the systems, that is the computer system, the database management system and the specific database model. In other cases only one or two of these aspects may be unfamiliar. The time taken to educate a user is not only the time for the initial training period but also includes time for any necessary repetitive relearning. This is very relevant for naive users who use the system so infrequently that they tend to forget many of the details relating to its use and so may require continual retraining. Such retraining may be in the form of self instruction where the user may consult training or reference manuals.

1.3 - The impact of the relational database model

An important development which has had major consequences in simplifying to some extent the use of database systems is the relational database model. It was hoped that the relational database model with its "structural simplicity" [Codd 1982] and its easy to understand visual conceptualisation would enable enquirers to access the database system directly and thereby improve the availability of the information and the benefits from its use. Writing about the introduction of the relational database Codd [Codd 1983] stated that

> "in many cases, end users can now handle their own problems by direct use of the system instead of using applications programmers as mediators between them and the system."

It is true that the relational model is easier for the user to comprehend than complicated network models. However, serious difficulties remain for naive users, who are still required to possess an understanding of the syntax of relational systems and their query languages, and of the composition of the particular relational model used in the application. Relational databases, unlike network databases, shield their users from the details of storage mechanisms. Hence there is no need for the user to "navigate" a relational database in the sense in which this term has been used in the past. Nevertheless a detailed level of understanding is required if users are to specify their queries in a formal query language and to direct their queries through the structures of the relational model.

Rajinikanth and Bose [Rajinikanth & Bose 1986] attribute many of the problems encountered with the use of the relational model to its limited functionality and state

> "The two main limitations of the relational model are its semantic scantiness and lack of any kind of deductive capability"

It is our aim to simplify the retrieval process by augmenting the relational model with new features which incorporate additional semantic knowledge. The use of the semantic knowledge helps to alleviate the existing limitations of the relational model, and also reduces the complexity involved in using the system and the need for the naive user to have prior knowledge of the system before using it.

1.4 - The cost benefit ratio in database use

As we have already stated, a major constraint on the expansion of information systems and on the benefits which can be derived from them is the need for user training. The time required to train a user (Tt) can be considered some function f_t of the time (Et) that has elapsed since the user last issued a request, combined with the user's overall computer literacy (Cl) and the complexity involved in using the system (Sc). Thus:

$$Tt = f_t(Et, Cl, Sc)$$

Obviously the greater the elapsed time the more likely it is that the naive user has forgotten the previous training and so will require more retraining.

The cost of training a user (Tc) is comprised of the actual costs incurred for providing training (Ac), plus the costs incurred from the loss of productivity while the user is being trained (Ip). The value of Tc is also proportionally effected by the time taken to train a user. Thus:

$$Tc = f_c(Tt, Ac, Ip)$$

Assuming we have a measure of the gross absolute benefit (Ab) or improvement in an individual's performance due to the use of the information extracted from the system then we can define the Marginal Query Benefit (MQB) as the net benefit for a single query and the Aggregate Query Benefit (AQB) as the summation of the MQB. Thus:

$$MQB = Ab - Tc$$

$$AQB = \sum MQB$$

We can illustrate the typical nature of these functions in the form of the following two graphs

ABSOLUTE BENEFIT AND TRAINING COST PER QUERY

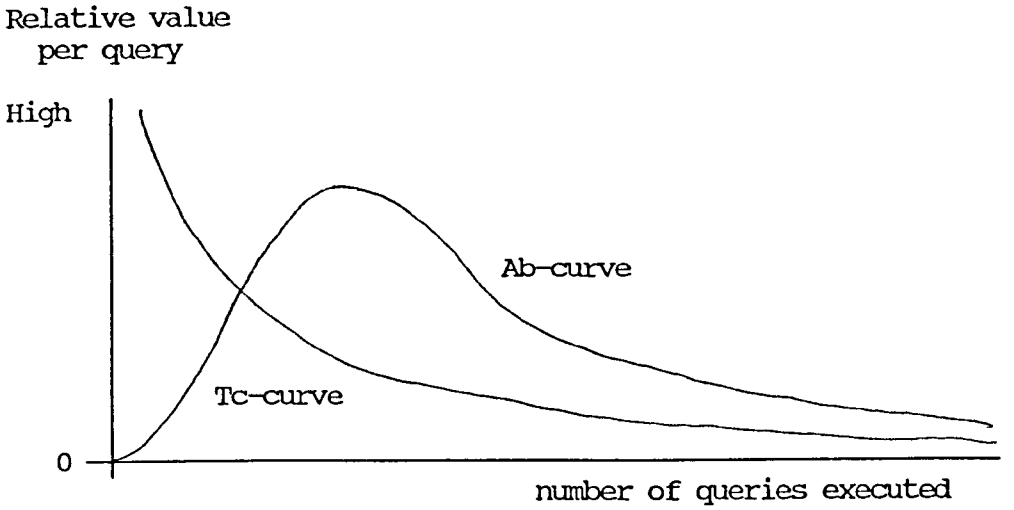


figure - 1.1

AGGREGATE AND MARGINAL QUERY BENEFIT

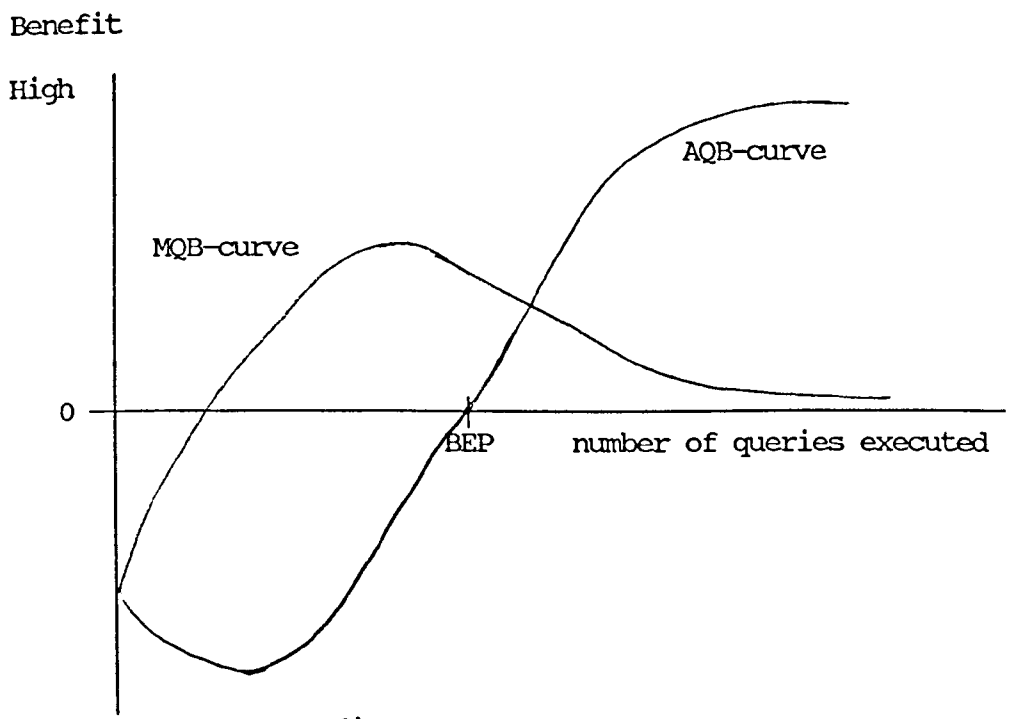


figure - 1.2

In figure 1.1 the shape of the training cost curve (Tc) represents the notion that, as a user uses the system more, the relative cost of training him for an individual query becomes less. The shape of the absolute benefit curve (Ab) in figure 1.1, shows the initial increase in absolute benefit that a user can derive from querying the system as he becomes more experienced in the use of the system. The shape of the Ab curve shows also that the absolute benefit the user can derive per query, may diminish if the system becomes so accessible that the user begins to issue requests which lack necessity and so have a lesser benefit value.

Due to the initial training costs there is inevitably a brief period when the net aggregate query benefit is negative. For frequent users of the database who issue a high number of queries, the marginal benefit easily outweighs the marginal cost and so the MQB is positive. However, for naive users who execute a low number of queries the relatively high cost of training may outweigh the query benefit so there is a possibility that the MQB will be negative.

The marginal query benefit curve (MQB) and aggregate query benefit curve (AQB) shown in figure 1.2 are drawn from the typical Tc and Ab curves illustrated in figure 1.1. When the AQB is positive then there is a corresponding positive benefit to be gained from the use of the information system. However, if the AQB is negative then the costs of providing training outweigh the benefits to be gained. This implies the existence of a break even point (BEP). The lower the

value of the BEP the more beneficial it is to train naive users, and the greater the value which can be gained from using the database.

The total benefit for an organisation is the summation of these aggregate benefits for its members. To improve the overall benefit of the system we therefore need to increase the MQB. By raising each MQB curve we also raise the AQB curve which correspondingly reduces the value of the BEP

There are two main approaches to increasing the value of the MQB. The first approach attempts to increase the absolute benefit by improving the quality or usefulness of the reply. This has the effect of raising the Ab curve (figure 1.1) which correspondingly raises both the MQB and AQB curves (figure 1.2). The second approach is to reduce the cost of training. The cost of training can be reduced by reducing the time required to train users, which itself can be reduced by simplifying the complexities involved in using the system. Reducing the cost of training causes the Tc curve (figure 1.1) to be lowered which has the effect of raising both the MQB and AQB curves (figure 1.2).

In this thesis we will concentrate on the latter aspect, but it will be shown that the technique used to eliminate certain complexities for the user will also enable the database system to reflect more clearly the semantics of its data.

1.5 - The need for a user assistant

Sophisticated software, which acts as an assistant to the enquirer, can be used to shield the user from the complexities encountered when using an information system. Such software is commonly referred to as an intelligent front end (IFE) [Bundy 1984]. The provision of intelligent front ends reduces the complexity of using database systems and correspondingly simplifies the training required for naive users.

An IFE for a database system helps the user to formulate his queries. It creates a conceptual stepping-stone which allows stage translations or the refinement of a user's request into a formalised query (see figure 1.3).

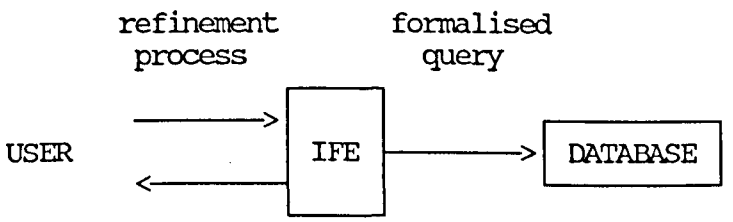


figure 1.3

The IFE assumes much of the responsibility for the specification of the retrieval. This reduces the complexity of using the system (Sc) and hence simplifies query specification, decreases training time (Tt), and allows greater benefit to be gained from using the database system.

To facilitate the provision of such user assistants we have considered the use of a logic based representation of the domain. Such a representation provides the system with a better "understanding" of the domain semantics. Using this representation we are able to create domain specific interfaces which shift much of the burden of retrieval specification from the user onto the system.

1.6 - An overview of the thesis

In this chapter we have outlined briefly the benefits and costs of enabling naive users to retrieve information directly. In chapter two we outline in detail the problems encountered by such naive users, giving specific examples relating to current database management systems. We also describe several of the systems which have been proposed to assist users in the retrieval of information.

In chapter three we describe how the application of a logic based representation of the domain can help to alleviate many of the problems previously outlined.

In chapter four we identify the user view and the general concepts of a user view which we wish to represent. This chapter also shows how the incorporation of greater semantic representation in the design of the relational model can simplify the task of the user assistant.

Chapters five, six and seven report on the practical work that has been carried out to implement a user assistant which invokes a logic based representation of the domain semantics. Chapter five outlines the practical ways of connecting an IFE to a database management system, chapter six describes the internal architecture on an IFE and chapter seven reviews the operation and user dialogue of the user assistant.

Finally, chapter eight summarises the implementation of the user assistant and outlines the ways in which this system can be improved. We conclude the chapter by considering the long term improvements to both database systems and expert systems which can be derived from the coupling and combination strategies outlined in this thesis.

CHAPTER - 2

QUERYING A DATABASE SYSTEM

A variety of query systems are available for the retrieval of information from databases. Many of these systems offer powerful and flexible facilities which can be skillfully exploited by experienced and regular users. However, the scope and complexity of these systems have caused considerable difficulties for naive users when querying database management systems and have thus created barriers to the accessibility of database systems. This chapter describes these problems and reviews several systems which have been proposed to help alleviate these problems by simplifying the retrieval process. First, the general characteristics of such systems are reviewed, then each system is described individually in depth. Finally the problems which we believe are still outstanding for the naive user are discussed, along with the difficulties associated with the implementation of such retrieval systems.

2.1 - THE DEMANDS ON INTERACTIVE QUERY LANGUAGES

The standard, most efficient method for retrieving information from a database is for a user to specify his request in a formal query language. All current database management systems provide formal interactive query languages. These languages are designed to be used by the three classes of database user, namely the database

administrator (DBA), the application programmer and the end user. These languages must permit the expression of all of the system operations that the members of each of the user groups may wish to perform. These languages can therefore be considered as being "all things to all men".

The query language can generally be used either interactively or by embedding the command statements in a high-level programming language such as COBOL, PL/1 or PASCAL. It is more usual for applications programmers and database administrators (DBA) to use the embedded form of the language and for end users to use the interactive form.

The query languages must provide the users with the ability to express all of the basic database manipulation tasks. These tasks as described by Date [Date 1986] are:

Adding new files
Inserting new data into existing files
Retrieving data into existing files
Updating data into existing files
Deleting data into existing files
Removing existing files

The provision of such basic tasks is imperative for the successful operation of any database management system, both for the maintenance as well as the querying of the database system. Languages

which provide full manipulative facilities can be considered as Full query languages. These query language are formally defined and have a rigidly enforced syntax which ensures that the specification of a task is not ambiguous.

The specification of an operation in these Full languages may be quite complex and so the user is required to undergo formal training in the use of the system. As stated in section 1.4 such training diminishes the benefits to be derived from the use of the database system.

The function of these languages is quite satisfactory for the purpose of mass data management or regular standard operations that tend to be performed by DBAs , applications programmers or end users who are skilled in the specification of queries. However, for infrequent or inexperienced users the complexity involved in specifying query operations creates barriers to retrieval, even when the task to be performed appears simple and straightforward. As stated by King [King 1977]

> "a user who is not a computer specialist finds input in a formal language, however well defined, sufficiently repugnant to discourage him from using the system"

2.2 - THE BARRIERS TO QUERYING ENCOUNTERED BY NAIVE USERS

Naive users attempting to express their queries in a form which the system will understand are faced with two main problems. The first is the need to specify a query strictly in accordance with the syntax of the database query language. The second is the requirement that the user must be aware of the actual "relational composition" [Cuff 1984] of the database model. This is required to enable the user to navigate the relational structures to resolve his query. This navigational process is equivalent to a semantic match .i.e. the user is matching his interpretation of the world against the relational model. The explicit semantic information for the domain is not represented in the relational model, instead it is held implicitly in the relational structures. A further problem is that a real world situation may frequently have more than one possible relational representation.

These restrictions can easily be illustrated by considering the specification of a simple example query when translated into the query languages of several of the currently available database systems.

2.2.1 - SQL

SQL [IBM 1984 DATE 1986] is now widely regarded as the standard query language for relational database systems. Retrieval using SQL is based on the SELECT command. By considering the following apparently trivial query we are able to illustrate the problems encountered by naive users when using this language.

Fetch the names of all the students who are taught by Prof Smith.

When coded in SQL for the relational model given in Appendix A this is expressed as:

```
SELECT STITLE, SFIRSTNAME, SSURNAME FROM STUDENT WHERE
 SNUM IN ( SELECT STUDENT
           FROM ATTEND
           WHERE
            COURSE IN ( SELECT CODE
                        FROM COURSE
                        WHERE
                         LECTURER = ( SELECT LNUM
                                      FROM LECTURER
                                      WHERE
                                       LTITLE = 'PROF'
                                      AND
                                       LSURNAME = 'SMITH')))
```

As with all of the interactive query languages the user is required to know the syntax of the language for query expression and be fully aware of the relational model. The problem of semantic navigation or semantic matching of the query to the relational structures can be seen if we explicitly represent the implicit semantics of the nesting structure for the coded SQL command.

```
student is taught by a PROF SMITH if
    ( student attends course
        ( course given by lecturer
            ( lecturer name is PROF SMITH  )))
```

It should not be the responsibility of the naive user to navigate the database structures. Query language specifications can obviously become vastly more complex when navigating through anything other than a trivial "toy" model.

The use of facilities such as 'UNION' although imperatively useful for data retrieval can also increase the complexity of the query expression, thereby increasing the level of difficulty for query specification by the naive user.

2.2.2 - QUEL

QUEL [Stonebraker 1980, Epstein 1981] is the query language used by the INGRES database system. In QUEL the basic retrieval function is the RETRIEVE command. Reconsider the previous query

> Fetch the names of all the students who are taught by Prof Smith.

This query can be coded in QUEL for the relational model given in Appendix A as:

```
RANGE OF S IS STUDENT
RANGE OF A IS ATTEND
RANGE OF C IS COURSE
RANGE OF L IS LECTURER
RETRIEVE ( S.STITLE, S.SFIRSTNAME, S.SSURNAME )
WHERE
 S.SNUM = A.STUDENT AND
 A.COURSE = C.CODE AND
 C.LECTURER = L.LNUM AND
 L.LTITLE = "PROF" AND
 L.LSURNAME = "SMITH"
```

Again the enquirer is responsible for expressing the query in a form which corresponds to relational structures. The semantic navigation of the database structures is if anything more difficult in QUEL than in SQL as the simple user notion that "a STUDENT attends a COURSE" requires the obscure specification

```
S.SNUM = A.STUDENT AND A.COURSE = C.CODE.
```

QUEL like all of the other full query languages gives us great flexibility in expressing our queries but this power is wasted on the simple queries of the naive user.

2.2.3 - RAPIDE

RAPIDE [LOGICA 1984] is not a full query language as it only supports interactive retrieval, insertions, updates and deletions. To illustrate the complexity of using RAPIDE we will reconsider our previous example query. This query when coded in RAPIDE becomes:

```
SEARCH STUDENT
   SEARCH ATTEND STUDENT = STUDENT.SNUM
      SEARCH COURSE CODE = ATTEND.COURSE
         SEARCH LECTURER LNUM = COURSE.LECTURER AND |
                         LTITLE = 'PROF' AND |
                         LSURNAME = 'SMITH'
         SHOW STUDENT
         ENDSEARCH
      ENDSEARCH
   ENDSEARCH
 ENDSEARCH
EXECUTE
```

"|" is the line continuation marker

A naive user would find extreme difficulty in expressing queries in this form, as once again the user is responsible for both structuring the retrieval request and understanding the data model. The responsibility for the user to control the execution of the request is greater with RAPIDE than with the other full query languages. When executing a nested loop where a file appears in both an inner and outer loop then it is the user who must ensure that the retrieved tuples do not over write each other. Such system activities, although necessary to ensure correct retrieval, unnecessarily complicate the retrieval process and prevent naive users from having direct access to the system even though such interactive systems have been designed for end user access.

2.2.4 - Query by example

Query by example (QBE) [Zloof 1977] is both a full query language and a user interface system. QBE with its graphical form attempts to simplify the retrieval process and removes many of the syntactic restrictions which are common to the other languages.

As with all the interactive query languages QBE comes in two forms the first is the familiar two-dimensional tabular form with the skeleton representation of the tables. The second form of QBE is a linear one which can be inserted into other programming languages such as PL/1 and APL [Bontempo 1984].

Applying our previous example query to the QBE system we gain the following screens

STUDENT	STITLE	SFIRSTNAME	SSURNAME	SNUM
	P.	P.	P.	_SN

ATTEND	STUDENT	COURSE
	_SN	_CC

COURSE	CODE	LECTURER
	_CC	_LN

LECTURER	LTITLE	LSURNAME	LNUM
	PROF	SMITH	_LN

However, even for this fairly simple example query the user still needs to have a considerable knowledge of the syntax of the query expression. In addition the user is still required to have a full understanding of the relational model of the database. If he wishes the user can obtain information defining the relational structures but this does not include the semantic interpretation of the structures.

In summary all database systems require comprehensive query languages so that they may function correctly. These query languages although available for interactive use by the end users are still complex enough for the users to require a degree of formal training. Such training needs to include instruction both in the expression of syntactically correct queries which conform to the query language syntax, and also in semantically correct queries that conform to the semantics which are represented by the relational structures of the database model.

2.3 - DEDICATED RETRIEVAL SYSTEMS

To simplify the retrieval process and so reduce the need for the formal training of users, specialised systems have been developed which are in general dedicated to the retrieval task and ignore the other five manipulative tasks. These retrieval systems can be categorised by their architecture or alternatively by their interfacing strategy. The latter aspect is described in section 2.4. These systems are either enclosed self-contained retrieval systems

which have their own internal database, or the retrieval system is a user front end interface which "sits" on top of an external database management system and shields the user from many of the complexities which arise when directly accessing the database.

Self-contained retrieval systems lose many of the benefits associated with using a database management system [Date 1986] The main advantage of using a self-contained system is that it greatly simplifies prototype development. However, the small size of the database can create a false impression of the system's performance. This often occurs when the retrieval system depends on performing a full search of the entire contents of the database every time it parses a query. This activity would render such systems useless if applied to a large external database.

The motivation behind the development of these dedicated retrieval systems is to simplify database usage and so increase the accessibility of database systems for naive users. To design and maintain two entirely separate systems, one for the experienced user and a second for the non-experienced user would clearly be wasteful and lose much of the benefit to be gained from increased use of the stored information [King 1977]. Also the duplication of data would eliminate many of the advantages of having a centralised database.

A single system which can cater for both experienced and naive users can also be wasteful. Such waste is caused by the additional processing performed by the system to understand an experienced user's

query, where the user is content with and efficient at expressing his queries in a powerful and flexible formal query language. This wastage can be eliminated if the experienced user is given the option to access the database system either via the user assistant or the database's own query language.

2.4 - Strategies for interactive systems

As we stated above dedicated retrieval systems can also be categorised by the interfacing strategy they adopt. There are several possible strategies for interaction with the user, which fall into four major categories: natural language; limited language; prompted or menu driven and graphical.

2.4.1 - Natural language systems

The optimal interface system would be one where the user could enter his query in the form that he conceptualises it. Kummel [Kummel 1979] translates Lakoff's [Lakoff 1971] statement

> "all thinking which takes place in the human mind functions in Natural Language".

Therefore it might be thought that to understand natural language is to understand the "natural logic" and meaning of the words. Yet such systems which purport to understand natural language, (Eliza [Weizenbaum 1966]), are exceedingly shallow in their understanding of the underlying meaning.

For most database systems the domain of interest is very limited in relation to the real world as a whole. Hence there is less need for a query system to provide full natural language capabilities. As Wallace [Wallace 1984] states

> "computers can cope with so few of the functions of natural language that nobody has ever dared claim to have written a real natural language understanding system"

When we refer to natural language understanding systems for database systems we are really referring to limited language understanding systems [Kelly 1977].

2.4.2 - Limited language systems

All of the fears expressed about the over expectation of system performance by the system's users [Gevarter 1983] will hopefully be allayed if the interfacing system functions "responsibly". This means that a system which encounters a term or phrase which it does not understand should not simply ignore the unknown item and continue its attempt to translate the language fragments that it does recognise. Instead the system should gracefully reject the query or consult the user for further explanation to help it understand the offending term or phrase. It is not a responsible course of action for the system to assume that an item it does not recognise has no importance and so can be ignored in the query translation. Therefore to function responsibly the system must be able to recognise what is outside its domain of knowledge and take the appropriate action.

All of the systems outlined in section 2.5 although referred to as natural language systems are in fact limited language system.

2.4.3 - Prompted / Menu driven systems

Systems adopting this type of interaction strategy have great potential for small applications. This type of interaction uses only a fraction of the time and processing power required for natural language systems and so is ideal for use on systems with very limited resources. In Cuff's [Cuff 1984] justification of menu-based query systems he cites the problems of a natural language system as outlined by Gevarter [Gevarter 1983]

1 It encourages an unrealistic expectation of the system's power

2 The linguistic limitations of such a system are not as well defined as they are with a formal language. They can appear sporadically and unexpectedly, when the system rejects an unknown word or a grammatical construction, or when it lacks background knowledge.

3 NL's richness often makes sentences ambiguous. One has to rely on the implementation being prepared to consider all possibilities, a situation that does not arise with a formally-defined language.

4 Because much of the vocabulary and knowledge that people want to use when querying a particular database may be specific to it, an NL system has to be partly recast for each domain of discourse.

5 There are several technical problems, such as anaphora and ambiguity, requiring effort in areas that do not concern more formal systems. If it is to have an acceptable interface, an NL system is inherently more complicated to implement If too much of this burden is put back subtly on to users (as, for instance, in lengthy attempts to remove ambiguity via dialogue), they may react against it.

It is important to remember these points when considering natural language systems. We have already mentioned the concern about over expectation by users of the system.

The concern shown about the inherent problem of ambiguity in natural language is justified if such languages are to be used to specify queries. To counter this problem the system should recognise and respond "responsibly" if it encounters an ambiguously specified query. It is important to remember that when attempting to understand natural language we are also attempting to understand a user's own concepts as he perceives them [Lakoff 1971]. If the concepts in the users communication are ambiguous then it is of benefit to the user if the system enters some form of clarification dialogue which will make the user aware of the ambiguities and hopefully enable them to be resolved. This would be the natural action if the user were communicating with another person.

The formal structures of a menu-based querying system severely restrict the systems flexibility for representing a user's query. However, this restriction helps eliminate any system representation of ambiguous queries, and can alleviate the need for the user to know the underlying model structure. This simple rigid mode of querying seriously limits the flexibility of query expression for the user.

As we have previously stated the domain of the database is limited but the combinations of data objects to be retrieved and the

understanding and expression of connections between objects is enormous. Thus when dealing with so many possible combinations the expression of queries, other than trivial ones, can become cumbersome when using this simple rigid mode of querying.

To partially overcome these difficulties a number of systems have been developed to combine natural language fragments in menu-based systems [Tennant 1983, Cuff 1984].

2.4.4 - Graphical systems

Probably the best known graphical query system is Query By Example (QBE). As with menu-based systems QBE also eliminates the possibility of ambiguous query expression. As we outlined in section 2.1, QBE is a full query language retrieval system so unlike the menu-based systems it provides for a greater degree of query expression. Also like all the other full query languages QBE requires the users to know the artificial syntax imposed by the system and to possess an understanding of the structures of the relational model. QBE achieves its increased expressive power over menu-based systems at the expense of increasing the syntactical complexity of query specification. Queries with other than trivial retrieval constraints require a considerable degree of syntax understanding if they are to be resolved correctly. Consider the query

fetch all employee names for employees in the toy department who earn more than 10,000 or who work in the hardware department and earn more than 20,000

EMPLOYEE	NAME	SALARY	DEPT
	P.	S1	D1

CONDITIONS
(S1,D1) = ((10000, TOY) or (20000, HARDWARE))

This is a fairly simple query but the difficulty arises as the expression does not conform to the simple one dimensional retrieval constraint that QBE is so good at handling. Once queries become anything other than trivial then their expression becomes as complicated as in any traditional type of programming language.

2.5 - A REVIEW OF CURRENT DBMS QUERY INTERFACES

Many practical implementations of the retrieval systems categorised in sections 2.3 and 2,4 have been developed. By considering several of these implementations, or proposed implementations, we are able to identify the ways in which these systems simplify the retrieval process. We are also able to identify the restrictions enforced by each system and the effect of such restrictions on both the performance and implementation of the systems.

2.5.1 - RENDEZVOUS

Rendezvous was originally outlined by Codd [Codd 1974] at IBM San Jose, California, although, the development quoted by Bates [Bates 1986] is from an IBM research report dated 1978. Rendezvous highlighted the needs of casual users for natural language type interfaces for relational database systems.

Rendezvous attempted to create a natural language dialogue based on clarification rather than the dialogues of the earlier natural language systems such as Eliza [Weizenbaum 1966] and SHURDLU [Winograd 1971]. Each of these systems involved the user in a dialogue which, although providing a medium for the exchange of information, did not "pursue" the unknown or ambiguous information so as to fully clarify the user's input. Codd described the dialogue of Eliza as a "stroking dialogue", and that of SHURDLU as a "contributive dialogue". To explain these classifications of dialogue Codd used the following example statement

> I went to a downtown dealer yesterday and bought a very expensive car. Afterwards, I felt I should not have done it.

Then described the different dialogue responses

STROKING: I am sorry you have feelings of regret. What will you do now?

CONTRIBUTIVE: I also bought a car recently and discovered it has a very high gas consumption.

CLARIFICATION: A downtown dealer? Which one, and what do you mean by 'very expensive'?

The task of clarification is vitally important in retrieval systems as the system must ensure that it fully understands the user's request if it is to adequately answer it. However unnecessary use of the clarification dialogue may mislead the user into believing that there exists an alternative. To illustrate this consider the following example of a dialogue involving clarification.

USER: Is the course 01CS taught by Prof Greene?

SYSTEM: Which Prof Greene do you mean
1 Prof Abraham Greene
2 Prof Benjamin Greene

USER: Prof Abraham Greene

SYSTEM: No Prof Abraham Greene does not teach course 01CS

Such clarification dialogue gives a degree of credibility to the alternative that the course 01CS is taught by Prof Benjamin Greene. When a clarification dialogue is used it implies that there is a need for clarification due to the positive existence of an alternative.

From a sample dialogue of a user session in Rendezvous the system appears very capable of handling the complex language structures of the users' queries, although the examples were run on a trivial model.

The system lacks any form of back-tracking and so seems unable to identify and cope with ambiguous queries which have more than one correct parse.

No reference is made to the system's understanding, representation or translation of user concepts. We are therefore led to believe that the system can not handle such user concepts as subsets or structured objects. The system appears to represent only the relational database model.

As no reference is made to the domain representation it is therefore difficult to estimate the systems transportability between different domains and database models.

2.5.2 - PLANES

PLANES (Programmed LANguage Enquiry System) [Waltz 1977] was developed at the University of Illinois. The system is a limited language front end interface which consults an extracted portion of a navy aircraft maintenance database. The system was developed for "nonhierarchic record-based databases".

PLANES natural language understanding capability is based on an augmented transition network (ATN). The system incorporates the semantic model information with the language specification. Such grammar representations where the domain is represented in the grammar are known as semantic grammars [Wallace 1984]. They make both the language understanding and the system itself dependent on a single model, and such dependency makes it difficult to transfer the systems to other different domains. Thus PLANES using a semantic grammar is dedicated to the single model of aeroplane maintenance

records. Later systems (Intellect, TQA, TEAM) deliberately separate the domain specific information from the general language information, thereby making it easier to transport the systems to different domains.

For such a limited domain the systems inability to handle ambiguous query parses was not exposed. PLANES once having found a correct parse would assume it to be correct and would not search for alternative correct parses.

Planes incorporates a spelling checker to recognise unknown words. From the example dialogue the system appeared to recognise all words that were stored in the database. This tends to suggest that the system performed an entire search of the database contents, or the contents of the database were prepared in some form of inverted file.

2.5.3 - ROBOT / INTELLECT

ROBOT (and the later commercial version Intellect) [Harris 1984, IBM 1983, A.I Corp 1980] is a limited language front end system. It allows enquirers to express queries in "normal English" as "No training is needed as Intellect speaks your language" [IBM 1983]. The system's main strength is in its natural language understanding capability. It has an excellent understanding of mathematical functions and aggregates and the system provides a facility for handling anaphora. Anaphora is the use of words relating to earlier

words and in the terms of query dialogue systems this relates to the understanding of pronouns and the specification of a query based on a previous query. e.g.

USER: Who teaches the course 01CS
SYSTEM: Prof Arthur Smith
USER: How old is he
SYSTEM: 55

Thus "he" is an anaphoric reference to "the person who teaches 01CS".

As a front end system Intellect interfaces with several current database management systems although the efficiency of such connections is not discussed in the literature. According to Tennant [Tennant 1981] the early ROBOT system used a process of inverted files to recognise and match strings to database terms. In respect of time, such an exhaustive search technique may prove costly when applied to a large database.

The system is claimed to be easily transportable between domains. However the representation of the domain semantics as perceived by the user is not described. The system does separate the domain information from the language understanding information, which supports its claim for transportability. When the domain semantics are not embedded in the language description then the resulting system is more flexible and more transportable. However the separation of the semantics from the language can lead to a loss in semantic understanding. It is stated [Wallace 1984] that the

Intellect system can not distinguish between the two questions "Who sold a car?" and "Who was sold a car?"

This shows the need for access to the semantic information when performing a query parse, the required semantic information being

SELLER sold OBJECT

BUYER was sold OBJECT

Although the language understanding component needs access to the semantic information, such semantic information does not have to be combined with the language information. Both information sources can be kept separate and so avoid the transportability problems associated with semantic grammars. Systems such as Intellect which are generalised for many domains do not have the semantics in the grammar. Instead they require sophisticated methods for handling the verb phrases.

The literature makes no reference to how the system identifies or reacts to queries which are ambiguous.

2.5.4 - RABBIT

Rabbit [Tou 1982] was developed at Xerox PARC. Rabbit is a menu-based system with natural language fragments and can be considered self-contained. The system attempted to introduce a new mode of querying. This new retrieval process was based on the human

cognitive process of "iterative refinement by analogy". The process described in RABBIT as "retrieval by reformulation" offers users instances of an answer which a user can accept or reject. If a user rejects an offering then "the system tries to encourage the user to articulate what is wrong with the instance presented". The system then proceeds to make further offerings based on the users responses.

Rabbit is implemented in the Smalltalk programming language. The system uses the KL-ONE knowledge representation language to define the conceptual structure of the domain. The system does not interface with an actual database management system instead it simply uses the KL-ONE instances.

2.5.6 - IRUS

IRUS (Information retrieval using the RUS language parser) [Bates 1983] was developed at BBN laboratories Massachusetts. IRUS is a limited language system. IRUS high-lights the need for independence of the interfacing system from both the domain model and the database system. The system achieves this independence by using an intermediate language to represent the internal parse of the query This language is known as MRL (Meaning Representation Language).

From the textual examples shown we can conclude that the system recognises every term that is stored in the database. This either means that the system operates on a small enough database so as it

can store it internally or it performs an entire search of the database.

Unlike most of the other systems IRUS does not confirm its correct translation of the user query by offering a paraphrased version of the users query instead it directly proceeds to parse it. Paraphrased queries are helpful and reassuring for the naive user.

2.5.7 - IR-NLI

IR-NLI (Information Retrieval Natural Language Interface) [Guida 1983] developed at the University of Udine Italy. The system is a limited language front end system. Although initially designed for on-line database interaction, the proposal for the first stage implementation had restricted interactive activity with the external database. This made the system more reminiscent of a self-contained retrieval system. The proposed system was "conceived for off-line use without interaction with the database" and so uses the "snapshot method" (see section 6.7.1) for data extraction.

The system was useful in natural language interface development as it addressed the major problem of "meaning understanding". The system attempted to identify the underlying user goals of the query and distinguished between the surface comprehension, that only aims at representing the literal content of the language expression, and the deep comprehension that captures the goals and intentions which lie behind the utterances. The understanding of the underlying

meaning is immensely beneficial as it allows the recognition of a semantic context which can be used to improve the systems ability to understand natural language.

2.5.8 -TEAM

TEAM (Transportable English Access database Manager) [Grosz 1983] is a natural language system. Implemented in Interlisp with a Prolog database, it can be considered a self contained system. TEAM attempted to overcome the problem that, as Grosz remarked,

> "natural language interface systems have used techniques that make them inherently difficult to transfer to new domains and databases"

To achieve transportability, TEAM separates the operational information for language, domain and database model. This simplifies the process of changing or restructuring the database model. By maintaining a separate language component, transfer between domains is also simplified.

Pursuing the independence ideal TEAM has its own in-built domain knowledge acquisition routines. This provides partial automation of the process of transferring the system between domains.

2.5.9 - A Robust Portable Natural Language Interface

Developed by Ginsparg [Ginsparg 1983] at Bell Laboratories and implemented in Franz Lisp, this system is aimed at improving the problems of independence and portability. Unlike TEAM the independence is from a particular database rather than a domain model.

The natural language understanding component of the system uses semantic nets. The system demonstrates a characteristic not shown by other systems of dividing the query into distinct "concepts". These concepts can be executed separately. This allows the system to isolate and report on any false concepts or assumptions made by the user, e.g.

> Does supplier X supply parts for projects located in London

The system is able to respond

> there are no projects located in London.

The system has identified that the user query is based on the assumptions that there exists a supplier X and that there exist projects which are located in London. If the system ignores such user assumptions and simply gives the response NO, Then the system's apparent recognition of the assumption, that projects are based in London, is given a degree of credibility. The use of sub concepts, though vastly improving the user interactions, make interfacing to an external database management system more complicated, as the

system itself has to receive and act upon the intermediate results generated by the queries of the sub concepts.

The external database interface component (DBAP), which is also written in Franz Lisp, is designed to generate a query in an "augmented relational form". This query can then be translated into the database management systems own query language. We are unsure about the use and scope of this system as the only reference to a "real database" is one that has been "abstracted" from an on-line system. This appears to suggest that the system does not interact with a database on-line and so vastly simplifies the execution of the sub concept queries,

2.5.10 - HERCULES

HERCULES (Heuristic Retrieval; a Casual User Language System) [Cuff 1984] was developed at Essex University. The system which is implemented in ULISP, is a menu-based system which has natural language fragments.

The system gives the user a framework in which to place his natural language query fragments. This simplifies the task of natural language understanding as the conditions or statements are already associated with database terms which the system understands. The system explicitly depicts all facts known about an entity type. This includes all attribute and connection names. By explicitly depicting

all of the possible connections the system eliminates the need for the user to navigate the database.

As with all menu-based systems one is limited by the physical size of the screen, and so although it worked well for the very small example database we remain sceptical about the application of menu-based systems to the querying of larger and more complex models. Menu-based systems lack the superior expressive power of natural language systems. This limited ability for query expression causes difficulty when a query links distinct conditional clauses in some convoluted expression involving "and" and "or". Consider the following query

> fetch the names of people who are either over 50 years old and work in the toy department or are under 20 years old and work in the furniture department

Such queries are difficult, if not impossible, to express in systems which enforce a framework or structure for the expression of queries. Cuff identifies this limitation and states

> "HERCULES is poorly equipped in its present form to handle problems that need to connect leaves[1] by a Boolean OR"

This problem is not specific to HERCULES, as QBE suffers from a similar problem. To specify such queries in QBE the enquirer is required to explicitly state the connections between the constraint

---

[1] A leaf is a page or pane of a menu which represents an entity

clauses. To specify such connections the user must possess a good understanding of the language syntax (see section 2.4.4).

2.5.11 - QPROC / NEL

QPROC (and later NEL) [Wallace 1983, Wallace 1984, West 1986] was developed in association with ICL and the BBC. QPROC is a self-contained limited language system. NEL was the QPROC system applied to an external database management system.

QPROC was a demonstrator to show how a Prolog system could be used to decode and represent user queries. The system's language understanding component forms an internal logic representation using what are termed Descriptions and Qualifiers (D & Qs). These representations isolate the users intended meaning. The system can handle very curt query requests.

The system has two main weaknesses. First, the system is unable to deal with ambiguous parses. That is, it stops when it finds a correct parse. Secondly, the system performs a potentially disastrous action by ignoring unknown terms and proceeding to translate the fragments of the query which it does understand. Although, before executing the translated query the system does report that it has ignored certain terms.

NEL was developed to overcome the limitations of QPROC. As outlined by West [West 1986]

> "A major limitation of QPROC was that the database itself had to be implemented in Prolog and so only small toy applications could be implemented"

NEL instead used the ICL database management system Querymaster as the back-end database. NEL was used to formulate "LIST" commands for the database system. Unlike QPROC, NEL is unable to scan the entire contents of the database to match items specified in user queries with items actually stored in the database. Instead the user is required to specify all of the words, which he believes are stored in the database, in quotes. e.g.

Tell me the customers for the county of 'DEVON'

The NEL system shows its understanding of a user request by displaying a coded Querymaster command. Although an improvement on the totally obscure internal D & Q representation given by QPROC, it still confronts the user with an unfamiliar expression representation and the user is given no opportunity to agree or disagree with the proposed parse.

Both systems are unable to handle any form of aggregation.

Regarding the transportability of the NEL system, this requires a knowledge engineer to prepare both the database for the natural language enquiry and the NEL internal data model.

2.5.12 - TQA

TQA (Transformational Question Answering system) [Damerau 1985] is a natural language front end system applicable to external SQL-like databases. TQA is designed to make systems more transportable by simplifying the customization required when transferring an interface system to a new domain. TQA uses a customization program which is run by the DBA to determine certain essential information not contained in the database tables, such as synonyms and permissible subjects for verbs used in queries. TQA improves its transportability by using database-independent transformational rules, instead of the more common semantic nets. As we have previously outlined a loss in semantic representation can occur when not using semantic networks. TQA does partially suffer from this problem and Damerau outlines the work still unresolved and states a need for inheritance information to overcome the loss of the semantic representation provided by an is-a link. Damerau believes the solution to this problem will be achieved by using some form of superset-subset feature.

2.5.13 - BAROQUE

BAROQUE (Browse and Query) [Motro 1986] This an interfacing system which uses an artificial language. It is aimed at naive users who wish to browse through a relational database. The system does have a separate query mode whereby user can query the database using the formal underlying relational query language. The system uses a semantic network to represent the entire database and users are able to traverse this net. The browse system is limited to only four commands, called:

What is it?
What is known about it?
What is the connection?
Any others like it?

Such a browsing system is very useful as it allows the user to have access to the stored data and the semantic information described in the semantic representation. The system traverses the semantic net so as to offer further information relating to an object or entity. Such a mode of dialogue is similar to the "contributive dialogue" [Codd 1974].

The facility "like" which attempts to find connections between objects, simply matches object values and has no relevance to or recognition of the true semantic connections between the objects. Motro states the reason for this simplification is due to the

problem of the "connection trap". Motro then proceeds to attribute this problem to the limited semantic representation capability of the relational model.

2.6 - SUMMARY

We began this chapter by outlining the needs of the experienced database user and how these needs have resulted in database query languages becoming flexible and powerful tools for expressing queries. We then proceeded to consider the problems encountered by naive users when using such query languages and this illustrated the need for the provision of dedicated retrieval systems, which attempt to reduce the complexity involved in querying a database.

These dedicated retrieval systems were considered in general and we concluded that the front end limited language type of interface had the most potential. Such systems captured the benefits of using a database management system, and offered the simplest mode of interaction for naive users. The gains in simplification of query expression from the menu-based systems were outweighed by their deficiencies in expressing anything other than trivial user queries.

An important aspect, not included in most of the systems we have described, is the need for retrieval systems to perform meta-queries. That is, to be able to query their own semantic information. The use of semantic information is an ideal method for

explaining to the naive user, in the terms and phrases he understands, what is actually represented in the database.

Considering the implementation aspects of the systems reviewed we found a major problem to be the trade off between semantic representation of the domain and system transportability. The use of semantic nets or semantic grammars allows the system to achieve a greater understanding of the meaning of the user's request. However, because the semantics are so embedded in the language understanding, these systems become dependent on a single domain and require a vast effort to alter them for use on another domain. Conversely, the more generalised systems although easy to change to different domains have only a shallow understanding of the semantic meaning. What is required is a detailed representation of the semantic meaning of the modelled situation. This needs to be stored in a form of knowledge module which is totally isolated from the representations of the language understanding module, the module containing the relational database structure information and the actual item values stored in the database system.

The problems of implementing dedicated retrieval systems and the performance of such systems could be alleviated with better techniques for the formal representation of the semantic concepts which a user has about the model. By combining the expanding technologies of the expert system community with the existing database management systems we are able to represent explicitly much of the semantic information for the domain. In particular, it is easy to maintain

separation of the knowledge modules in a rule-based environment as the logic predicates are already distinct.

Before we start considering the implementation of such a combined system we need to consider in detail the ways in which such a combination can be employed to assist the interaction between users and database systems.

CHAPTER - 3

USING EXPERT SYSTEMS TO ENHANCE DATABASE MANAGEMENT SYSTEMS

We have stated several of the problems associated with querying databases and have described several systems that have been proposed to simplify the process of retrieval. When considering the proposed systems we identified the need for them to possess an understanding of the semantics of the domain. However, improving a system's ability to represent the semantic information of a domain had an adverse effect on the system's transportability. In this chapter we consider how these two problems can be reconciled by using an expert system as an "interface assistant" between the user and database system. We outline, in detail, the improvements to the user interaction that can be derived from such an interface assistant. We then describe the simplification of query expression and improved functionality that can be derived by expressing queries and user views in a logic form as compared with the use of a relational form.

3.1 - THE RELEVANCE OF EXPERT SYSTEMS

In this chapter we consider the use of expert systems and more specifically the use of logic to represent the model. The task we are proposing corresponds well to the definition of an expert system given by Feigenbaum [Feigenbaum 1982]

> "An Expert System is an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution."

In this case the "knowledge" is the domain knowledge describing the database model in terms of the real world and the "inference procedures" are the method of interpretation of the description. This process of interpretation can be used to relax the formality of the query language.

D'Agapeyeff [1983] states the more formal description given by the British Computer Society's Expert Systems specialist group

> "An 'Expert System' is regarded as the embodiment within a computer of a knowledge-based component from an expert skill in such a form that the machine can offer intelligent advice or take an intelligent decision about a processing function. A desirable additional characteristic, which many would regard as fundamental, is the capability of the system on demand to justify its own line of reasoning in a manner directly intelligible to the enquirer."

Interpreting this quotation in respect to the tasks we are trying to perform, the "expert skill" is the task of translating the user query into the DBMS query language, and the "intelligent advice" is the user query expressed in the DBMS query language. The "additional characteristic" of explanation will also be outlined in this chapter.

3.2 - ENHANCING THE DATA RETRIEVAL PROCESS

The initial effects of using expert systems will be felt in the area of user interaction with the database and in particular with the simplification of query specification for naive database users.

The conventional mode of communicating with a database system can be considered to be that of master to slave. The database management system is merely a dumb slave retrieving data objects as explicitly specified by the user. The database management system has no power for inference nor the ability to take the initiative in any operation[1]

In previous chapters we have identified that the user must possess two types of specific system knowledge. The first is knowledge of the syntax of the query language, in order that a query may be specified in an acceptable form. The second is an understanding of the structure of the database, so as the user is able to map his query, which is in the terms of the real world, into the structure of the database model. Such requirements are condemned by Larson and Wallick [Larson & Wallick 1984] who write

> "Users should not be required to know or remember the contents of the database, the structure of the database, or the formal syntax of a query language".

---

[1]Note that we are discussing just the user interface and are not concerned with the complexities of controlling multi-user access or mapping internal views to physical schemas

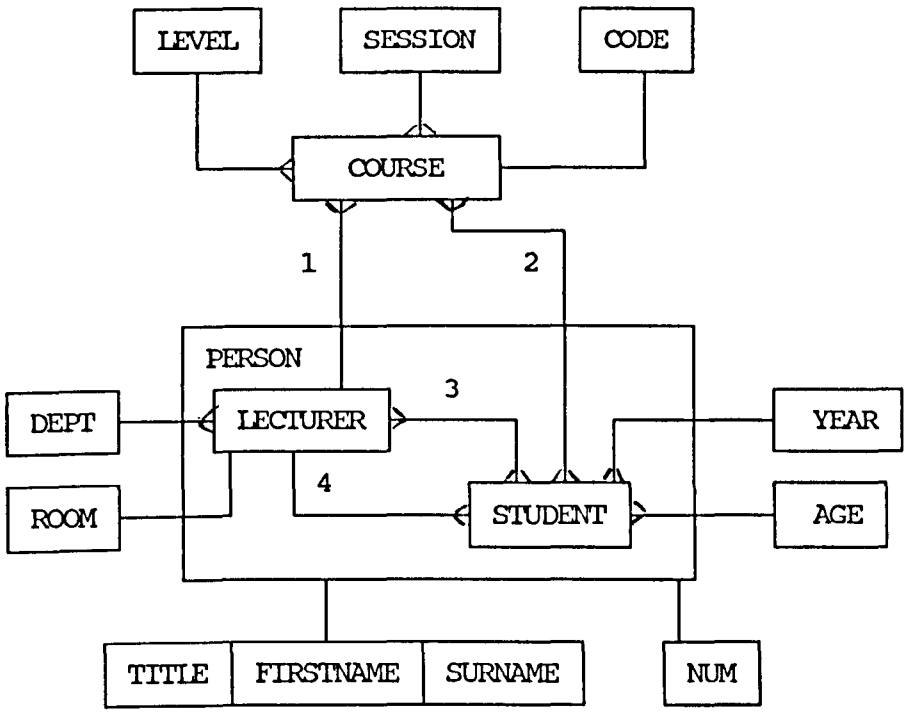
We now consider how both the problem of query mapping and the problem of formal query specification can be eased by using the facilities which expert systems provide.

3.2.1 - Simplifying the mapping process

When a user conceptualises a query he does so in the terms and structures of his own personal view of the 'world'. Therefore, in order to satisfy a user's query these conceptualisations must be translated into the specific structures of the database which represent the modelled situation.

To illustrate this problem let us consider the model shown in figure 3.1. This user view represents an "external schema" as envisaged by an enquirer (The notational semantics of such models is described in chapter 4).

The corresponding relational model for this user view is given in appendix A.



The numbered links have the following meanings

LINK 1 -	LECTURER	LECTURES	COURSE
LINK 2 -	STUDENT	ATTENDING	COURSE
LINK 3 -	LECTURER	TEACHES	STUDENT
LINK 4 -	LECTURER	PERSONAL TUTOR TO	STUDENT

Other non numbered links correspond to "HAS A" links

figure - 3.1

To illustrate the problems faced by inexperienced users consider for the given model the situation where a user wishes to ascertain which LECTURERS are personal tutors to which STUDENTS (This corresponds to link 4 in Figure 4.1). To satisfy this request it is essential for the user to know how the database is structured. If each student has one and only one personal tutor, as in this example, then there will probably be an attribute PERSONAL_TUTOR (PTUTOR in the model in appendix A) of the relation STUDENT. If however it were a many to many relation, then the two relations STUDENT and LECTURER might be

connected via a third relation TUTORS. The impact of the two possible relational representations can be seen quite clearly by looking at the two very different SQL queries which would be needed to implement the user's intentions:

QUERY: fetch the names of students who have Dr Green as their personal tutor

SQL translation if each student has only one personal tutor-

```
SELECT TITLE, FIRSTNAME, SURNAME
FROM STUDENT
WHERE PERSONAL_TUTOR = ( SELECT NUM
                         FROM LECTURER
                         WHERE
                           SURNAME = 'GREEN'
                          AND
                           TITLE = 'DR')
```

SQL translation if each student may have many personal tutors-

```
SELECT TITLE, FIRSTNAME, SURNAME
FROM STUDENT
WHERE NUM IN ( SELECT STUDENT
               FROM TUTORS
               WHERE
                TUTOR = ( SELECT NUM
                          FROM LECTURER
                          WHERE
                           SURNAME = 'GREEN'
                          AND
                           TITLE = 'DR')
```

The latter example also illustrates the difficulties which may confront users as a result of the restricted functionality of current database query languages. In order to conform to the relational representation of the model a user request which is

conditional on a sub-request must be nested with an explicit specification of all of the sub-queries. A language having this restricted functionality can be thought of as a low level query language. As a consequence even an apparently simple query such as

QUERY: get all students who attend course 01CS

must be expressed in a nested form by the user.

```
SELECT *
FROM STUDENT
WHERE NUM IN ( SELECT STUDENT
               FROM ATTEND
               WHERE COURSE IN ( SELECT NUM
                                 FROM COURSE
                                 WHERE CODE = '01CS' )))
```

Low level retrieval places all the responsibility for formulating a query on to the user. To express his query he needs to know how to navigate the rigid syntactic structures and this in turn means he must understand the database structure for the specific model.

As we have seen in chapter two, even the graphical database interface Query by Example requires a certain degree of programming for retrievals involving joins. It achieves this by making the user place variable names in the spaces representing relation attributes.

By using a logic representation we are able to specify user queries in an alternate form. If we reconsider the first example query:

QUERY: fetch the names of students who have Dr Green as their personal tutor

We can express this in a logic form as:

```
query(Title,Surname,Initial) <-
          student(Title,Surname,Initial,*,*,Snum,*) &
          personal_tutee_of(Snum,Lnum) &
          lecturer('DR','GREEN',*,Lnum,*).
```

The predicates student and lecturer represent their respective relational entities, while the predicate personal_tutee_of is the representation of the user perceived connection between the entities. Thus, expressing queries in a logic form allows explicit reference to user concepts without the need to directly navigate the structures of the relational model. The actual specification of such queries is still complex. We therefore see these queries as a "stepping stone" to be used as an internal representation of the user's query, expressed in the concepts of the user's view. The logic query can then be executed against the information stored in the database. By explicitly coding such user perceived connections we are able to reduce the need for the user to understand fully the database structure. For the predicate personal_tutee_of the coding would simply be:

```
personal_tutee_of(Snum,Lnum) <-
          student(*,*,*,*,*,Snum,Lnum).
```

The method of using logic predicates to represent user conceptualisations is similar to using conventional user views, however in section 3.3 we contrast these methods and describe many of the advantages that the representation of such views in a logic based language such as Prolog, has over the expression of views in a relational language such as SQL.

3.2.2 - Reducing formal syntax

The second problem with conventional query language systems is that their syntax is formal, stringently enforced and every term used in a query must be defined explicitly even when its implied context is obvious. The problem of formally specifying a query in the query language can be simplified by using natural language or semi-natural language, in the form of limited vocabulary systems [Good 1984, Kelly 1977]. This relaxation leads to additional functional requirements for the system. These requirements include the handling of ambiguity and provision of an explanations facility to allow users to verify both the interpretation of their initial query and the retrieved results. All of these facilities can be provided by using expert systems.

3.2.3 - Inference to simplify query specification

The requirement for an explicit specification of each term referred to in a query can be overcome if simple inferences are made on the context of words in the query. Consider the following example query

QUERY: In which session is course 01CS taught?

Without further definition we can reasonably assume that '01CS' is some form of identification code for the relation COURSE. We can help verify this inference by automatically checking the internal schema where we obtain the information that the key field of COURSE is a fixed four character string and this therefore gives further

credibility to the inference. The inference made in the previous example is fairly straightforward as only one relational entity is referred to.

The process of inferring attribute and relation names is made more complex when more than one relation is involved. This problem can be seen in the following example query:

QUERY: Get the code for all courses attended by Mr Blue

It is more difficult to infer the relational attributes which are associated with "MR BLUE". However, by using the external schema knowledge that COURSES are ATTENDED_BY STUDENTS, we are able to infer that "MR BLUE" may be an identification of the entity STUDENT. By consulting the internal schema we see that it does not match the key attribute NUM which is defined as an integer. However, it does match the composite object name (See section 4.2.3 for a description of composite objects).

All of the inferences that are made should be verified by the user so that at any stage of the inference process the user can redirect or discard any incorrect inferences. To enable the user to perform such actions we need to provide mixed initiative dialogue.

3.2.4 - Mixed initiative dialogue

Mixed initiative systems are vitally important in systems where inferences are made. The ability for the user to regain the initiative and redirect the system's actions can save the time which will be wasted if the system pursues an incorrect inference.

In order to explain the functions of intelligent interface systems it is sometimes easier to relate their actions to a scenario where two people who are not totally conversant in each others language try to communicate e.g. an English tourist and a foreign official. The need for mixed initiative systems can be seen if we consider such a scenario

The tourist may ask the question

Where is the bank nearest to the bus station ?

The helpful foreign official attempts to answer this question based on his own limited knowledge of the English language. The official from his partial knowledge recognises the "WHERE IS THE" and "NEAREST" so understands the query is one requiring directions. He also identifies the object "BUS STATION". From these "known fragments" he infers that the tourist wants directions to the bus station qualified by "NEAREST" and so replies

ahh! you want the central bus station - you turn left at....

and proceeds to give directions to the Bus station. In response to this the tourist naturally interrupts and thereby retakes the initiative and attempts to correct the wrong assumption the official has made. The tourist interrupts and redefines his query

No, I wish to go to the bank

It is therefore important to provide facilities to allow the user to interrupt and regain the initiative so as to redirect any incorrect inferences and stop any needless execution.

Such a scenario also illustrates the problem that can be encountered by dedicated retrieval systems, such as QPROC and NEL, which ignore terms and phrases that they do not recognise and proceed to interpret the fragments that they do understand.

3.2.5 - Providing explanations

Another important facility that needs to be provided is that of giving explanations. This allows the user to ascertain what inferences have been made and what task the expert system is currently trying to solve. Explanation can be provided in two distinct areas. The first is explanation to verify that the system has correctly interpreted the user's query. The second is justification that the results which have been retrieved are correct. Allowing the enquirer to request justification of the results he receives helps to ensure that what the enquirer asked and

what the system thought the enquirer asked were the same. Therefore justification can improve the integrity of the results.

3.2.6 - Handling ambiguity

In order to simplify the user interaction, we are attempting to reduce the formal specification of a query. However, this increases the possibility that the user may submit an ambiguous query. Using a back-tracking facility gives us an ideal way of handling such queries. When faced with an ambiguous choice the system can choose the path it considers most likely and derive a parse for the query. If the user does not agree with the parsing of a query he can say so. The system will then back-track, revise any inference it has previously made and endeavour to construct an alternative parse which again can be offered to the user for approval. The scope for ambiguity in the English language makes this ability to respond to ambiguous query specifications particularly important when attempting to understand the users intended meaning. The following example query illustrates the problem. In the absence of any mathematical notion of precedence among the logical connectives, there is more than one possible parse of the query

QUERY: Get the names of students who attend courses which are attended by the student named ADAMS or students who are older than 30

Possible query parses -

Parse 1

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,*,Snum,*) &
            attends(Snum,Course) &
            attends(Snum1,Course) &
            student(*,Surname1,*,*,Age1,Snum1,*) &
             (  Surname1 = 'ADAMS' ;
                gt(Age1,30)    ).
```

Parse 2

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,Age,Snum,*) &
            (  attends(Snum,Course) &
               attends(Snum1,Course) &
               student(*,'ADAMS',*,*,*,Snum1,*) ) ;
            gt(Age,30).
```

The user's intended AND OR ordering may not be the "correct" one according to the operator priority rules. However, an intelligent language should be able to deal with these ambiguities and by the use of backtracking combined with user interaction the system should eventually be able to arrive at the user's intended query.

Using predicate logic helps us to distinguish possible parses as we can specify the queries involving "or" as several separate queries (See Appendix B for the respecification of the two previous parses). By specifying several queries we change the "or" from being explicit in the predicate to implicit in the execution of the predicates.

It is important that the user is made aware that more than one possible interpretation of his query exists. The enquirer should be required to confirm that the interpretation offered by the system is

the interpretation he intended. This action helps prevent an erroneous parse from proceeding by default. Most of the dedicated systems reviewed in section 2.5 were unable to cope with queries that had more than one acceptable parse. These systems simply found an acceptable parse and assumed it to be the only one.

3.3 - USING LOGIC TO REPRESENT RELATIONAL DATABASE VIEWS

In the previous section we have used logic to represent user perceived concepts. This has been compared with the activity of creating and using conventional relational user views. In this section we describe several of the advantages to be obtained from specifying user views using logic rather than a standard relational language.

The expression of a view in logic still makes it a view as defined by Codd [1982]

> "A view is a virtual relation defined by an expression or a sequence of commands"

Therefore our logic representations are a form of user view. Using logic programming or more specifically Prolog in combination with relational databases is not new, such work has been described by Gray [Gray 1984] and Zaniolo [Zaniolo 1986]. Zaniolo states that

> "Prolog constitutes an attractive domain-orientated query language for relational databases"

he proceeds to explain that

> "The power of Prolog as a logic language surpasses that of relational calculus, since it is relationally complete [Codd 1972]".

Using a logic language allows us to handle procedural objects rather than the more usual static data object. We can also simplify the definition of user views and allow the definition of these views in a procedural or recursive form.

3.3.1 - Simple view definition

For all the following examples we will use the following relational model

EMPLOYEE

NAME	DEPT_NAME	MANAGER_NAME	YEARS_SERVICE

DEPARTMENT

NAME	FLOOR

figure - 3.2

We can easily define a user view for the above relations which represents the concept that an "employee" works_on a "floor". This view is specified as a virtual relation which has two attributes, the employee's name and the floor which is the location of his department. In SQL this would be:

```
CREATE VIEW EMP_FLOOR_NO
    AS SELECT EMPLOYEE.NAME, DEPARTMENT.FLOOR
       FROM EMPLOYEE, DEPARTMENT
       WHERE
           EMPLOYEE.DEPT_NAME = DEPARTMENT.NAME
```

By considering a logic predicate as a relation we can define the same view succinctly in logic as:

```
emp_floor_no(Name,Floor) <- department(Dept,Floor) &
                            employee(Name,Dept,*,*).
```

It is easy to see the simplification achieved by specifying the query in logic rather than as a conventional user view. It is also far easier to directly execute the logic representation. This simplification of execution is due to the fact that the logically expressed view is of a type which permits "equal opportunity".

3.3.2 - Equal opportunity interaction

Equal opportunity interaction is a feature which allows similar problems to be phrased in many different ways. To illustrate this consider the simple mathematical problem

$$2 + 3 = X$$

If we fully understand this problem then a solver should have no difficulty in resolving the similar problem

$$2 + X = 5$$

This facility is easily represented in Prolog as:

```
add(2,3,X).
```

and

```
add(2,X,5).
```

Although this facility of allowing arguments to be either input or output is not unique to Prolog, as it is available in QBE, it is acknowledged that "Prolog predicates form an obvious basis for the construction of equal opportunity systems." [Runicman & Thimbleby 1986],

In database terms a user would regard the following two queries as similar problems

```
JOHN SMITH works on floor X
    X       works on floor 1
```

Therefore if the system can also regard these as similar it will simplify the task of translation.

For systems to be considered intelligent they must be able to show some understanding of a problem no matter in what order it is phrased i.e. sometimes our queries may be verifications.

The primary characteristic of equal opportunity is that objects can either be input or output i.e. there is no difference between the known input objects which are the retrieval constraints, and the data objects to be retrieved. Conventional relational languages do

not support such non object distinction. They conform to a pattern of

Retrieve List of objects

Subject to conditions

If we compare the SQL view defined in 3.3.1 with its logic representation we see the advantage of using equal opportunity specification in resolving queries

QUERY Fetch names of employees working on the first floor

DATABASE CALL

```
SELECT NAME
FROM EMP_FLOOR_NO
WHERE
    FLOOR = '1'
```

LOGIC CALL

```
Floor = '1' &
emp_floor_no(Name,Floor).
```

QUERY fetch the floor number which John Smith works on

DATABASE CALL

```
SELECT FLOOR
FROM EMP_FLOOR_NO
WHERE
    NAME = 'JOHN SMITH'
```

LOGIC CALL

```
Name = 'JOHN SMITH'
emp_floor_no(Name,Floor).
```

To a user the evaluation of the user concepts working_on and works_on are the same. This similarity is preserved in the logic

representation as the same predicate is used. Whereas when we use the relational specification of the view we have to re-specify the query to make the FLOOR attribute the unknown or retrieval object, and the NAME attribute the input or constraint object.

3.3.3 - Procedural view definition

Using our logic representation we can expand the idea of views to include procedural information. Consider the calculation of annual holiday entitlement for each employee. Suppose that in our example company this is calculated as a basic twenty days plus an additional one day for each year of service up to a total maximum holiday entitlement of 30 days. This can easily be represented in our logic form as

```
holiday_ent(Name,Days) <- employee(Name,*,Years_ser,*) &
                          calculate_holiday(Years_ser,Days)

calculate_holiday(Years_ser,30) <- ge(Years_ser,10) & / .
calculate_holiday(Years_ser,Days) <- Days := 20 + Years_ser.
```

Notice the use of the Prolog cut "/". This is important to control the search strategy. It should be realised that procedural view definition is not unique to a logic representation as PRTV [Todd 1976] provides facilities for such procedural views.

3.3.4 - Recursive view definition

In addition to the fairly simple procedural expression, as shown above, we can also use recursive definitions. To illustrate this consider the situation where we want to find out all the managers for all employees.

```
manager(Emp,Man) <- employee(Name,*,*,Man).
manager(Emp,Man2) <- employee(Name,*,*,Man1) &
                     manager(Man1,Man2).
```

We can easily see that there are many advantages to be gained from using logic, and in particular Prolog, to express queries or user views.

3.4 - enhanced user facilities

Using expert systems not only simplifies the task of extracting data but also gives us the opportunity to provide new facilities to improve the overall process of query satisfaction.

3.4.1 - Meta-level querying

One such facility is the ability to query the system's understanding of the model. When reviewing the system in section 2.5 we noted that very few of these systems offered any opportunity for the enquirer to query the domain.

Meta querying focuses on the actual attributes or on objects in the domain. For example

QUERY: What is an identification number

where the system response may be

An identification number is the key attribute of student and lecturer

The attribute is defined as an integer in the range 100000 to 999999

Alternatively, it may be a meta query relating to an explicit connection between entities, such as:

QUERY: What is the connection between students and lecturers

where the system response may be

There are two explicit connections

Students are taught by lecturers

Students are personal tutees of lecturers

As we are already storing such information in our logic view specifications it is fairly straightforward to provide interactive querying of the system knowledge, and we consider the practical implementation of such a querying mode in section 7.1.

3.4.2 - Understanding a user request

An important area of user interaction is that of trying to understand what the actual task is that the user wishes to perform.Helpful database system interfaces are designed to allow free expression by a user of his query. But even when a query is logically expressed and logically answered the result may not be the one desired. This high-lights the need for a means of answering the query which incorporates a meta-interpretation or task identification of the actual user goal. To illustrate the problem consider the following scenario.

FACT

A train leaves for London at 08:20

QUESTION

Does the train to London leave at 08:10 or 08:30

LOGICAL ANSWER

No

USEFUL ANSWER

No, it leaves at 08:20

This may not only apply to the negation or null answer. If the response is affirmative then we may still need clarification.

FACT

A train leaves for London at 08:20

QUESTION

Does the train to London leave at 08:10 or 08:20

LOGICAL ANSWER

Yes

USEFUL ANSWER

Yes, it leaves at 08:20

The terse logical answer is not very helpful for solving the implicit user query of - "What time does the train to London leave in the time vicinity of 08:20". It is only when the persistent enquirer explicitly follows up his enquiry with additional questions that he is finally able to satisfy his actual but implicit goal.

Useful answers can be categorised in two ways. The first scenario is where they correct a user misconception about the model i.e. the

user has a belief that a train to London leaves at either 08:10 or 08:30. The second scenario is where the useful answer is a classification of the ambiguous answers to the users implied goal.

Detection and recognition of implied user goals is of major importance in the accessibility of querying systems. For such systems to be widely used they need more facilities than just accepting a query specification in a free format. The acceptance and use of such supposedly 'helpful' systems will be hampered if the enquirer still has to state the entire goal explicitly even when it is obvious.

These dialogues also show how abrupt "yes" "no" answers when given by the system may be of limited value, even when they are logically correct.

3.4.3 - Handling negation and null values

Using a logic representation we are able to handle null values directly [Wilkins 1986]. If our null value is an "unknown" null, as opposed to a "not applicable" null value [Gray 1981], then we can either infer a value from surrounding data or we can use the expert system idea, taken from frame-based systems, of an unknown variable being able to inherit values from a pre-specified super class. In effect inheritance provides dynamic default values which are calculated on the known facts and so reduce the uncertainty caused by unknown facts.

The known data set can be expanded by the use of directly specified negation, without the need to rely on negation as failure.

e.g.

Consider a situation for the departmental store model

SALES_STAFF

NAME	AGE	SALARY	DEPT	ADDRESS

DEPARTMENT

TITLE	LOCATION	MNGR.NAME

figure - 3.3

where we know that "J. SMITH", one of the sales staff, does not work in the stationary department but we do not know in which department he does work. This information could not be represented in the simple Departmental Store database as it requires the explicit storage of negation. Explicitly storing both positive and negative data would lead to integrity and redundancy problems. The Departmental Store database would therefore not be able to give the name "J. SMITH" in response to the query

QUERY: What are the names of all employees not working in the stationary department

Representing this problem in logical form we can show both how a conventional database deals with negation and how Expert Database Systems should be able to deal with explicit negation.

Conventional Database

```
works_in(Dept,Name) :- sales_staff(Name,Age,Sal,,Dept,Add) &
                       not Dept = 'UNKNOWN' .

not_works_in(Dept,Per) :- works_in(Dept1,Per) &
                          department(Dept,*,*) &
                          not Dept = Dept1.
```

i.e. a person does not work in a known department if a department exists which is different from the one in which the person is known to work

The only way to achieve negation with this method is by using the closed world assumption and allowing negation by failure. This method of negation can lead to problems in incomplete databases which have unknown or null values.

Using Expert Database Systems allows the specification of explicit instances of negation which help to overcome the previous problems. This can be achieved by imposing semantic checks to ensure integrity. The Expert Database Systems uses the above rules plus the additional rules

```
not_works_in('STATIONARY','J. SMITH').
contradiction('works in') :- works_in(Dept,Person) &
                             not_works_in(Dept,Person).
```

This method allows Expert Database Systems to store negation explicitly in incomplete database systems, and hence give more complete answers than those of conventional database systems

3.5 - SUMMARY

In this chapter we have considered the use of logic and more specifically the use of Prolog to represent the modelled domain. We have shown how such a representation can help alleviate the current problems of query specification and database navigation faced by naive users. We have also shown how the specification of user views in logic is not only more expressive than using a relational language as it allows recursion but is also far simpler to use when attempting to translate and resolve user queries. Thus a logic representation is an ideal internal form for representing not only the domain but also the coding of user queries specified in the concepts of that domain. Finally we described the new aspects of meta querying which a semantic representation of the domain provides, and the way in which Prolog's horn clause logic allows us to represent negation explicitly and thus distinguish it from nullity.

We will now proceed to consider the requirements for implementing such a combination of systems so as to demonstrate the potential benefit that this combination offers. One of the fundamental requirements is that we fully understand the user view and are able to represent the concepts specified by such user views. In the next chapter we analyse a method for capturing and explicitly representing many of the semantic concepts which are implicit in user views.

CHAPTER - 4

REPRESENTING THE SEMANTICS OF DATA

In the previous chapters we stated several problems faced by naive database users and described how the use of an expert system acting as an interface assistant, between the user and the database system, can help alleviate many of these problems. We also considered the need of such a system for a representation of the domain semantics and proposed the use of a logic based representation of the user's view to satisfy this need. In this chapter we consider the task of realising such user views, and describe in detail how the preparatory analysis and remodeling of the data to bring out the user's semantic information can simplify the implementation of these logic specified views.

We begin this chapter by considering how a user perceives the world domain in which he expresses his queries. We next consider the problems of using a conventional entity-relationship design method to produce a relational model to represent these user perceptions. From the resulting relational structure we illustrate the effect that the lack of semantic representation has on the expression of user queries. We then propose a design representation which captures more of the user concepts pertaining to the domain. Using this method we design a second relational model. Finally we compare the translation of user queries for the two relational representations

of the domain, and demonstrate how the translation task of the expert system has been simplified

4.1 - The need for greater semantic content in database design

In the previous chapter we outlined two major problems faced by naive users of database systems. These were formal query specification and query mapping. In order that our interface assistant may help alleviate these problems, it must be able to translate user concepts into the formalised database structures in which they are represented.

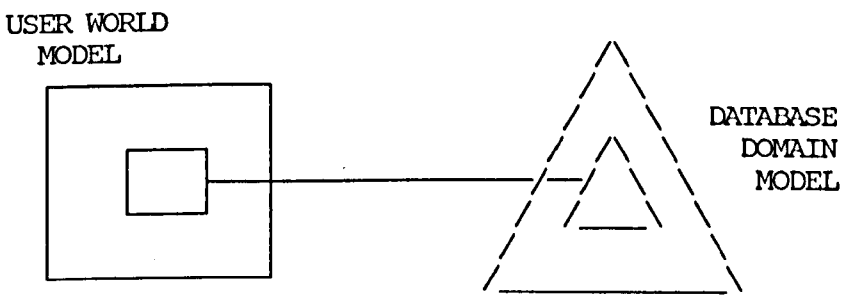


figure 4.1

Figure 4.1 shows the mapping of a query (the inner square) specified by the user in the terms of his own conceptualisation of the real world (the outer square), being mapped onto a specific representation in the database model (the inner triangle).

It is far easier to perform the translation process when the two representations are conceptually similar. By bringing these

representations closer together, that is reducing the conceptual distance between the two representations, we can help simplify the task of translation that the interface assistant has to perform. As Boguraev and Sparck Jones [Boguraev & Sparck Jones 1983] write when describing a translator between the "semantic content of the user's query" and the "administrative structure of the target database"

> "it is necessary to reconcile the user's view of the world with the domain model"

There are two possible ways in which the representations can be brought closer together. The first is to educate the user to understand the database representation, so that his perception of the world includes an understanding of the relational model. The second method is to design and express the constructs of the relational model so that it is closer to the user's own model, thereby making the relational model more representative of the semantic information for the domain.

The first method defeats our objective that the user should not be required to know the relational structure so we will not pursue this. However, it is important to remember that the user's perception of the world includes the querying system itself. Educating a user to understand a database schema for a domain, will probably not alter the user's fundamental perception of the world but it may alter the way in which he expresses his queries when interacting with the database. This is simply because he knows what the database understands. Such a method of education has been

proposed by Roussopoulos and Mark [Roussopoulos & Mark 1986] for use with self describing data models.

Since the first method is inappropriate we will give further consideration to the second alternative, namely to design the database structures so that they more closely model the users own concepts of the domain.

4.2 - The User Model

To achieve a closer representation we must first understand and be able to define the user model. The user model we propose is similar to the conventional external schema, however it is augmented to be more representative of the semantic domain information. The proposed external schema is similar to the End-User level of DIAM II [Senko 1976]. The DIAM II representation of the domain was used to assist the query language FORAL to understand domain contextual queries.

To illustrate the benefits that can be derived from improving the model representation we will consider a familiar Course_Student_Teacher example. We first need to define the general user concepts of the example which we wish to incorporate into the relational representation.

4.2.1 - Primary and subset objects

When defining the user model we need to specify the primary and subset objects. These two classes of objects both correspond to entities in the conventional entity-relationship model. A primary object can be considered a "base" entity which is not a subset of any other entity. A subset object is defined as an entity which is a subset of either a primary object or another subset object which has a primary object as a super-set. A subset object can not be a subset of an object for which it is a super-set.

In the Course_Student_Teacher domain we can identify the following primary and subset objects:

PRIMARY OBJECTS

Person	Course
Department	Faculty

SUBSET OBJECTS

Person ::- (University Employee, Student)

University Employee ::- (Teacher, Secretary)

Teacher ::- (Professor, Senior Lecturer, Lecturer)

Secretary ::- (Personal Secretary, Departmental Secretary)

Entities belonging to either of the two object classes are attributed certain characteristics which explicitly represent the information to be modelled in the domain.

Distinguishing between the two types of objects and representing them explicitly allows us to encapsulate the user notion of inheritance into the model definition. This notion of inheritance refers to the way in which a subset object inherits or is accredited with all of the attributes and relationships associated with the object for which it is a subset. Notice that a subset may be a subset of a subset.

The subsets are not necessarily complete and are not mutually exclusive. Thus we can refer to an instance of a super-set object even when it is not defined as belonging to any of the specified subsets. This allow us to state information relating to a person even when it is not known to which subset, either student or university employee, that he belongs.

The explicit representation of subsets helps the expert system in determining the scope of a user request. e.g. if the user refers to TEACHERS in the context of this example then it is explicitly obvious that the user is referring to

A group of PEOPLE who are UNIVERSITY EMPLOYEES
and are TEACHERS

Thus we are able to associate the sub group TEACHERS with all the attributes and relations associated with its super groups. Hence, we have gained an invaluable insight and understanding of the global scope of a user's reference to an entity.

Languages for schema representation already exist, for example DAPLEX [Shipman 1979] and ADAPLEX [Smith et al 1981]. The schema language of ADAPLEX allows us to define sub sets in the schema as follows:

```
type person is entity
  name      :string[1..40];
  age       :integer;
  sex       :(male,female);
end entity;

subtype university_employee is person entity
  salary    :integer;
  office    :string[1..10];
end entity;
```

Using this specification it is possible to represent inheritance. Such schema information needs to be made available to the interfacing system.

4.2.2 - Non referenced objects

When dealing with entities in the real world certain facets of these entities may be considered characteristic attributes e.g. a person's address. However such characteristics may themselves be entities which can be referred to. In the database world, because it is closed, many of the possible references apply to objects outside the domain of the database. Hence characteristics become simply attributes with no external references. As Addis [Addis 1985] writes

> "The primary entities, although always potentially recognisable as being complex, are considered atomic within the context of a task domain. Change the task and what is considered an entity may also change".

To illustrate this consider the concept that

a course is taught in a session

To represent this concept we could store "session" as an attribute of the entity "course", where the value which can be held by the attribute "session" is either first or second. Using this structure to store session causes all external references to "session", as an entity, to be lost. However, just because these references are lost in the database representation does not stop a user, who specifies his query in the terms of his own concepts about the world, referring to "session" as an entity. For example, a user may pose the query

QUERY: which courses are taught in February ?

Thus session has, in the user model, a characteristic of time although this is not represented in the relational model. The explicit reference to the attribute session as if it were an entity can be compared to a user reference to an actual stored entity e.g. COURSE. If we know the user concept that

a course is attended by students

Then to understand the user query

QUERY: which courses are attended by John Smith ?

we need to realise that the user is making a silent reference to the entity STUDENT i.e. the entity student is not explicitly mentioned but is implied by the phrase "course attended by" and is identified by the string "JOHN SMITH". This illustrates the way in which users

refer to objects, which may be stored as characteristics in the database world, in the same way that they refer to objects which are stored as entities.

To improve our understanding of a user request we must know what the user considers an entity even if such entities are not referred to as entities in the database model domain. Therefore the domain information should include information identifying any user perceived entities which are stored as attributes. In our example domain these could include the following:

NON-REFERENCED OBJECTS

Session	Level
Year of study	Age
Title	Forename
Surname	Street
Town	County
Country	Course Code
Identification Number	

Even characteristic objects such as identification numbers can be considered as entities with a date of issue.

QUERY: which identification numbers were issued in August

The database model can therefore be seen as a filter on the world domain, as obviously we can not store all of the infinite world references to an entity. However, we must realise that users may refer to any of these objects as entities.

4.2.3 - Composite objects

Certain objects in the model are referred to by the user as a single object. In reality these objects may be made up of distinct objects which can themselves be referred to individually. By explicitly incorporating this information into the model we aid the recognition and understanding of the user's request. The following composite objects exist in our Course_Student_Teacher example model.

```
COMPOSITE OBJECTS
  Name ::- (Title, Forename, Surname)
  Address ::- (Street, Town, County, Country)
```

The user may refer to such composite objects as if they were a single object

QUERY 1: What is the age of the person named Mr John Smith?

QUERY 2: What is the age of the person named John Smith?

QUERY 3: What is the age of the person named Mr Smith?

In our model name is not a unique identifier so any of these queries may return many results.

Thus a user may conceptualise the object NAME as a concatenation of several other objects. The user's concept of composite objects must be expressed explicitly so as to define all valid concatenations.

Having a definition of a valid combination gives us a new facility for evaluating the semantic equivalence of composite objects as implied by the user. Considering the above example, a user would

expect any information retrieved for query 1 to be retrieved also by queries 2 and 3. Thus the tuples satisfying the conditional part of query 1 can be viewed as a sub set of both the conditional parts of the other queries.

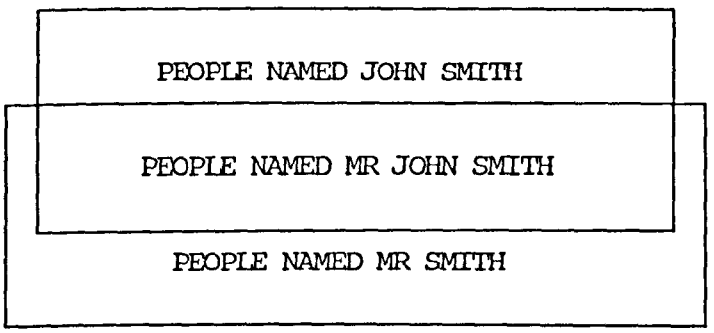


figure 4.2

The explicit definition of implied semantic equivalence is far superior to any form of simple attribute value matching. Improving our understanding and representation of the user's concept of the world helps to simplify the process of translating the user's query into the formalised structures of the database world.

4.2.4 - Relationships between objects

Having described the data objects as perceived by the user, we can now consider the relationships between the objects. When considering the relationship links between objects we are only interested in their existence and type, and not with their philosophical interpretation. For example, it will suffice to know that a STUDENT is TAUGHT BY many TEACHERS. We are not concerned with the underlying meaning of the relationship TAUGHT BY nor any relevance it may have

to the principles of learning. When we refer to the "type" of a relationship we mean the N to M relationship between the objects i.e. one to many, many to one, many to many. Returning to our Course_Student_Teacher example, we can explicitly define the relationships between the objects as follows

RELATIONSHIP CONNECTION TYPES

1. A Person has only one name.

   More than one Person can have the same Name.

2. A Person has only one age.

   More than one Person can have the same Age.

3. A Person has only one address.

   More than one Person can have the same Address.

4. A University Employee has only one salary.

   Many University Employees may be paid the same.

5. A Teacher may be a personal tutor to many students.

   A Student has only one personal tutor.

6. A Teacher may lecture many students.

   A Student may be lectured by many Teachers.

7. A Professor has his own Personal Secretary.

   A Personal Secretary works for only one Professor.

8. A Teacher can have many Departmental Secretaries working for him.

   A Departmental Secretary works for many Teachers.

9. In each year of study there are many students.

   A Student can only be in one year of study.

10. A Student can attend many Courses.

    A Course can be attended by many students.

11. A Student can be in more than one Department.

    A Department can have many students in it.

12. A Department can have many University Employees.

    A University Employee belongs to only one Department.

13. A Person can only belong to one Faculty.

    A Faculty can have many People belonging to it.

14. A Course is taught by one and only one Teacher.

    A Teacher may take many Courses.

15. A Person has a unique Identification Number.

    An Identification Number is had by one and only one Person.

16. A Person has only one sex.

    More than one Person can have the same gender.

17. A Department belongs to only one Faculty.

    A Faculty has many Departments.

18. Many Courses can be taught in each session.

    A Course can be taught in more than one session.

19. A Course can only be taught at one level.

    Many Courses can be taught at each level.

20. A Course has a unique Code.

    A Code is had by one and only one Course.

Having defined all the objects and relationships we can now proceed to design a relational model to represent the Course_Student_Teacher model.

4.3 - Relational Database Design

Conventional database design is heavily dependent on the structures provided by the database system. The table structure of relational databases allows for the logical decomposition of the model into simple entity groups. Each group then has a table to represent it and other tables representing particular relationships between the groups. The process of decomposition [De Bra 1986] although making it easier for the user to visualise the database model, can complicate the query specification task. The task of translation is frustrated due to a loss of the semantic domain information in the explicit relational representation of the domain. This loss of information includes the loss of a representation of the user's macroscopic perspective of the domain. If we are to answer a user's query adequately it is important that we are able to understand fully the scope of the user's query. To achieve this we need an explicit representation of the user's macroscopic perception.

4.3.1 - The entity-relationship model

A conventional methodology used in relational database design is entity-relationship modelling [Chen 1976, Parkin 1982]. When using this technique the problem of the loss of semantic information still persists. This problem is due to the logical sub set decomposition of the domain. This problem can be illustrated by considering the Course_Student_Teacher model. Consider specifically the inter-entity relationships

A PERSONAL SECRETARY	WORKS FOR	A PROFESSOR
DEPARIMENTAL SECRETARIES	WORK FOR	TEACHERS
COURSES	ARE TAUGHT BY	A TEACHER

These relationships were fully defined in section 4.2.4. If we attempt to draw the Entity-Relationship model for these relationships we observe a problem in representing all of the given information

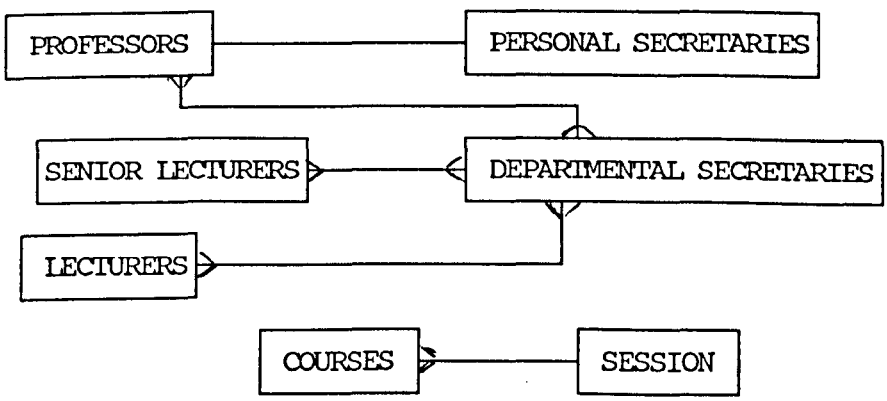


figure 4.3

The model in figure 4.3 allows us to represent explicitly "the works for" relationships. However, we are unable to code the constraint that a course has one and only one teacher. Also, if we design the model with a single super-set entity TEACHER then we lose the information that every Professor has his own personal secretary (figure 4.4).

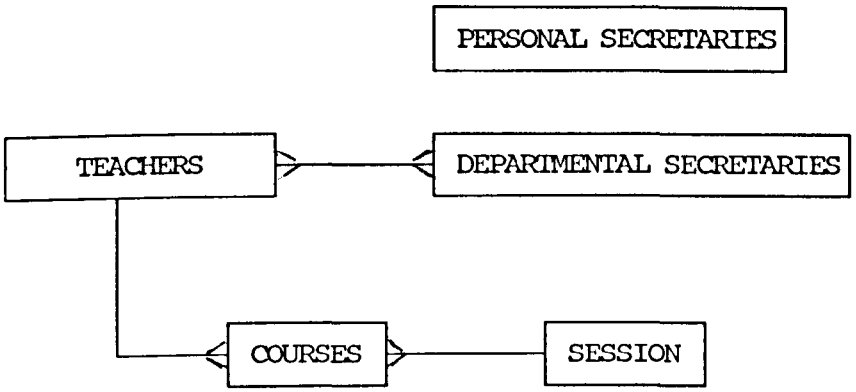


figure 4.4

If we use an "is_a" link we can combine these two models. Figure 4.5 shows these combined models with some additional representation of the macroscopic perspective describing SECRETARIES and UNIVERSITY EMPLOYEES. However, by using such a representation we encounter the problem that we are unable to represent explicitly the information relating to the inheritance of entity subsets.

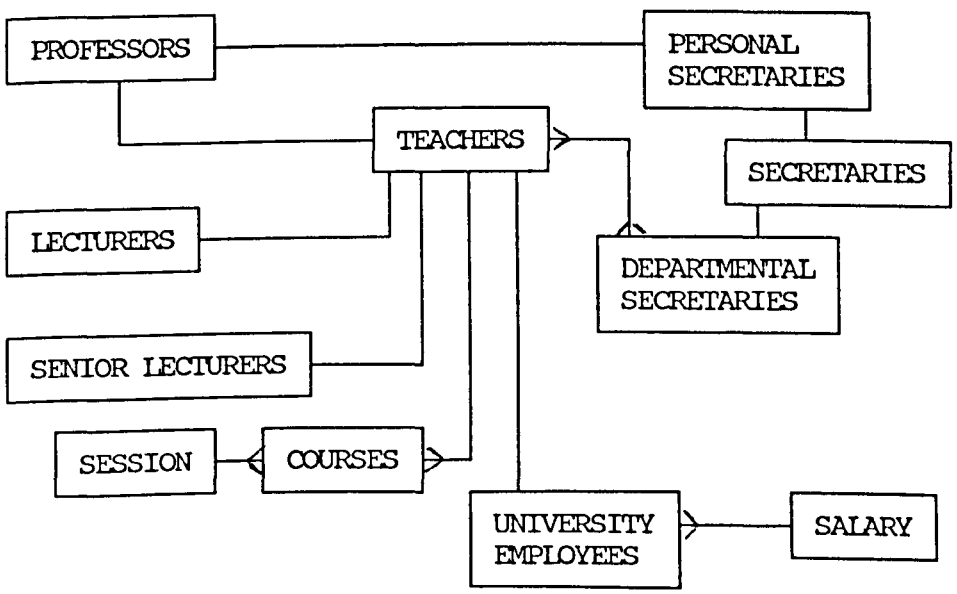


figure 4.5

The following description illustrates this loss of explicit representation of inheritance information.

Knowing that a UNIVERSITY EMPLOYEE is paid a SALARY and a PROFESSOR is_a UNIVERSITY EMPLOYEE we therefore know that a PROFESSOR is paid a SALARY.

However the model in figure 4.5 has no explicit representation of this direct link, that a

A PROFESSOR IS PAID A SALARY

If we compare this with the indirect semantic link that a

PROFESSOR TEACHES A COURSE IN A SESSION

then there is no notational distinction between these two relationships. However, to the user there is a considerable semantic gap between the direct possession of a characteristic and an indirect link with a characteristic via another entity.

To distinguish the relationships we have adopted the notation of nesting subset entities. Thus we can redraw the previous model as figure 4.6. Such sub-entity distinctions correspond to those proposed for the extended relational model RM/T [Codd 1979].

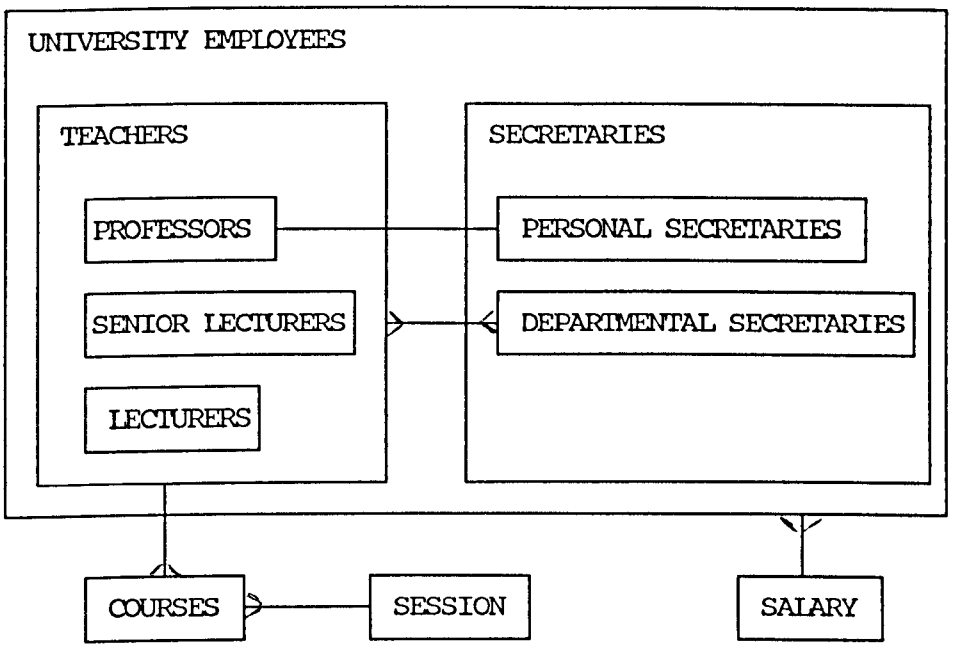


figure 4.6

The semantic distinction between direct and indirect links is vitally important when translating a user's query, as it gives us the ability to distinguish inter-entity and intra-entity relationships, even when both are inter-relational links.

By simply applying the conventional logical decomposition of the entity relationship design method to the Course_Student_Teacher model we derive the relational structure in figure 4.7.

CONVENTIONAL RELATION DATABASE MODEL

PROFESSORS

NUM	NAME	AGE	ADDRESS	DEPT	SALARY	SEX	SEC

SENIOR LECTURERS

NUM	NAME	AGE	ADDRESS	DEPT	SALARY	SEX

LECTURERS

NUM	NAME	AGE	ADDRESS	DEPT	SALARY	SEX

PERSONAL SECRETARY

NUM	NAME	AGE	ADDRESS	DEPT	SALARY	SEX

DEPARTMENTAL SECRETARY

NUM	NAME	AGE	ADDRESS	DEPT	SALARY	SEX

STUDENT

NUM	NAME	AGE	ADDRESS	YEAR	P_TUTOR	SEX

COURSE

CODE	LEVEL	TEACHER_ID	SESSION

STUDENTS_DEPT

STUDENT	DEPT

ATTENDS

STUDENT	COURSE

DEPT_FACULTY

DEPT	FACULTY

figure 4.7

N.B. - NAME is composed of TITLE, FIRSTNAME and SURNAME. ADDRESS is composed of STREET, TOWN, COUNTY and COUNTRY.

This model, although storing the domain data, has failed to capture much of the explicit macroscopic semantic information. Therefore if we are to allow users to express their queries in the terms of the semantic concepts with which they conceptualise the world, then we must specify the lost semantic information externally i.e. in the system which processes the data rather than the data itself. The information is then available for access during queries. However, the more semantic information we can represent in the model the less work has to be done at the interaction stage.

4.4 - MODELLING USER CONCEPTS IN A RELATIONAL STRUCTURE

Having stated that the design of the relational model can adversely hamper the function of an "expert" system interface, we now proceed to propose a structured design method which helps us to incorporate more of the semantic information into the relational model. As Berman states [Berman 1986]

> "There are considerable advantages in basing a design on a semantic data model ... The model is sufficiently easy and 'natural' for end-users to employ"

Our proposed method can be divided into four distinct stages. The four stages are:

1) Rationalisation

2) Unification

3) Duplication removal

4) Composition

4.4.1 - Diagrammatic user model syntax

Before we describe the design stages in detail, we must first define the syntax for our proposed user model representation.

CLASSES AND THEIR PROPERTIES

A class is identified by a box with a label in the top left hand corner.

An enclosed shaded label implies the class is composed of some predefined concatenation of the classes contained within.

The arrows connecting a class to other classes represent the properties of that class.

A class which is completely enclosed by another class is a subset of the original class.

The properties of a class are also the properties of all of the subsets of that class.

CLASS CONVENTION

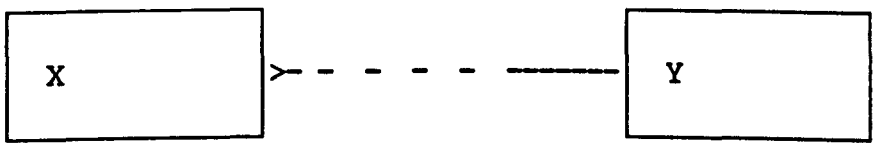
Class connections are bidirectional.

This is read as

for every X there is one and only one Y.

for every Y there are many X.

OPTIONAL CLASS CONNECTIVITY



This is read as

for every Y there exists many X.

for some X there exists one and only one Y.
(This also means that
for some X there are no Y).

Obviously the arrow could be broken over its entire length which would imply - .

for some X there exists one and only one Y

for the remaining X there are no Y

for some Y there are many X, and

for the remaining Y there are no X

4.4.2 - Rationalisation

The first stage is rationalisation. This is where classes with similar characteristics are linked together to form super-set groups, e.g.

> Students have the characteristics name, address and age
> Lecturers also have these characteristics, so we can form a united super-set of 'Person' which has these attributes;

By further rationalisation of the user model and using the user modelling syntax as described in section 4.4.1 a diagrammatic representation of the user model can be drawn (figure 4.8).

USER MODEL

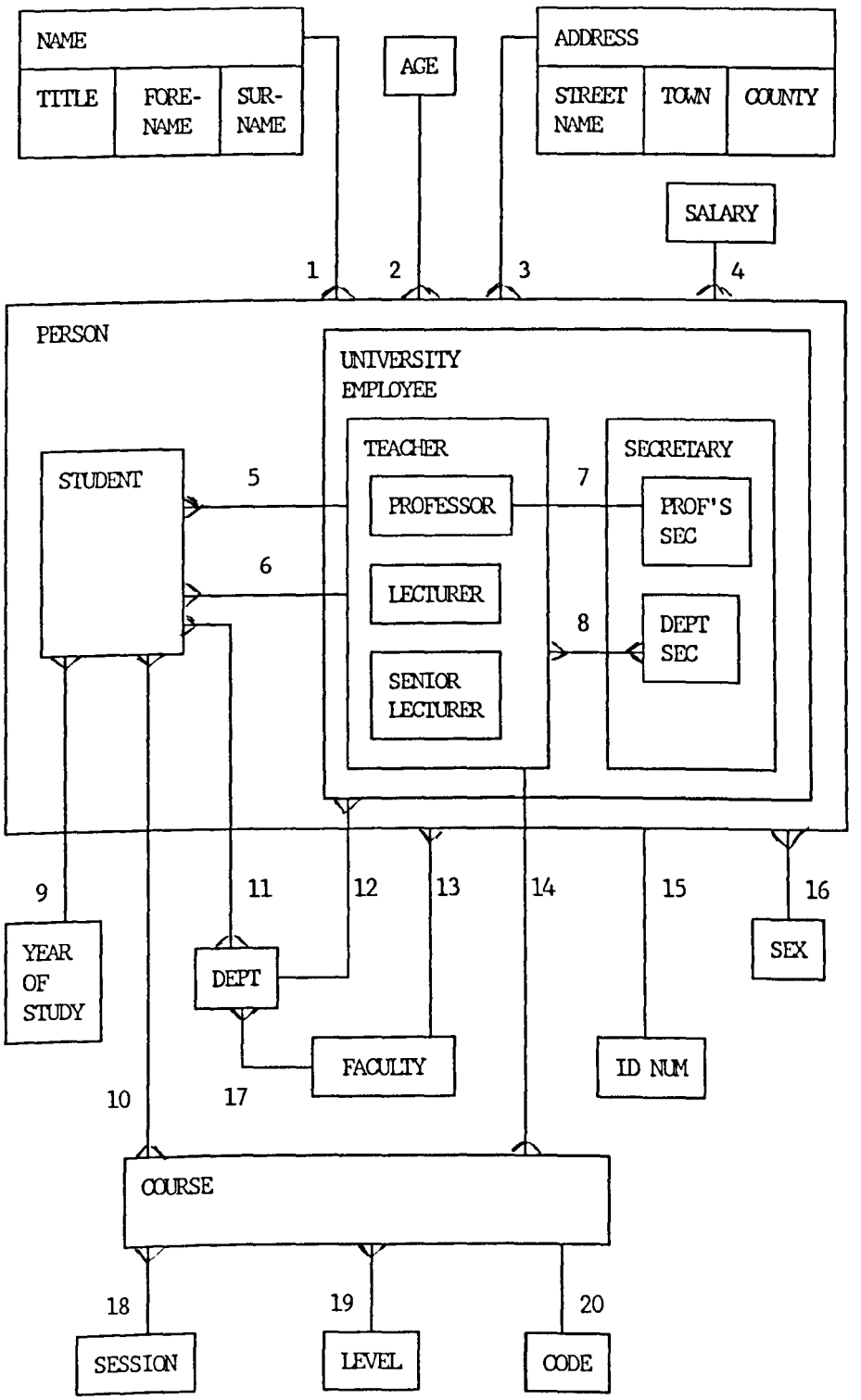


figure 4.8

4.4.3 - Unification

The second stage is to unify the user model. The unified model is very similar to the original User model except that common relationship links have been 'unified' to form a single link. This is achieved by changing a fixed link to be an optional link.

The first link unified was the "works for" link

The Original user model has links where

7. A Professor has his own Personal Secretary.

   Personal Secretary works for only one Professor.

8. A Teacher can have many Departmental

   Secretaries working for him.

   A Departmental Secretary Works for many Teachers.

Combined with the subset information

A Teacher can be a Lecturer, a Senior Lecturer or a Professor.

A Secretary can be a Personal Secretary or a Departmental Secretary.

this allows us to form the unified link

A Secretary may work for (one or) many Teachers. A Teacher may have many Secretaries working for him.

The process of unification is a form of generalisation and so appears to cause a loss in model knowledge. However for user interaction such generalised information is more useful as it shows the connections between groups of entities and not the isolated

sub-entity groups. In fact there is no actual loss of stored data as all of the previously held data is still stored.

The second link unified was the "in department" link The original user model has links where

11. A Student can be in many Departments.

    A Department can have many Students in it.

12. A Department can have many University Employees.

    A University Employee belongs to only one Department.

Combined with the subset information

    A Person can be a Student or a University Employee.

this allows us to form the unified link

    A Person may be in (one or) many Departments. A Department has many People in it.

Again this generalisation gives us the information that members of a Department are people and have all the attributes of the group Person. Applying this process to the user model we derive the unified model shown in figure 4.9.

UNIFIED USER MODEL

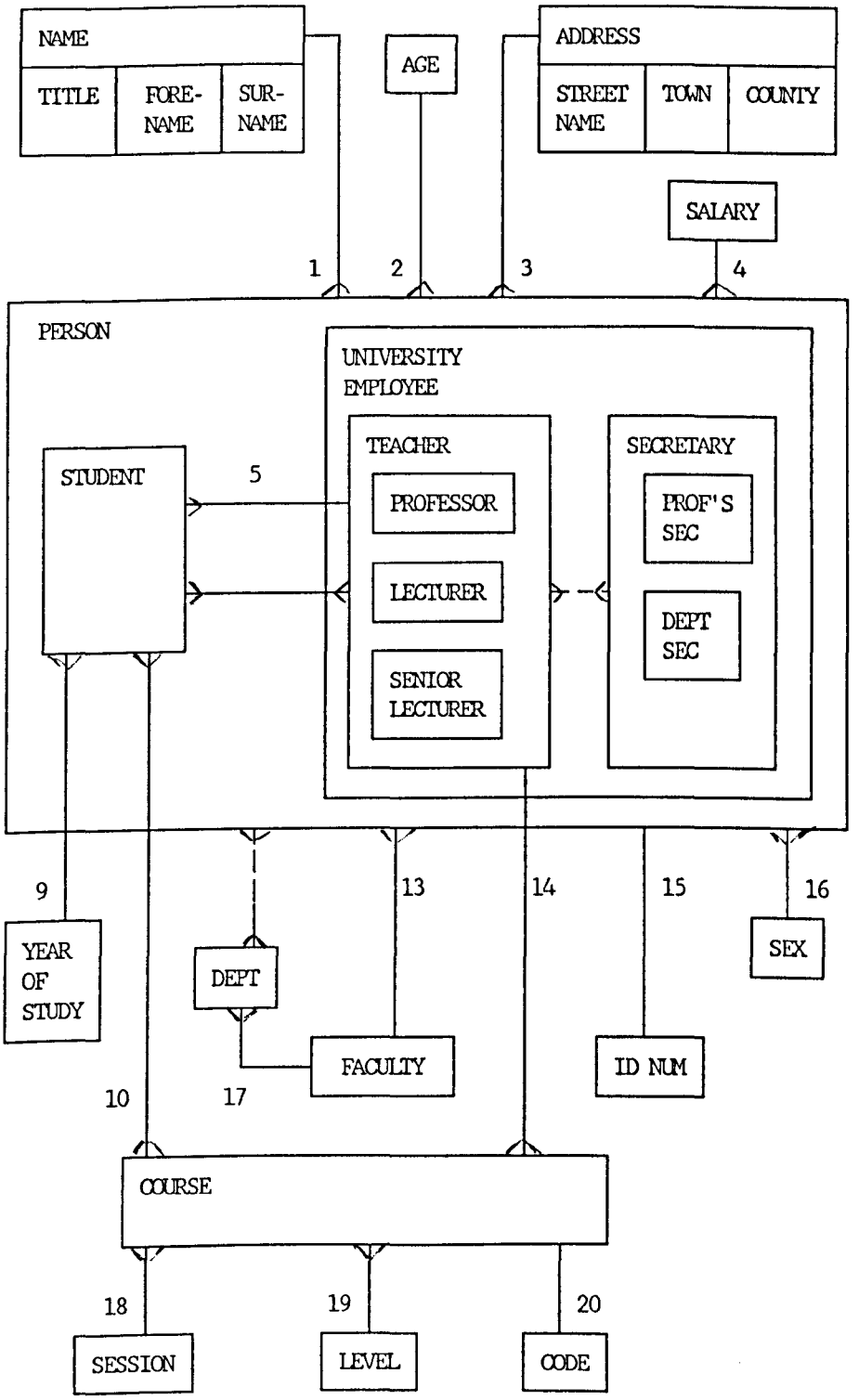


figure 4.9

4.4.4 - Duplication removal

The third stage of the process is to remove any duplicate links before the database structure is formed. There are several relationship links that are displayed in the user model which, although they seem explicit to the user, are in fact implied by other links in the model. Therefore it is important that when the users model is formalised in a database structure these links are not duplicated, as this may lead to problems with consistency constraints.

The first such duplicate link is "Teacher lectures Student" From the original model

10. A Student can attend many courses.

    A Course is attended by many Students.

14. A Teacher takes many Courses.

    A Course is taught by one and only one Teacher.

Therefore if we know which courses a Student attends we can deduce which Teachers lecture the Student :-

Teacher lectures Student = Teacher takes Course &
Course attended by Student

The equals sign can be read as "if" and ampersand as "and"

The inverse of this link "Student lectured by Teacher" must also be satisfied

Student lectured by Teacher = Student attends Course &
Course taught by Teacher

This makes the explicit storage of "lectures" unnecessary.

The second such duplicate link is "Person is member of a Faculty".

From the original model

17. A Department belongs to one and only one Faculty.

    A Faculty has many Departments.

(Unified link 7+8).

    A Person may be in (one or) many Departments.

    A Department has many People.

Coupling the above assumptions with the assumption

13. A Person can only belong to one Faculty.

Thus we can conclude that all of the departments which an individual person is a member of are in the same faculty, and that a person is therefore a member of only the faculty which the department(s) he belongs to are in.

Person member of a Faculty = Person in Department(s) &
Department(s) belong to Faculty

The inverse is also true

Faculty has as a member Person = Faculty has Department(s) &
Department(s) has Person

Applying this process to the unified user model we derive the model shown in figure 4.10.

USER MODEL - DUPLICATE LINKS REMOVED

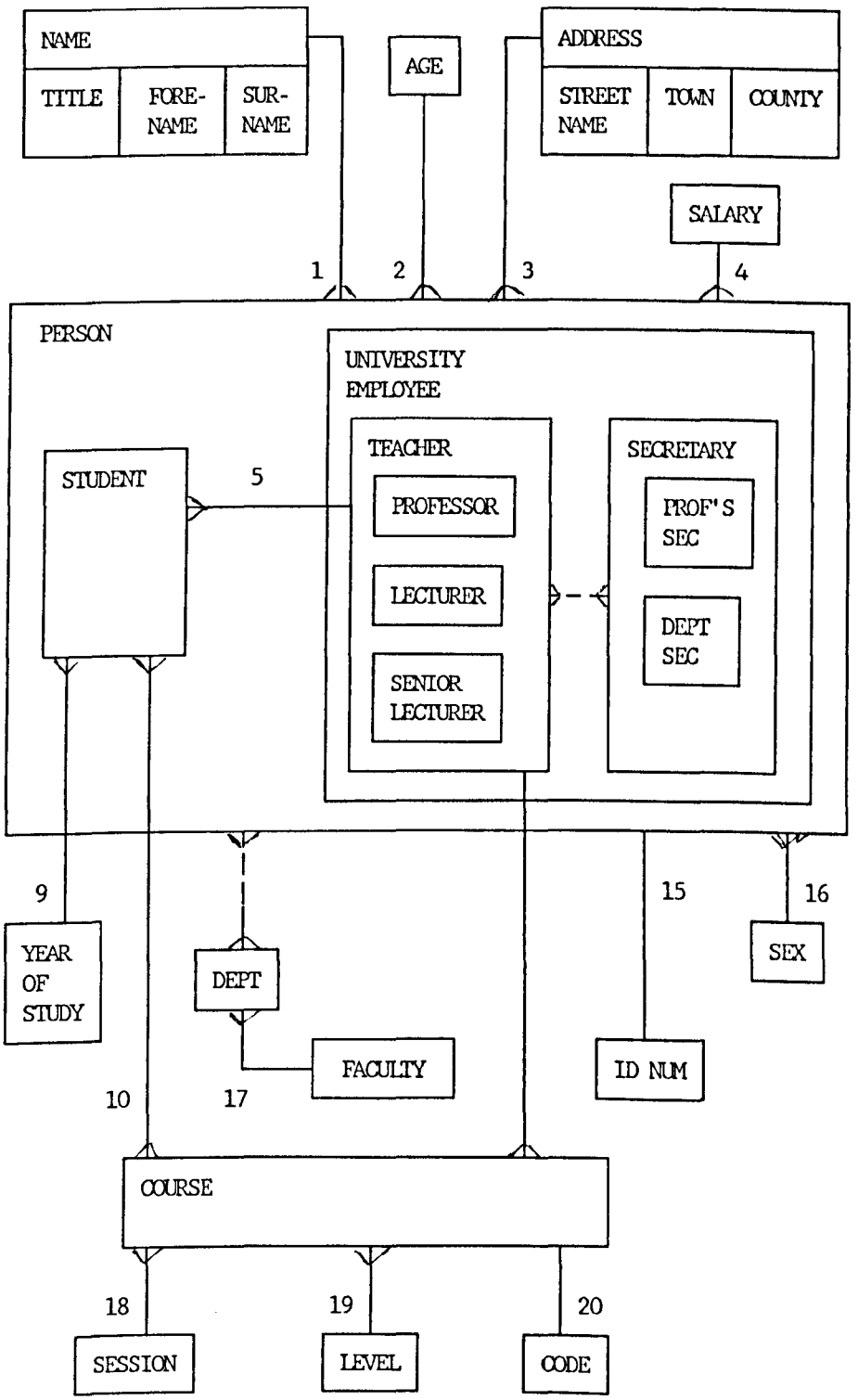


figure 4.10

4.4.5 - Constructing the relational model

The fourth and final stage is to create the actual relational tables. This is a simple process from the final diagram.

1) Every class which is a super-set has a table defining the subsets.
2) Every class which has a single arrow from it has the class it points to as an attribute.
3) An arrow depicting uniqueness is only represented once.
4) A many-to-many arrow is represented by a separate table
5) No class table is represented twice.

Following these simple rules we can create a database with a structure which is more representative of the user model (figure 4.11).

RELATIONAL DATABASE MODEL REPRESENTING USER CONCEPTUALISATION

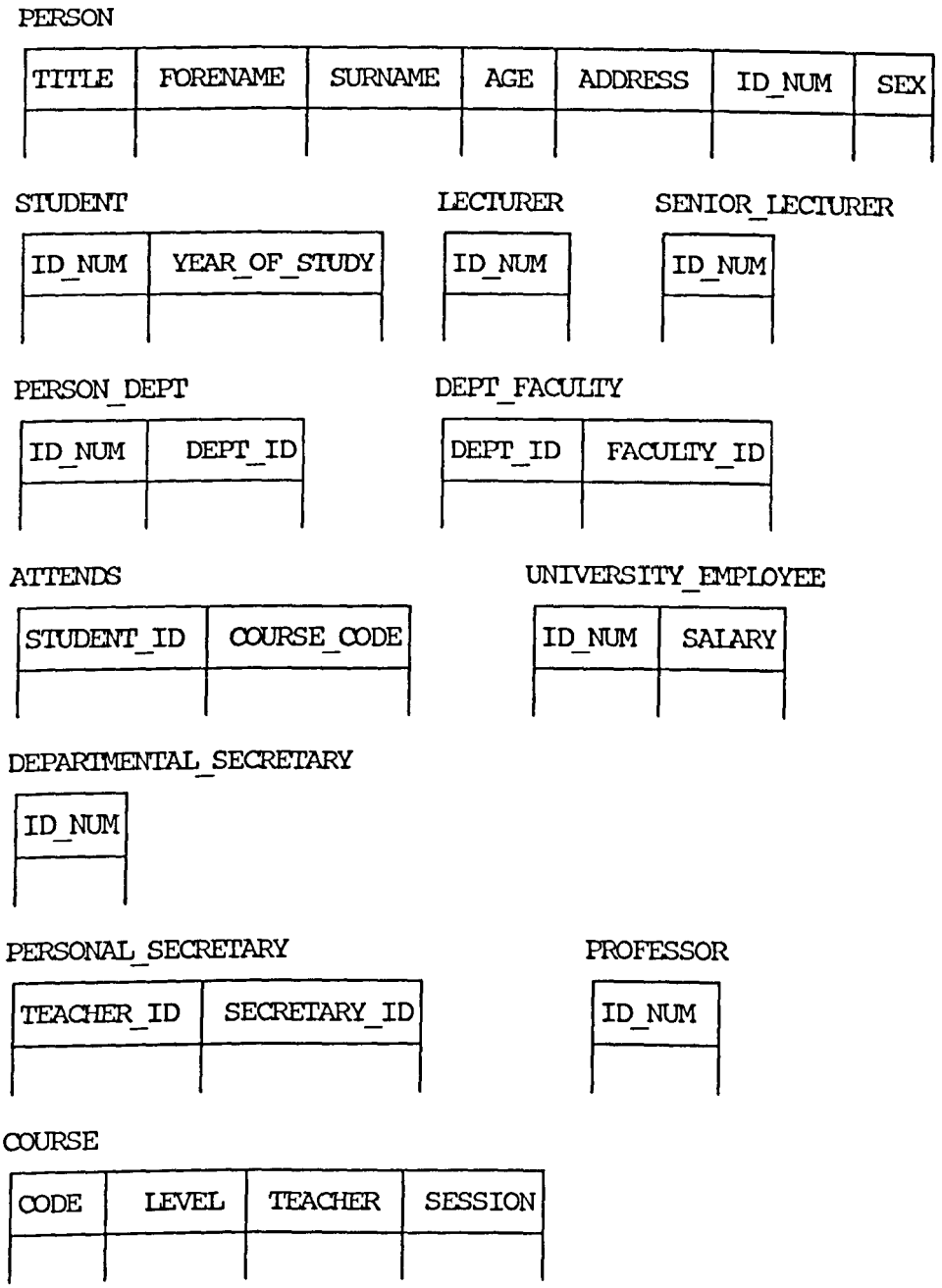


figure 4.11

N.B. ADDRESS is composed of STREET, TOWN, COUNTY and COUNTRY.

4.5 - THE EFFECT OF GREATER SEMANTIC REPRESENTATION ON QUERY SPECIFICATION

Having specified two relational structures (figures 4.7 and 4.11) we are now in a position to illustrate the simplifications that can be made to the specification and hence to the translation of a user's query. In these examples we will use a logic representation of the queries. The corresponding SQL translations of these queries can be found in Appendix C.

QUERY 1: Fetch the names of teachers teaching courses in session 1

DATABASE STRUCTURE 1

```
query(Title,Forename,Surname) <-
       ( professor(Idnum,Title,Forename,Surname,*,*,*,*,*,*) &
         course(*,*,Idnum,1) ) |
       ( lecturer(Idnum,Title,Forename,Surname,*,*,*,*,*) &
         course(*,*,Idnum,1) ) |
       (
          senior_lecturer(Idnum,Title,Forename,
                                        Surname,*,*,*,*,*) &
          course(*,*,Idnum,1) ).
```

DATABASE STRUCTURE 2

```
query(Title,Forename,Surname) <-
          person(Title,Forename,Surname,*,*,Idnum,*) &
          course(*,*,Idnum,1).
```

This simple query of a super group shows the problem of reuniting logically separated groups. When translating the above queries into a relational language the "|" (or) performs a similar operation to that of the UNION command. Modelling the structure with explicit super sets allows us to realise that when an attribute is referred to we immediately know the scope of the reference e.g. when a user

refers to the composite name in our model we know the user is referring to person

QUERY 2: Fetch the names of senior lecturers teaching courses taught at level 2

DATABASE STRUCTURE 1

```
query(Title,Forename,Surname) <-
          senior_lecturer(Idnum,Title,Forename,
                                    Surname,*,*,*,*,*) &
          course(*,2,Idnum,*).
```

DATABASE STRUCTURE 2

```
query(Title,Forename,Surname) <-
          person(Title,Forename,Surname,*,*,IDNUM,*) &
          senior_lecturer(Idnum) &
          course(*,2,Idnum,*).
```

Query 2 shows that even when sub sets are explicitly referenced there is little complication in specifying the required sub group "senior lecturer" of the larger super-set entity "person".

QUERY 3: fetch the names of people who earn over £ 20000

DATABASE STRUCTURE 1

```
query(Title,Forename,Surname) <-
      (professor(*,Title,Forename,Surname,*,*,*,Salary,*,*) &
       gt(Salary,20000) ) |
      (lecturer(*,Title,Forename,Surname,*,*,*,Salary,*) &
       gt(Salary,20000) ) |
      (senior_lecturer(*,Title,Forename,
                                Surname,*,*,*,Salary,*) &
       gt(Salary,20000) ) |
      (personal_secretary(*,Title,Forename,
                                Surname,*,*,*,Salary,*) &
       gt(Salary,20000) ) |
      (dept_secretary(*,Title,Forename,
                                Surname,*,*,*,Salary,*) &
       gt(Salary,20000) ).
```

DATABASE STRUCTURE 2

```
query(Title,Forename,Surname) <-
        person(Title,Forename,Surname,*,*,Idnum,*) &
        university_employee(Idnum,Salary) &
        gt(Salary,20000) ).
```

This query shows that reference to a sub group which itself is a super group is also simplified in our new model (i.e "university employee" is a sub group of "person" and super group for "teachers" and "secretaries").

4.6 - SUMMARY

In this chapter we have considered the specification of many of the concepts constituting the user view, and have outlined a method for diagrammatically representing such concepts. We identified the notion of conceptual distance between representations, and stated that it is easier to perform a translation between two representations when they are "closer" together. We described a method for enhancing the macroscopic perspective of the relational model by improving its representation of the user view, and thereby bringing the two representations closer.

By considering the specification of queries in Prolog, we illustrated the significant improvements to be derived from this enhanced representation. We found that this approach aided both the understanding of the scope of the query and vastly simplified the expression of such logic queries.

CHAPTER - 5

COUPLING EXPERT SYSTEMS AND DATABASE MANAGEMENT SYSTEMS

We have considered the benefits which an expert system approach could provide for database users. In this chapter we describe and investigate possible methods for coupling expert systems and database management systems. Our aim is to specify a system which is seen as a single unified structure, which can be used to implement our proposed data retrieval system. We start by considering the principal strategies for forming a single system. We then proceed to develop a specific architecture. This proposed architecture helps us to identify the need for a communications link between the expert and database systems. Using systems which are currently available, we consider the practical implementation of the communications link. We then describe the problems which frustrate such a link. Finally we propose several methods to alleviate these frustrations.

5.1 - APPROACHES TO CONSTRUCTING A COMBINED SYSTEM

It is important to ascertain initially the functional requirements of a combined system. This will allow us to determine the degree of combination of the two component systems, the expert and the database systems. The degree of combination refers to the dominance each of the underlying systems' characteristics has on the functionality of the final system. Factors such as the predominant

type of search technique required by the final system must be considered. The type of search techniques corresponds to the different search strategies of the two underlying component systems. One must also consider the trade-off between Optimisation and Prototyping and between Compilation and Interpretation.

The degree of combination is dependent on which of the construction strategies is chosen. There are three distinct strategies for combining the systems [Stott Parker 1984].

> The first is to enhance the query language of an existing Database Management System so that it incorporates the facilities which an Expert System possesses

> The second method is to create a means of communication between an existing Expert System and a Database Management System [Vassiliou 1985].

> Finally the third method is to add to an existing Expert System the multiple user secondary storage access and management routines which are necessary for the implementation and control of a large database.

These strategies illustrate how the functionality of the resulting system will be effected by the method of combination chosen. For systems which require highly efficient mass data retrieval then the

first system would be most appropriate. For a system which is strongly dependent on knowledge-directed processing then a system like the third would be best.

We have decided however, to implement the second design. This was because it gave us a more flexible and general system with the greatest opportunity to maximise the advantages to be obtained from both systems. By using systems which were already inexistance it saved us considerable time in development and we were able to build demonstrators relatively quickly to illustrate the functional operations of our proposed system.

5.2 - MAINTENANCE OF COMPONENT INDEPENDENCE

Another important reason why we adopted the second design strategy was that it maintained the independent identities of the two systems. Although we are attempting to construct an apparently single unified system we are keen to maintain the independent physical identities of the two systems. We see several advantages arising from this separation.

Maintaining system independence is beneficial to us as we are designing systems only for specific user groups, such as naive users. By maintaining the separate identities we do not exclude any current application of either system. Therefore we do not affect or hamper access by users who are outside our specified user group. As

shown in figure 5.1, group 2 users are not affected by the introduction of an expert system front end.

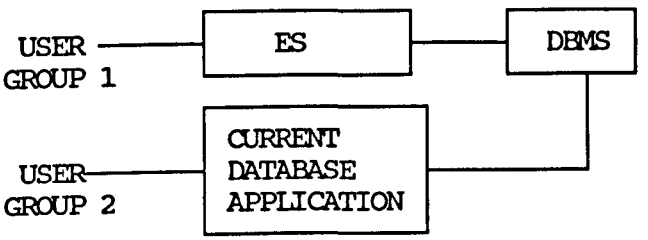


figure 5.1

Maintaining independence from a given database management system allows us to use the interface as a means of providing a standardised database interface.

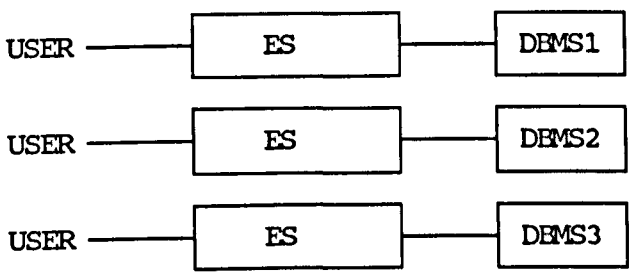


figure 5.2

As shown in figure 5.2 the users interact with the a common front-end and are oblivious as to which database management system they are actually using. This standardised front end shields the users from needing to know any system specific information.

This notion of a single standard front-end and multiple database back-ends may be pursued so as the expert system becomes a manager to a distributed database system or a gate-way advisor to discrete

database systems. Such a gateway system is illustrated in figure 5.3.

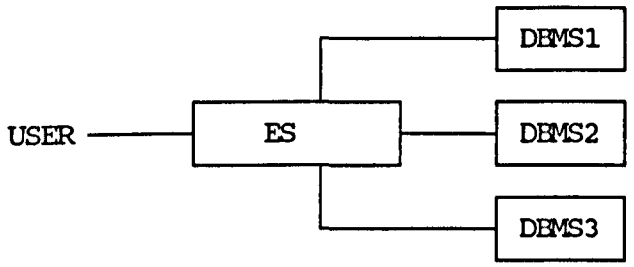


figure 5.3

5.3 - ARCHITECTURE FOR MULTI-USER UNIFIED SYSTEM

When coupling expert systems and database management systems it is important to preserve the functionality of each of the component systems in the final system.

One such function is the multi-user access facility of database management systems. We have identified two possible architectures which preserve this facility. The first is to create the system by providing each user with access to an isolated copy of an expert system, which is able to communicate with a shared Database Management System (DBMS). Figure 5.4 illustrates how using this architecture we can create a virtual multi-user system.

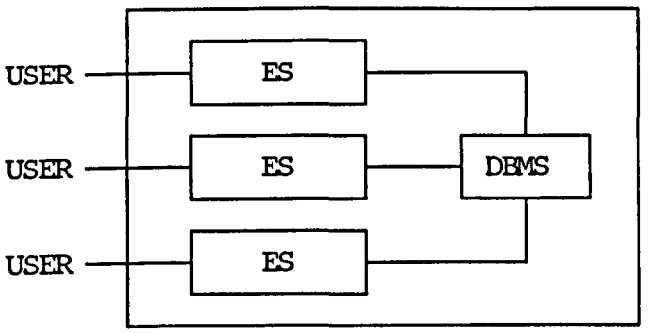


figure 5.4

The outer box depicts the virtual shared system.

This configuration is really only a simulation of a shared system. Although it uses a common database each user's expert system is isolated and hence is unable to share the knowledge represented in the expert systems as a whole.

A fully shared system with a single Multi-User Expert System front end (MUES see figure 5.5) provides a uniform interface to users while interacting with the database.

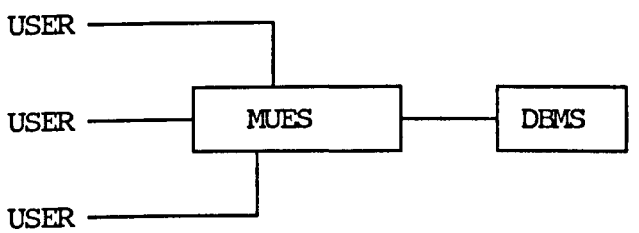


figure 5.5

Thus the MUES manages the multi user function. Although gaining the benefit of a global data management system the fully shared system

(figure 5.5) loses several of the advantages asociated with the virtual shared system of figure 5.4.

The first structure (figure 5.4) gives us greater opportunity for future development of adaptive interfaces for individual users and maximises the available benefits of both of the existing component systems. As the DBMS already has shared access management facilities it is far easier to implement the structure of figure 5.4 than it is to construct that of figure 5.5. We have therefore chosen to implement the structure illustrated in figure 5.4.

5.4 - THE INCORPORATION OF DOMAIN DATA INTO THE ARCHITECTURE

Even though we have decided on the general structure of the system, certain aspects of the combined system architecture still remain undefined. These aspects occur where the component systems' functions overlap.

One such aspect is the storage and manipulation of domain description information, the expert domain data. This data can either be stored in the actual database or represented in the expert system. The locating of the expert domain information is dependent primarily on its volume and to some extent on its nature. When we refer to the nature of the data we are referring to the way in which it is represented, this may be as data objects or production rules etc.

The first method of storing the expert domain information in the expert system leads to a situation where we have an isolated linear combination (figure 5.6). Using the second method we can derive a fully integrated system [Brodie 1984] (figure 5.7). To the enquirer however both systems appear integrated.

LINEAR COMBINATION

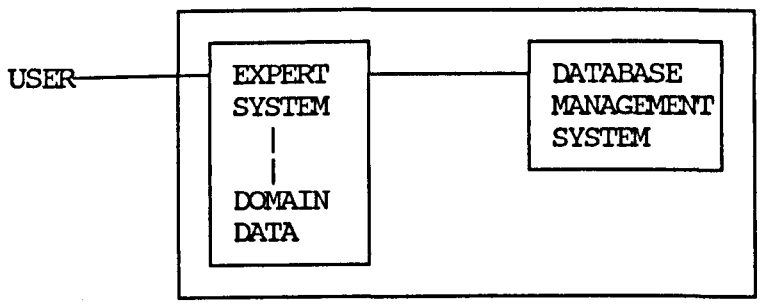


figure 5.6

INTEGRATED COMBINATION

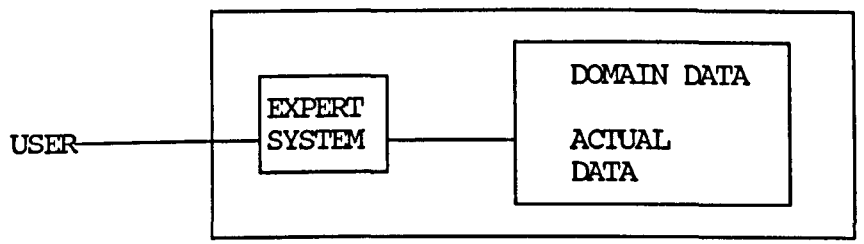


figure 5.7

The integrated architecture requires the expert system to handle both queries and updates to the two types of data. The greatest advantage for such a combined system is in the future development of large Expert Systems which will be able to share not only the data objects but also the domain knowledge. Implementing such a system would require considerable work in knowledge representation. As we

were not explicitly concerned with this we have chosen the structure represented in figure 5.6.

The structure we have chosen (figure 5.6) satisfies our requirement for functional independence of the two systems. This is in contrast to the integrated architecture which violates this requirement as the expert system can not function without the database system, although the database system could function alone.

If we apply the domain data structure of figure 5.6 to our chosen multi-user architecture (figure 5.4) we can derive the complete architecture for our combined system, which is displayed in figure 5.8 .

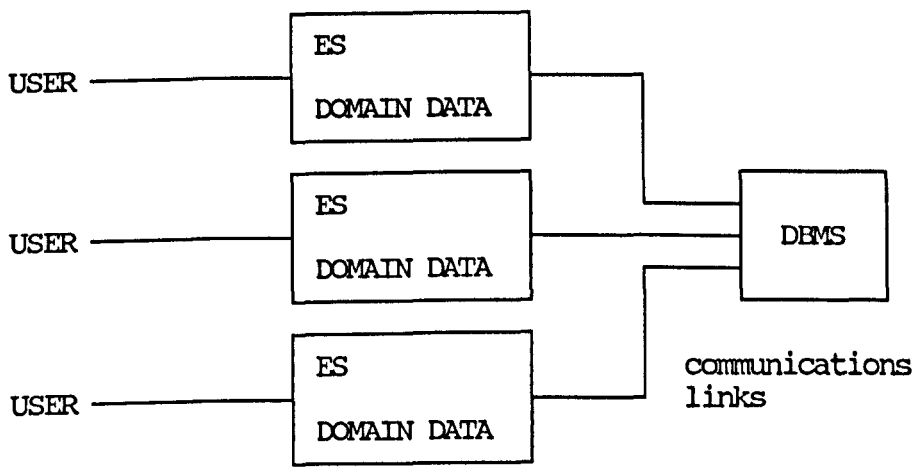


figure 5.8

As we have previously stated we wish to make use of existing systems, therefore in order to implement our chosen architecture we need only create the communication link between the two system components.

Our chosen architecture is similar to the one proposed for the Difead system [Al-Zobaidie 1987]. The Difead system proposes the use of an independent sub-system (ISS) to control the interaction between the expert and database system.

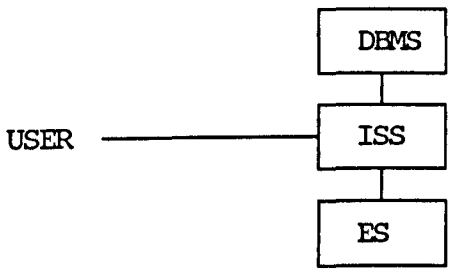


figure 5.9

We regard the control of interaction as an expert system task. We therefore have only one expert system, and represent the Difead application expert system (ES in figure 5.9) as our expert domain data which can be used by our single expert system.

Having formalised the architecture for our proposed system (figure 5.8), we can now proceed to consider the communication link between the two component systems required to implement the combined system.

5.5 - THE COMMUNICATIONS LINK

The communications link for our chosen architecture must satisfy the following two rules.

1. To enable the expert system to extract any item of data that is stored in the database which the interface user is allowed to access.
2. The use of the communication link should not impinge on the operation of either system so as to hamper the usage of either system in isolation or in combination.

The first objective covers two criteria namely that the interface must

1. have the power to extract **all** data items that are available to the user.
2. be allowed to extract **only** the data items that are available to the user.

This second criteria requires the combined system to enforce the user security access checks used by current database management systems. This strengthens our desire for using existing systems, as to satisfy the security objective the expert system has only to make the underlying DBMS aware of the identity of the interface user.

5.6 - APPROACHES TO IMPLEMENTING THE COMMUNICATION LINK

Our initial objective of enabling the ES to extract any required item of accessible data is frustrated by the interactive nature of the combined system. The primary difficulty with an interactive system is that queries are not known in advance. It is therefore necessary for the ES to create the database queries dynamically during the interaction. Thus the communications link or coupling must be able to handle queries which have not been previously specified.

Jarke & Vassiliou [1984] suggest four forms which the coupling between the Expert Systems and Database Management System can take:

Elementary data management within the ES. Data facts are held as part of the main ES program

Generalised data management within the ES. Data facts are held in simple files which are accessed by the ES.

Loose coupling of the ES with an existing DBMS. Loose coupling (Snapshot) is where data extractions occur statically before the actual operation of the ES. The extracted data is copied and stored as a separate expert system database.

Tight coupling of the ES with an existing DBMS. Tight coupling (On-line Interaction) is where the database appears as an

extension of the ES. Data extraction from the database occurs during the operation of the ES.

It is only the last two techniques 3 and 4, which are of interest to us as it is only these which interface to existing DBMS. The third and fourth methods offer two different types of strategy for building dynamic interfaces to existing DBMS. However, it is only interfaces of type four that are truly interactive. We now proceed to describe different methods for implementing the two types of DBMS link, loose coupling and tight coupling.

5.7 - LOOSE COUPLING

5.7.1 - The snapshot

The first technique known as 'taking a snapshot' would require the dumping of either the entire contents or a domain sub-section of the DBMS. Accesses (i.e. queries) would then be made to these files independently of the DBMS, eliminating the need for any further DBMS operation. The data in the dumped files could be updated at set time intervals. These time intervals would be dependent on the volatility of the data. The accessing strategy for the dumped files would be controlled by the interfacing program. The main advantage of this method is its simplicity; the only DBMS command ever specified would be the DUMP command. This increases the degree of independence of the ES from any one particular DBMS. The main disadvantage of

the snapshot method is that it extracts the data only once so that any updates to the actual database can lead to inconsistency problems. Also this method fails to take advantage of the look-up or addressing facilities and aggregation facilities that can be obtained by using the DBMS. The problems of this method are accentuated when we are dealing with large databases. It would be impractical to down-load the entire database therefore only a chosen sub-section can be down-loaded. This may lead to problems with determining which section of the database is required. Finally there is also the obvious waste of duplication. In 5.11 we compare an implementation of a snapshot method with several other methods.

5.8 - TIGHT COUPLING

The following three techniques all interact with a DBMS at run time and use at least some of the facilities the DBMS provides. All of these techniques implement in different ways the concept of tight coupling.

The three techniques are:

predefined all-tuple retrieval
semi-dynamic querying
fully dynamic querying

5.8.1 - Predefined all-tuple retrieval

The simplest technique for interactive retrieval, retrieves each and every tuple of a relation as and when the relation is queried, this is irrespective of whether the tuples satisfy the constraints of the original query. All of the condition testing, attribute selection, aggregate calculation or expression evaluation, has to be carried out in the interface host language. We have investigated the use of such an interfacing strategy for the following two systems

RAPPORT - FORTRAN

The RAPPORT to FORTRAN interface, as it is currently available, requires that all FORTRAN programs containing RAPPORT commands (the CPI or pseudo FORTRAN program) must be preprocessed by the RAPPORT compiler before they are compiled by the FORTRAN compiler. This makes it almost impossible for the interface to handle true dynamically generated queries directly. Therefore a predefined "all-tuple" retrieval system was implemented. This system used predefined FORTRAN subroutines which could retrieve one tuple at a time from any of the relations in the database and then pass this tuple to a higher-level interface which could test if the tuples satisfied the initial retrieval requirements. If they did then the values of the attributes would be processed.

The following figure illustrates the general layered structure of the interfacing system when precompilation of the queries is required. It also shows the need for hand-crafted model-dependent FORTRAN retrieval routines.

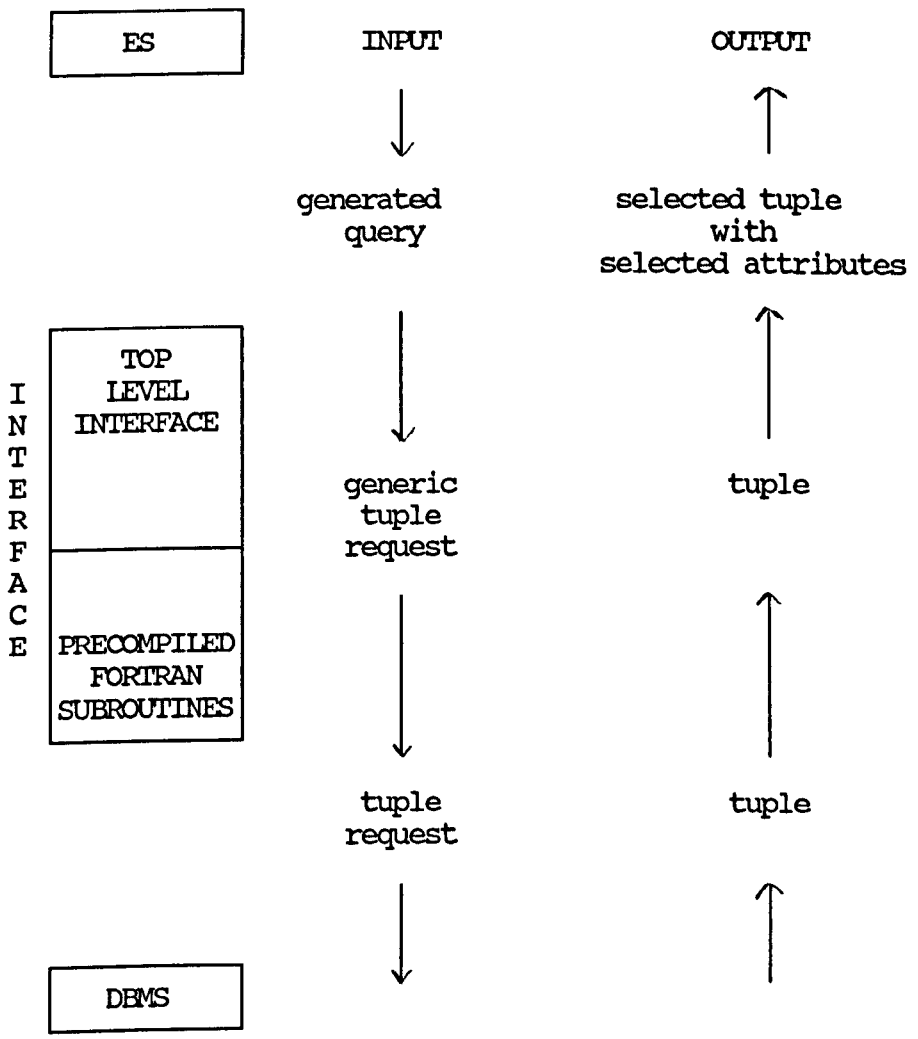


figure 5.10

The ES to DBMS interface is divided into two. The top level is dependent on the underlying DBMS and the second level, containing the hand-crafted subroutines, is dependent on both the underlying DBMS and the actual database model.

SQL/DS - FORTRAN

We have also considered using the SQL/DS - FORTRAN interface but did not proceed to an implementation as the link required would be very similar to the previous system. Like RAPPORT the SQL/DS - FORTRAN interface demands that the FORTRAN program, with embedded fully defined SQL commands, must be preprocessed. This fixes the commands and severely limits the ability to generate queries for a non-expert user.

The use of predefined all-tuple retrieval is only a slight improvement on the snapshot. It avoids consistency problems as the data retrieved is more up to date but is extremely inefficient. Vast amounts of data are retrieved which will never be required and no advantage is taken of any of the retrieval strategies provided by the database.

The restrictive problem of predefinition appears initially to indicate that FORTRAN is a poor host language for database interaction. However FORTRAN does have one advantage as a host language in its ability to handle type constraints. The type constraint is a problem encountered by many other host languages.

A type constraint occurs when retrieved values can be assigned only to variables of the same type. This implies that the type of the attribute to be retrieved must be known in advance so that suitable variables of the same type can be defined.

e.g. For the following query

```
SELECT PARINO, DESCRIPTION, QONHAND
INTO VPARINO VDESCR VQONHAND
FROM INVENTORY
```

the program variables must be defined so that

```
INVENTORY.PARINO       is of the same type as VPARINO
INVENTORY.DESCRIPTION  is of the same type as VDESCR
INVENTORY.QONHAND      is of the same type as VQONHAND
```

In general the set of attributes to be retrieved is unpredictable, but when using FORTRAN this problem of type matching could be overcome by the use of the COMMON block area which has a free format and could be defined to hold any retrieved tuple. Surprisingly neither of the previously described systems allow the interfacing programmer access to this COMMON block area. However it should be noted that the use of COMMON has been the subject of debate in the discussions now taking place on a revised FORTRAN standard.

5.8.2 - Semi-dynamic querying

Semi-dynamic interaction is where predefined template queries can, by the use of in-built parameters, be adapted to form the required query. Two such systems have been studied

SQL/DS - COBOL

This interface allows the generation of semi-dynamic queries by using parameters which can be placed into the initial specification of the query. Assignment can then be made to these parameters in order to modify the predefined template query into the required query.

The following template query is for the relation Student in the Course-Lecturer-Student database.

```
SELECT *
INTO :student_record
FROM STUDENT
WHERE
  STUDENT.NAME    BETWEEN LOW_NAME    AND HIGH_NAME
    AND
  STUDENT.NUMBER  BETWEEN LOW_NUMBER  AND HIGH_NUMBER
    AND
  STUDENT.ADDRESS BETWEEN LOW_ADDRESS AND HIGH_ADDRESS
    AND
  STUDENT.AGE     BETWEEN LOW_AGE     AND HIGH_AGE
```

By assigning values to the parameters HIGH_attribute and LOW_attribute for every attribute in the tuple, we can form the required query.

Although the Semi-dynamic SQL/DS-COBOL interface is far from being the most efficient method of retrieval it is more selective and so is more efficient than retrieving every tuple, especially when some optimising technique is used by the database management system.

A similar method, although not quite as powerful, is used by the RAPPORT to PROLOG interface. This allows specification of equals constraints, the most efficient form of access constraint.

The form of the retrieval communication can be seen in the following example.

QUERY: fetch students aged 20

PROLOG: fetch(student,Name,Number,Address,20).

where student is a relation which has the attributes name, number, address and age.

It remains necessary for constraints other than "equals" to be handled in the interface, i.e. AGE = 20 is easily specied, however had the query been

QUERY: fetch students aged over 20

PROLOG: fetch(student,Name,Number,Address,Age) & Age > 20.

Then the constraint test is performed in the interface itself.

The RAPPORT PROLOG interface is reminiscent of the ALL-TUPLE retrieval method, although it does use parameters to specify equals constraints.

The semi-dynamic method is by no means ideal. It is an improvement on the two earlier methods, in that it is now possible to alter the retrieval conditions, but it is still unable to specify which tuple attributes are to be retrieved, so it still requires the projection of the tuple onto the required result format. Also it fails to take full advantage of DBMS facilities of addressing and aggregation, and like the previous methods it is unable to handle queries involving joins, such as:

Fetch students who are the same age as student Adams.

```
SELECT *
FROM STUDENT
WHERE
       AGE = ( SELECT STUDENT.AGE
               FROM STUDENT
               WHERE
                 STUDENT.NAME ='ADAMS' )
```

If the above example were implemented using a semi-dynamic method, then the age of the student with NAME 'Adams' must be obtained and temporarily stored, before the data for students with the same age can be retrieved. To resolve queries of this type, using any of the previously described interface strategies, would require the temporary storage of the intermediate results. This may lead to serious consistency problems.

5.8.3 - Fully dynamic querying

The ideal coupling would be one where we could specify all or any of the available DBMS operations during the execution of the interface. With fully dynamic querying we are able to do this. None of the queries requires precompilation as this is performed at runtime. Three systems of this type have been investigated.

SQL/DS - PL1

The SQL/DS to PL1 interface provides the facility for queries to be specified in a string which can then be compiled and executed. This means that queries can be generated in a free format in the most efficient form for the DBMS. The operation is divided into five stages

PREPARE
DESCRIBE
OPEN
FETCH
CLOSE

To overcome the problem of the type matching constraint, previously defined in section 5.8.1, the interface has a DESCRIBE stage. This specifies the data types of the attributes being retrieved. Information from the describe operation is placed in the SQL/DS data descriptor area (SQLDA). (see following example query for illustration of the description area). Once we have the type information it is possible to allocate or map pointers in the description area to program

variables of the same type (see figure 5.11). These pointers in PL1 are of the form

```
REF UNION ( LONG INT , INT , STRING )
```

Data items can now be retrieved via the SQLDA pointers into the program variables.

We now trace an example query processed by a generalised interface which is written in PL1 using the PL1 - SQL/DS interface.

EXAMPLE QUERY

From querying the string -

```
STRING = ' SELECT NAME,ADDRESS,NUMBER,AGE FROM STUDENT '
```

and passing it to the SQL descriptor we obtain the following information

SQLDA.SQLVAR.SQLTYPE	448
SQLDA.SQLVAR.SQLLEN	30
SQLDA.SQLVAR.SQLNAME	NAME
SQLDA.SQLVAR.SQLTYPE	449
SQLDA.SQLVAR.SQLLEN	50
SQLDA.SQLVAR.SQLNAME	ADDRESS
SQLDA.SQLVAR.SQLTYPE	500
SQLDA.SQLVAR.SQLLEN	2
SQLDA.SQLVAR.SQLNAME	NUMBER
SQLDA.SQLVAR.SQLTYPE	501
SQLDA.SQLVAR.SQLLEN	2
SQLDA.SQLVAR.SQLNAME	AGE

For each attribute there is also a pointer variable which specifies the location where the retrieved value will be stored.

SQLNAME defines the name of the attribute

SQLTYPE defines the type of the attribute

DATA CODE	DATA TYPE	INDICATOR VARIABLES
501	SMALLINT	YES
500	SMALLINT	NO
449	VARCHAR	YES
448	VARCHAR	NO

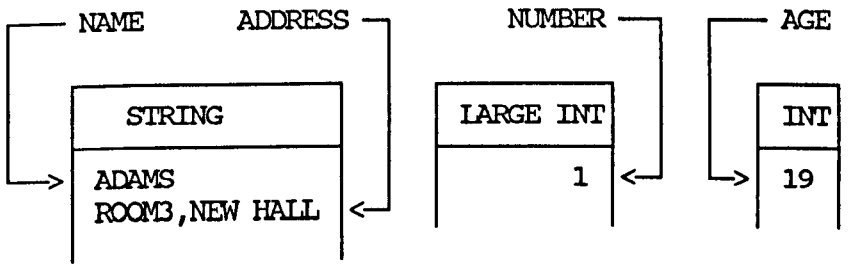
SQLLEN defines the attribute's store requirement in bytes

A full definition of the codes used in the SQL/Description area can be obtained from the SQL/DS manual [IBM 1984a].

EXAMPLE MAPPING

From the describe stage we can obtain all the information to produce the following mappings onto the internal program arrays of type STRING, LARGE INT and INT which are used to store the retrieved tuple attributes.

SQL pointer values are shown by arrowed connections.



MAPPING FUNCTION

TYPE	ARRAY POINTER	SQLDA SQLNAME
CA	1	NAME
CA	2	ADDRESS
IIA	1	NUMBER
ISA	1	AGE

figure 5.11

SQL/DS - PROLOG

This connection is similar to the SQL/DS - PL1 connection. DBMS queries can be passed in string format but the result is returned in a Prolog list structure. Again the DESCRIBE statement can be used to obtain the types of the retrieved attributes, although the DESCRIBE statement is not so important in Prolog as uninstantiated variables are type-less.

INGRES - CPROLOG

This connection is similar to the SQL/DS to PROLOG interface, although far simpler in its capabilities. It has been developed under UNIX and relies heavily on the facilities provided by UNIX, such as the piping of I/O. The Ingres call takes the QUEL query command in string form, and delivers a simple description of the retrieved results along with a retrieved tuple. Further requests retrieve other tuples which satisfy the initial request. This continues until all satisfying tuples have been retrieved.

5.8.4 - Comparison of dynamic interfaces in PL1 and Prolog

We have shown that the type constraint is a cause of difficulty for many systems in the handling of dynamic queries. PL1 overcomes this problem by the use of pointers, which can reference or map to a variable of the required type. Prolog overcomes the type constraint problem by its inherent structure of uninstantiated variables which are type-less. The elimination of the requirement for a mapping function, previously defined, makes querying in Prolog far easier than in PL1.

A description of the results is imperative for dynamic querying where the ordering of the retrieved results is not known in advance. Both communication links have access to the description area, which defines the data to be retrieved, and the Ingres system retrieves a description list whenever it retrieves a tuple

In comparing integrity control and locking mechanisms, it appears easier in PL1 with logical work units, whereas Prolog with its backtracking seems to makes it harder to enforce integrity control. However, as we are only considering querying systems the problem of integrity is not so important.

If we consider the interfaces for our purpose of constructing a communication link between an expert system and a database then it is more advantageous to use Prolog, rather than PL1. It is far

easier to design expert systems in Prolog, with its in-built inference engine and rule-based structure, than it is in PL1.

5.8.5 - Comparison of the implementations of interfaces

One of the major problems in building a combined system is customization [Damerau 1985]. We have tried, in all the coupling strategies that have been considered, to overcome this problem by layering the interfaces into generalised levels. Consider the following diagrams (figures 5.12 and 5.13) of two specific examples

PREDEFINED ALL TUPLE RETRIEVAL (FORTRAN-RAPPORT)

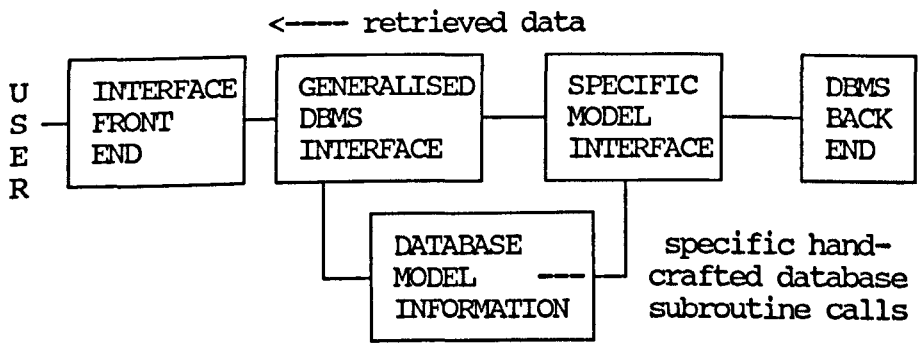


figure - 5.12

FULLY DYNAMIC QUERYING (SQL/DS-PL1)

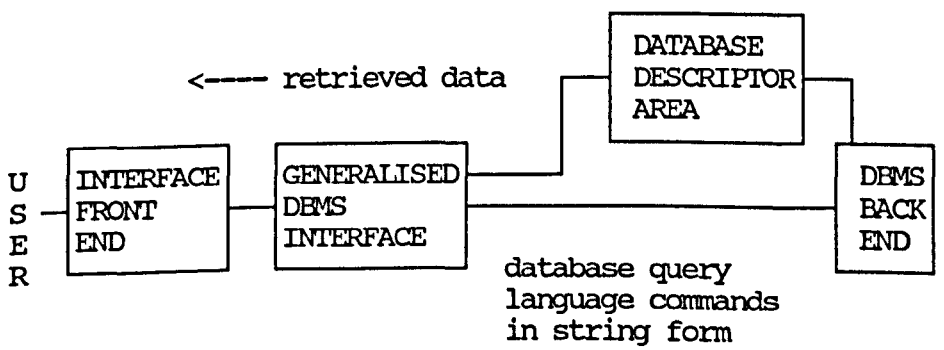


figure - 5.13

When fully dynamic querying strategy is employed it can be seen that there is no need for precompiled hand crafted subroutines. A mechanism for passing commands in string form directly to the DBMS is provided. This means the interface does not need to be customised for a particular model.

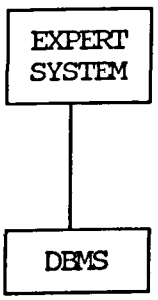
5.9 - The need for traffic control when using the communication link

Having analysed and developed several methods, or mechanisms, for communicating between expert systems and database management systems we are now able to identify further problems that may arise in the operation of such communications links.

The communications link which offers the best facilities for a single unified Expert System to Database Management System combination is that of a tight coupling. This type of coupling gives the interface programmer the facility to generate database system queries during the runtime of the interface. Thus the interface programmer has the opportunity to fully utilise the efficiency of the underlying database management system.

However, this efficiency can easily be lost if the communications link becomes a bottle-neck due to excessive use caused by poorly specified queries or lack of control over the inferential search technique of the expert system.

SIMPLISTIC DYNAMIC CONNECTION
(NO SYSTEM CONTROL OR QUERY OPTIMISATION)



No Traffic Control. The ES can repeatedly issue the same or as many poorly specified DBMS queries as it wants.

figure - 5.14

The problem of lack of traffic control could be crippling for the overall performance of the information system. If the performance was degraded so much as to make the coupled system unusable then the communications link would have failed to satisfy the second of our two requirement rules for the communications link (see section 5.5).

Dynamic coupling places the onus for the performance of the resulting information system on the interface programmer. It is his ability or knowledge of how to fully utilise the database management system's look-up or retrieval facilities which will determine the combined system's performance

5.9.1 - Poorly specified database management systems calls

In order to reduce the volume of information retrieved from the database system we should attempt to constrain the database queries as much as is possible. These constraints must not exclude any of the required tuples. This increase in constraint specification not only reduces the number of tuples retrieved but also gives the database management system more opportunity to perform indexed or optimised look-ups. Poorly constrained query specifications may lead to expensive linear searches of the database.

The efficiency problem in query evaluation is caused by using the inferential search of the ES rather than the efficient but rigid search technique of the DBMS. This problem can be illustrated by considering the possible specifications of an example inquiry, which use the dynamic SQL/DS to Prolog link.

For the following employee relation

EMPLOYEE

Name	Address	Age	Year_service

figure - 5.15

consider the following -

QUERY: fetch all information on all employees who have the name A SMITH or B JONES

In logic the specification of this query is not unique so we can specify it in several forms

Version - 1

```
employee(Name,Address,Age,Year_s) <-
                sql('select * from employee',
                     [Name,Address,Age,Year_s]).

person(Name,Address,Age,Year_s) <-
                employee(Name,Address,Age,Year_s)
                and
                     ( Name ='A SMITH'
                       or
                       Name ='B JONES' ).
```

Alternatively we could specify the previous query in the following form (although in terms of performance it is less well stated)

Version - 2

```
employee(Name,Address,Age,Year_s) <-
                sql('select * from employee',
                     [Name,Address,Age,Year_s]).

person(Name,Address,Age,Year_s) <-
                employee('A SMITH',Address,Age,Year_s)
              or
                employee('B JONES',Address,Age,Year_s).
```

If we were to specify the query in a database query language it would be of the form

```
select *
from employee
where
     name = "A SMITH" or
     name = "B JONES"
```

Using the dynamic query mechanism available, the optimum way to specify the query in logic and fully utilise the efficient retrieval power of the DBMS, would be

Version - 3

```
person(Name,Address,Age,Year_s) <-
                sql('select * from employee
                     where name = "A SMITH" or
                           name = "B JONES" ',
                    [Name,Address,Age,Year_s]).
```

The first and second versions access the database system via the predicate "employee" which itself calls the "sql" predicate which then carries out a linear search of the relation. The first logic specification is therefore merely a linear search of all the records in the database relation "employee" with Prolog handling the matching. The second specification is an even more wasteful double linear search of the same relation and again Prolog does the matching. The third specification is the most efficient as it is an indexed look-up which takes full advantage of the database management system's accessing strategies.

The best performance is therefore achieved when the database management system access is constrained by as many clauses as is possible. These constraints may be increased using domain knowledge. Constrained retrievals give the database management system more opportunity for using its own optimising techniques for determining the access strategy of the shared data. This will enable the system to perform efficient look-ups rather than it having to perform an expensive linear search. The straightforward linear search vastly reduces the benefits to be gained from using a database management system as opposed to a collection of records in files. The general guide given by John Miles Smith [1984] is that

> "The ES should only be used for those cases of inferencing where the power of its search mechanism is really needed. In other cases, simpler search mechanisms should be used. In particular, the DBMS should be delegated maximum responsibility for searching shared information".

This need for the interface programmer to be aware of intricacies in the system performance is reminiscent of the first uses of virtual memory store where, to gain the best use and performance from the system, a programmer was required to specify his own overlays.

The automatic solution of this problem is very difficult due to the unpredictability of the dynamic calls to the database system. The optimisation of the operations following the call, whose relevance to the dynamic query is only known at runtime, is extremely complex. The only real aid to help alleviate this problem is to give the interface programmers guide lines on how best to use the communications link and how certain specification may effect efficiency.

5.9.2 - Repeated database management systems calls

Another problem with the inferential search technique which causes traffic problems is the "trial and error method". This occurs where a logic predicate is repeatedly specified, so defining all the acceptable forms by which it can be satisfied. This querying operation may be considered to be of the "or" form i.e. try this specification of the predicate "or" if it fails then try another specification.

We can therefore write the previous example query as a double predicate definition thereby eliminating the explicit "or"

```
employee(Name,Address,Age) <- sql('select * from employee',
                                  [Name,Address,Age]).

person(Name,Address,Age) <- employee('A SMITH',Address,Age).
person(Name,Address,Age) <- employee('B JONES',Address,Age).
```

So even if we were to use only well defined query calls we could still have excessive communications traffic.

The following query is an example of a well formed database system query but one which is repeatedly called because of the poor definition of the logic predicate.

Query - How many days holiday is person "A SMITH" entitled to ?

```
Rules for holiday entitlement

Holidays 4 weeks (= 20 days) + 2 days for each years service up
to a total holiday entitlement of 6 weeks
(= 30 days)

Holiday_entitlement(30,Name) <-
              form_query( 'select * from employee
                           where name = "',Name,'"',Q) &
              sql(Q,[Name,*,*,year_service]) &
              year_service >= 5 & /.
Holiday_entitlement(Days,Name) <-
             form_query( 'select * from employee
                           where name = "',Name,'"',Q) &
             sql(Q,[Name,*,*,year_service]) &
             Days := 20 + ( year_service * 2 ) .
```

So the above query would be specified as

```
Holiday_entitlement(Days,"A SMITH") .
```

The "holiday_entitlement" predicate therefore makes a double call to the underlying database if the first predicate has failed. This

problem of repeatedly issuing the same query request may be accentuated by the recursive calls of a predicate. A method which has been suggested by Sciore and Warren [Sciore & Warren 1986] to overcome this problem is asserting the latest query calls in the Prolog database. It is envisaged that this would be similar to a small cache store of the form first in first out. However the duplicate storage of information which may be especially volatile would require careful management, as discrepancies may occur between the values stored in the database and those held in the cache store.

5.10 - Enhancement of the dynamic communication link using Prolog

Having previously outlined the problem of excess traffic and the need for traffic control in communications between expert systems and database management systems, we now propose an automated method to improve this communications link and hence reduce the traffic. We have analysed and timed several database queries for the SQL/DS to VMPROLOG link. This has enabled us to assess both the current problem and the effect of using our improved DBMS call. The analysis has involved calculating the virtual cpu time used to resolve each query, first for the "sql" predicate provided and then for our improved sql predicate "bsql". By comparing the results we can illustrate the need for the binding of database calls and estimate the value of the predicate "bsql".

5.10.1 - The example queries

In order to improve efficiency we attempted to identify the main types of queries which caused the current system to be inefficient. By analysing several example queries we were able to compare and contrast many of the aspects associated with Prolog to DBMS communication.

The following table lists the queries which were analysed.

TEST NUM	SQL QUERY
1	sql('select * from student where number = 6',[A,B,C,D,E],Er).
2	sql('select * from student',[A,6,C,D,E],Er).
3	sql('select * from student',[A,B,C,D,E],Er) & B = 6.
4	sql('select name,number from student where number=6',[A,B],Er).
5	sql('select name,number from student where name=''FABER''',[A,B],Er).
6	sql('select name,number from student',[A,B],Er) & A = 'FABER'.
7	sql('select * from student where number=6',[A,B,*,*,*],Er).
8	sql('select * from student',[A,6,*,*,*],Er).
9	sql('describe select * from student',A,Er).
10	sql('describe select name,number from student where number=6',A,Er).
11	sql('select name,number from student where number = 6',A,Er).
12	sql('select * from student where age = 54',[A,B,C,D,E],Er).
13	sql('describe select name,number from student where number=6',Des,Err) & sql('select * from student where number = 6',[A,B,C,D,E],Er).

Analysing the timing results for the above queries enabled us to compare our hypothesis of the communications issues which we initially believed to be important. These issues were:

1 - Comparing search strategies

Query 1 and Query 2 both retrieve the same tuple but query 1 uses the DBMS search strategy whereas 2 uses the Prolog matching technique.

2 - Current optimisation of DBMS calls

Query 3 also uses the Prolog matching strategy but this is performed outside the DBMS call. If the timings for Query 3 and Query 2 are similar then we have shown that the sql predicate does not take advantage of the assigned Prolog variables to utilise the DBMS search technique.

3 - Selected attribute retrieval

Query 4 and Query 5 test if there is any significant difference in retrieving only selected attributes rather than the entire tuple.

4 - Non key field retrieval

Query 5 and Query 12 test whether there is any loss in time if it is not the key field which we are retrieving on. (The key field in this example is the student number).

5 - String matching versus numeric matching

Query 5 and Query 6 test to assess the difference in matching strings rather than numeric values.

6 - Retrieval into anonymous variables

Query 7 and Query 8 test if anonymous variables effect the retrieval time. Values retrieved from the database into anonymous variables are discarded.

7 - Variable list versus specified list

Query 12 tests if using a single list rather than a specified item list (i.e. "A" instead of "[A,B,C,D,E]") has any effect.

8 - Non select commands

Query 9 and Query 10 gives us an indication of the timing of other sql commands, that is commands other than the select command.

9 - Composite DBMS calls

Finally query 13 gives us an indication of the effect on the timing of performing two sql commands in the same predicate.

5.10.2 - The timing of the DBMS calls

To measure the effect of the issues outlined we needed to time each of the queries specified above. We applied them to the following relation.

STUDENT

NAME	NUMBER	ADDRESS	AGE	TUTOR

figure - 5.16

The STUDENT relation had only seven tuples but this was still sufficient to illustrate the traffic problem.

To get a reasonable mean time we took 12 timings of the virtual cpu time used by 1000 iterations of each of the above DBMS calls. (test 0 was the timing for the 1000 iterative loop without any DBMS call) The mean results in milliseconds were as follows

GRAPHICAL REPRESENTATION OF QUERY TIMINGS

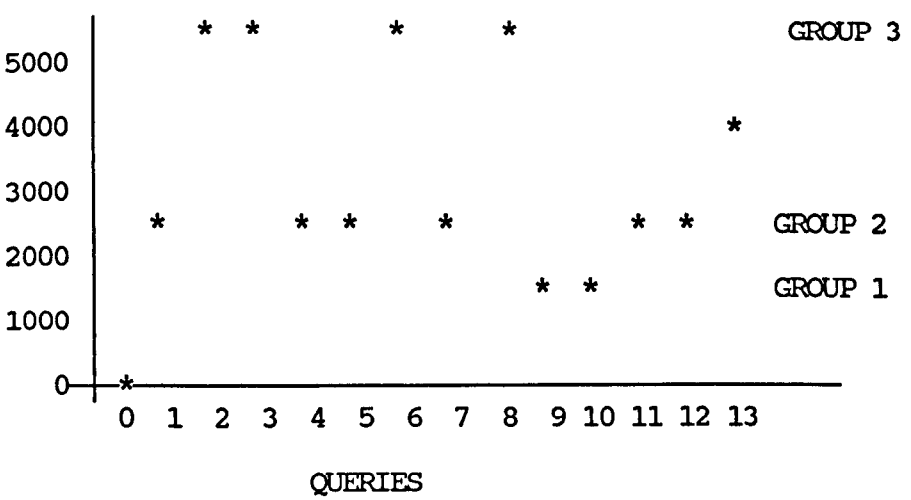


figure - 5.17

See Appendix E for actual timing figures.

We took two sets of timings, one at the weekend when there were very few users on the system, the other during a busy weekday. The marginal difference in the two sets of timings gives a good indication that the timings are a reasonable representation of the algorithm time rather than the swapping or paging algorithms or any other operating system overhead associated with time sharing.

It is easy to see from the results that the tests divide into the following three groups

GROUP	QUERY	TIME (Ms)
GROUP 1	9, 10	1500
GROUP 2	1, 4, 5, 7, 11, 12	2500
GROUP 3	2, 3, 6, 8	5500

figure - 5.18

(test 13 is a composite test of two tests from group 1 and 2)

By looking at the actual test calls we find that these groupings are fairly predictable

GROUP 1 - These are calls to the DBMS in which no actual tuple is retrieved.

GROUP 2 - These are all select statements with a condition part, that is they are constrained look-ups with the DBMS performing the matching process.

GROUP 3 - These are all select statements of the same relation, and in most cases the same tuple as those in group 2. However, in this group all of the attribute matching is performed by the Prolog system.

It is no surprise to see that the DBMS search technique is faster than the Prolog one. However, what is surprising is the fact that even for such a trivial relation the difference is so large.

We can therefore conclude it is advantageous to use the DBMS search as much as is possible.

The traffic problem of retrieving anonymous variables is shown not to be so significant so we will not concentrate on a remedy for this. Instead we will look at a method for automatically making greater use of the DBMS search technique, while reducing the use of the Prolog matching strategy.

5.10.3 - The binding of assigned variables

Having outlined the areas where improvement can be made we proceed to develop an automated method for making such improvements.

By considering the two Queries 1 and 2 we can demonstrate how our method improves the current interface.

```
1   sql('select * from student where number=6',[A,B,C,D,E],Er).
2   sql('select * from student',[A,6,C,D,E],Er).
```

As we know that the second field of the relation STUDENT is the NUMBER attribute we can semantically equate these queries. However, comparing the query times we see that Query 1 is evaluated in approximately half the time of Query 2. This has identified the need to develop a predicate which will translate a specified retrieval list value into a DBMS constraint. By performing this translation we utilise the DBMS search technique rather than the Prolog system match. By considering the previous result timings we see it is only worth binding to select statements which have no "where" part, and obviously we can only perform this binding when at least one of the values to be retrieved is known at the time of the DBMS call (i.e. the retrieval list is not an unassigned list or a list of unassigned variables).

The code to perform the required variable binding is contained in Appendix D.

5.10.4 - Comparing the bound and the normal DBMS call

Using this new version of the DBMS call, the "bsql" predicate we performed tests identical to those previously carried out. This gave us the following results

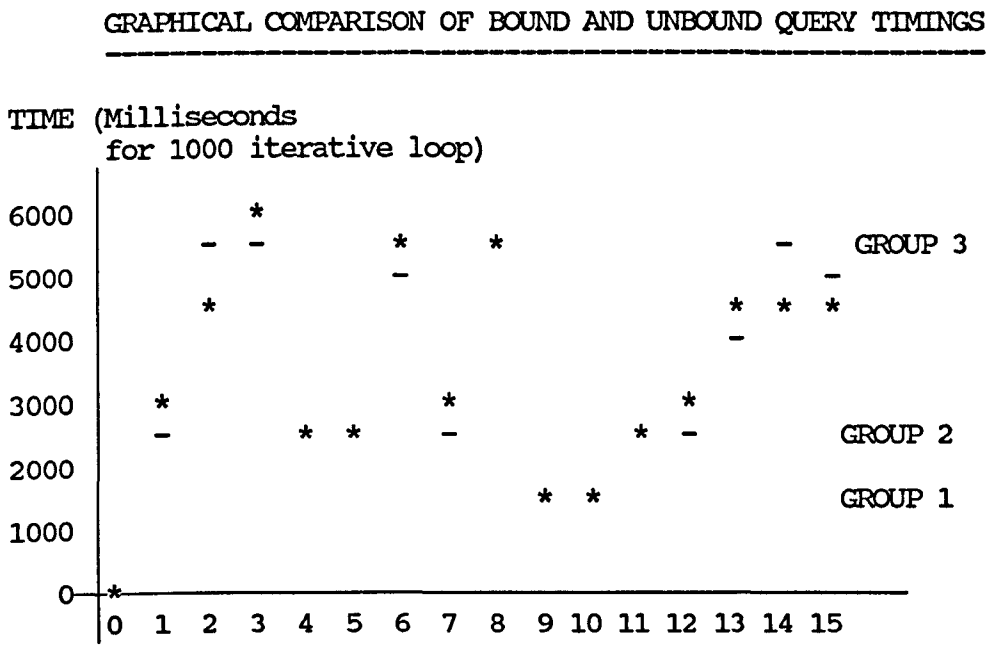


figure - 5.19

"*" represents virtual cpu time for 1000 iterations of the "bsql" predicate for the indicated query.

"-" represents virtual cpu time for 1000 iterations of the "sql" predicate for the indicated query where they are different from the "bsql" timings.

By comparing the results of the normal and bound calls we can estimate the advantage to be gained from using a binding on our seven tuple example relation.

From the results we can draw the following conclusions for each of the three groups. (See Appendix E for actual timing figures)

Group 1

For non select statements there is an approximate five percent performance loss. This is the overhead of testing to see whether the command is a "select" command.

Group 2

For "select" statements which already have a "where" part there is a mean degradation in time of approximately eight percent. Again this is because of the overheads caused by testing.

Group 3

The change in execution time for queries classified in this group varied. Some queries showed substantial improvements while others were showed a degradation in performance. The degradation in execution time occurs when the system is unable to produce a constraining "where" part. This is usually because the retrieval list is either an unassigned list or a list of unassigned attributes. The overhead for attempting to form a "where" part and failing causes an approximate degradation in time of seven percent. However, when the system is able to form a conditional part we can obtain a significant improvement in time of approximately sixteen percent (This improvement was for the trivial seven tuple relation, for relations containing more tuples this improvement will be considerably greater).

5.10.5 - The effect of predicate order on DBMS access time

The degradation problem in queries 3 and 6 of group 3 can be alleviated by reordering the query. If we perform the retrieval test before we perform the DBMS call, then in effect we are assigning to the retrieval list and so we know the attribute value when the DBMS is called. This allows us to take full advantage of the DBMS search technique. We can illustrate this by reordering the two queries as follows

```
   3  sql('select * from student',[A,B,C,D,E],Er) & B = 6
New3  B = 6 & sql('select * from student',[A,B,C,D,E],Er)
 AND
   6  sql('select name,number from student',[A,B],Er) & A = 'FABER'
New6  A = 'FABER' & sql('select name,number from student',[A,B],Er)
```

Using these reordered queries the time results are as follows

REVISED RESULTS SUMMARY

TEST QUERY	NORMAL	BOUND	REORDER BOUND	DIFFN	%DIFF
3	5634	6034		-400	-7.10
6	5248	5632		-384	-7.32
New3	5634		4489	1145	20.32
New6	5248		4303	945	18.01

figure - 5.20

Hence by optimally ordering the queries categorised in group 3 we can obtain an improvement in virtual cpu time of between sixteen and twenty percent.

The reordering process of performing the test before we execute the query, works because a Prolog test on an unassigned variable is really an assignment to that variable. Therefore by performing the test we are assigning a value to the Prolog variable. This value can then be used to constrain the database call. From the above results we can conclude the peculiar sounding programming rule, that:

> "All tests should be performed on the attributes of the relation to be retrieved, before any tuple is actually retrieved".

By doing this it allows us to use the DBMS search facility which substantially reduces the virtual cpu time used.

5.10.6 - The binding method for retrieval from larger relations

The real value of this binding can be seen when we run it against a larger relation. Using a large system catalogue, which has over 3000 tuples, and holds references to the computer manuals available from the University of Liverpool computer laboratory, we performed tests with examples taken from each of the three groups previously categorised.

The following table lists the test queries analysed for this new relation

TEST NUM	SQL QUERY
1	sql('select * from qpcat.catsingles where scatno = 3598', [A,B,C,D,E,F,G,H,I],Er).
2	sql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,3598,I],Er).
3	sql('describe select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],Er).
4	sql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],Er) & H = 3598.

Due to the magnitude of the virtual cpu time used to resolve these queries, instead of the 1000 iterative loop we only used a 10 iterative loop. This has caused truncation problems with measuring the faster routines and has led to the puzzling result that test zero took no time at all.

RESULTS SUMMARY 3000 tuple relation
(truncated to nearest millisecond)

GROUP	TEST	NORMAL	BOUND	DIFFN	%DIFF
	0	0	0	0	
2	1	28	30	-2	-7.14
3	2	15185	50	15135	99.67
1	3	14	15	-1	-7.14
3	4	16772	17041	-269	-1.60

figure - 5.21

These last results, although only from a small sample, show the dangers of using the ES inferential search on a large database. By automatically binding assigned variables we can sometimes provide sensationally superior times (over 99 percent improvement). However, poorly specified code will still cause problems. The benefits of

the binding method will be even greater in situations where predicates are used to perform a relational join.

5.11 COMPARISON OF TIMINGS FOR DIFFERENT COUPLING STRATEGIES

We can now compare the performance of our proposed system against some of the other coupling strategies which we have outlined.

To time these methods we will use the Computer Manual database and by using a query which requires a join we will accentuate any performance difference.

The techniques we will compare are the snapshot method, where we down-load the database into the Prolog system, the currently available SQL/DS to Prolog dynamic interface, and our improved form using the binding.

The example query we have chosen is:

> Retrieve all books published in the same year and by the same author as book with reference number 3000

This query has a simple logic representation of

```
cata(A1,B,C,D1,E1,F1,G1,3000,I1) &
cata(A2,B,C,D2,E2,F2,G2,H2,I2).
```

where cata is a predicate which retrieves tuples from the catalogue of manuals and attribute B is the author, C is the publication year and H the reference number. (The motive for representing a relation as a predicate will be discussed in the next chapter).

The Snapshot Coupling:

If we down load all the tuples and assert them as data objects of the predicate "cata", then applying the above look up takes approximately 175 milliseconds (This ignores initial set up time).

The Current Unconstrained Dynamic Coupling:

By defining the predicate "cata" as follows

```
cata(A,B,C,D,E,F,G,H,I) <- sql('select * from catalog',
                               [A,B,C,D,E,F,G,H,I],Err).
```

Then applying the initial look up takes approximately 3468 milliseconds.

The Bound Dynamic Coupling:

By defining the predicate "cata" using our bound database call as follows

```
cata(A,B,C,D,E,F,G,H,I) <- bsql('select * from catalog',
                                [A,B,C,D,E,F,G,H,I],Err).
```

Then applying the initial look up takes only 14 milliseconds.

These examples illustrate the great performance advantages to be obtained from exploiting the efficient search strategy of the database system. This combination of systems can improve the operation of the Prolog environment when reasonably large volumes of data are involved. In our example we reduced the VM/Prolog system's search time from 175 ms to 14 ms, which is a time reduction of 92 percent.

5.12 - SUMMARY

We have outlined several architectures and concepts for coupling Expert Systems and Database Management Systems. We have shown that fully dynamic querying is by far the best method. Fully dynamic querying allows the most efficient use of the DBMS without compromising the execution of the Expert System.

Although fully dynamic querying is the best method to use, it is still worth considering the other methods since, if we are to produce a DBMS independent system, we may need to communicate with existing DBMS' which do not provide the features required for fully dynamic querying.

We have shown that a simple connection of the two systems may result in a system which is unusable because of its inefficient performance.

We then proceeded to develop a method for controlling database access and hence improved the overall system performance.

Finally to illustrate the improved performance of our controlled database access method we compared its performance to several of the other methods we had described. This comparison should how by using the efficient look-up routines of a DBMS we could improve the operation of the Prolog environment.

CHAPTER - 6

INTERFACE SPECIFICATION

In chapter five we described several possible external architectures which could be adopted when connecting an expert system, acting as an interface assistant, to a database management system. In this chapter we describe the internal architecture of an interface assistant and specify a domain representation language which is to be used by this interface. The specific design of this interface assistant is based on the general observations made in chapter three and makes direct use of the proposed logic representation of the user view.

6.1 - A GENERALISED USER INTERFACE SHELL

In earlier chapters we have described many of the advantages to be derived from an interfacing system which aids naive users. These advantages can be very briefly summarised as:

Savings in time and money on training people to use both the DBMS system and the specific database model.

Increased use of the database information by casual users who would not normally be able to access the information directly.

These advantages are offset by the cost of writing such interfaces. It would be unlikely that an interface assistant which is dedicated to one database model could recoup its costs. A more profitable approach would be to develop a generalised interface which could easily be adapted for different domains [Damerau 1985]. We have therefore striven to produce an interface system which has a "Shell" type of architecture. Such shell like systems provide a generalised interface to the relational model but remain independent of any one particular domain.

To construct such a system it is necessary to identify the general information components which are required to fill the interface shell, and so enable the system to function adequately.

6.2 - AN ANALYSIS OF THE TASKS TO BE PERFORMED BY THE SHELL

The primary function of the interface shell is to help users to resolve their enquiries and thereby improve the usage of the database system. To achieve this the interface has to simplify the query process by removing the intricacies with which users of database management system are confronted (see section 2.2). The process of simplifying the query operation can be divided into the following five general tasks

1. Understand what the user wants.

2. Translate the system's interpretation of the users' queries into the terms of the database model.

3. Represent the database model query in the concepts of the database management system.

4. Code and pass the query to the database system so as it may be executed.

5. Supply the user with the results.

We can represent these five tasks diagrammatically as is shown in figure 6.1.

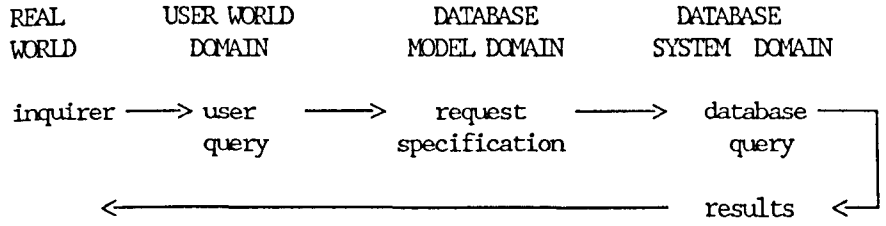


Figure 6.1 : The Distinct Stages of Translation

Having outlined the primary tasks we can now specify the modules which form the shell. These modules represent the three different domains and the translations between them (figure 6.1). The model representation languages used within these modules should provide a more procedural definition of the domains than is usual for schema

representation languages. By "procedural definition" we mean that these languages should not merely state the existence of concepts within the distinct domains but should represent how these concepts relate to the concepts in the other domains. These languages therefore code the actions required to translate between the domains. By encapsulating a procedural aspect in the representation of the domains it allows the interfacing system to execute these coded representations directly when the system translates between the domains.

The interface assistant we have developed is independent of any particular user view or relational model. This system known as IRIS (an Independent Relational model Interface System) merely provides a framework or a shell which can be "programmed" or primed with domain knowledge to produce a domain specific interface. The domain knowledge required for such a system can be categorised as:

Information describing a general user's interpretation of the real world domain, the user view.

Information defining the data stored in the underlying database with definitions of the data structures, the database model.

Information describing the semantic connections or translations between the user view and database model

Information relating to the specification of well-formed database queries

These four knowledge components are required by the interface system in order that it may perform the first four of the five primary task which we have previously outlined.

The interface shell can thus be viewed as a generalised interface manager with "slots" for these specific knowledge components. The structure of the shell is shown in fig 6.2.

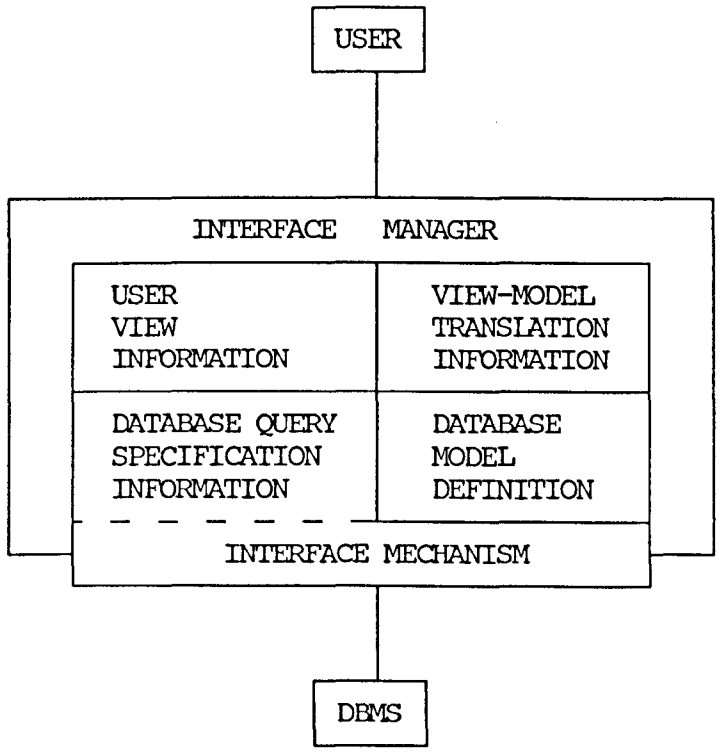


figure 6.2 : The Interface Shell

It is the function of the interface manager to control all interactions with the user and all of the accesses that are made to the domain information during the interaction. The interface manager shields the user from not only the intricacies of the underlying database system, but also the internal operations and domain representations of the interface shell itself. The interface manager is also responsible for initiating all accesses to the underlying database system. However, the interface manager does not directly interact with the database management system itself, instead all communications between the interface manager and the database management system are performed via the interface mechanism.

As described in chapter five (section 5.2), by using an interface mechanism we provide the interface shell with a level of independence from the underlying database system.

Two of the component slots of the interface, the "database query specification information" and the "interface mechanism" (figure 6.2) are specific to the underlying database management system. They are tightly linked together as their functions overlap. Hence, they may be considered as a single module.

Having outlined the general internal architecture we are now able to specify a language for each of the module components. These languages will allow each module to represent its particular aspect of the domain. The system we have developed is written in VM/Prolog

[IBM 1985] and the module language definitions we have used are all based on the mixed VM/PROLOG notation.

6.3 - Representing the user view information

To perform the first task of the interface, query understanding, we need to be able to interpret the entire scope of the request as it is perceived by the user. By understanding a user's query the system is able to understand not only how to satisfy the query but also to determine whether the system is capable of answering the query. By considering the following query we can illustrate the system's need for an understanding of the users conceptualisation of the domain in order that it may correctly resolve the user's queries.

Example Query

> What are the names of all people living in London who are older than 27 and are personal tutees of Dr Smith.

By analysing this query we can reveal many of the hidden assumptions made by the user. If we are to interpret this query correctly then the user's implicit knowledge, relating to these hidden assumptions, needs to be represented and hence made explicit. It is this implicit knowledge which we now seek to represent.

6.3.1 - Representing Implied Scope

If our example query was applied to a simple database situation then the word "people" may simply refer to a relation, as illustrated in figure 6.3.

PEOPLE

NAME	ADDRESS	AGE	PERSONAL_TUTOR

figure 6.3

However, it may not be as straightforward as this. The context of the query may imply that the questioner has in mind a subset of the relation "person", such as all "adults" or as is the case in our example "students". Alternatively a user reference to an entity may be a reference to a superset comprised of several subset entity groups, which the user may conceptualise as forming a united superset entity.

In order to understand scope intended by the user the primary entities and their subset entities have to be explicitly defined. This is a task for the interface programmer in consultation with the users. If the database is to be structured as suggested in chapter four then the interface programmer should either consult the DBA or perform the function of a DBA. The following coded representation is an extract from the user model for the Course_Teacher_Student domain (chapter four, figure 4.8), which specifies the primary and subset entities and the relationship between them.

```
entity person .
entity sponsor .
entity department .
entity faculty .

entity X if X direct_subset_of Y .
```

The definition of entities includes both the explicit definition of primary entities and the information that subset entities themselves can also be considered as entities.

```
professor           direct_subset_of  teacher.
lecturer            direct_subset_of  teacher.
senior_lecturer     direct_subset_of  teacher.
teacher             direct_subset_of  university_employee.
secretary           direct_subset_of  university_employee.
university_employee direct_subset_of  person.
student             direct_subset_of  person.

X subset_of Y if X direct_subset_of Y .
X subset_of Y if X direct_subset_of Z and Z subset_of Y .
```

The definition of subset entities includes the explicit definition of direct_subset relationships. The definition also states that

> An entity is a subset of all entities which have as a subset the entity for which it is a direct_subset_of.

Thus:

```
lecturer subset_of person
```

These definitions define the scope of reference of the users query as it should be perceived in the domain. Thus, when a user refers to person he is referring to an entity that may belong to the subset class university_employee or student. These subset groups may

themselves be further refined to a more specific sub-group, so that the person entity referred to may belong to the subset class secretary or teacher. This process of possible decomposition can be continued until the base entities, which have no further subsets are encountered.

Inversely if the user refers to the entity group lecturer then the user is referring to an entity which is a teacher and is a university_employee and is a person. The scope of a reference to the entity lecturer therefore inherits all attributes and relationships associated with entities for which it is a subset_of.

6.3.2 - Representing Implied connections

Phrases such as "living in", "older than" and "personal tutee of" imply connections between or within relations

e.g.

```
PERSON      living_in      ADDRESS
PERSON      older_than     AGE
STUDENT personal_tutee_of TEACHER
```

In the example query the connections "living_in" and "older_than" are both intra-relational connections, that is they refer to attributes within a relation. Implied connections become more complex when they are inter-relational, that is they imply a connection with another relation or another instance of the same relation.

For example

PERSON lives with PERSON

"lives with" is therefore an inter-relational connection. To satisfy such connections involves using joins. The implied connection "personal_tutee_of" is also an inter-relational connection, as it connects the two entities under_grads and teachers.

As with the representation of the implied user scope we also have to make the user's interpretation of the entity connections explicit. The following representation is a extract from the coded definition of the user view for the Course_Teacher_Student domain.

```
connect( attends, student, course, n, m, attended_by).
connect( aged, person, age, 1, m, '').
connect( older_than, person, age, 1, m, '').
connect( younger_than, person, age, 1, m, '').
connect( named, person, name, 1, m, '').
connect( resident_at,student, home_address, 1, m, '' ).
connect( resident_at,person, address, 1, m, '' ).
connect( teacher_to, teacher, student, n, m, taught_by).
connect( supervisor_to, teacher, post_grad, n, m, supervised_by).
connect( personal_tutor_to, teacher, under_grad, n, 1,
                                                personal_tutee_of).
```

N.B. The phrase "living in" is defined as a synonym for the phrase "resident at"

The first of the above connect statements can be read as:

The user perceives the existence of a connection called ATTENDS which is between the entity student and course. It is believed that

a student ATTENDS many (n) courses

and that

a course is ATTENDED_BY many (m) students.

Similarly, the user perceives the existence of a connection called PERSONAL_TUTOR_TO which is between the entities teacher and under_grad. It is believed that a teacher is PERSONAL_TUTOR_TO many (n) under_grads and that an under_grad is a PERSONAL_TUTEE_OF one and only one (1) teacher.

CONNECT STATEMENT		
NAME	attend	personal_tutor_to
BETWEEN	student course	teacher under_grad
TYPE	n m	n 1
INVERSE_NAME	attended_by	personal_tutee_of

figure 6.4

Although in our example only one link ever exists between two entities, it is possible to have many different links between the same two entities. For example if all students had personal tutors then there would exist two different links "teaches" and "personal_tutor_to" between the two entities teachers and students.

Perceived intra_entity references need additional information if they are to be resolved. This additional information states that the connection is an intra-entity connection and also defines the

comparison type. The following code is the definition of several of the intra-entity connections.

```
intra_entity(older_than,age,">").
intra_entity(younger_than,age,"<").
intra_entity(aged,age,"=").
intra_entity(named,name,rule).
intra_entity(resident_at,address,rule).
intra_entity(resident_at,home_address,rule).
intra_entity(born,date_of_birth,"=").
```

Thus the intra-entity older_than refers to the attribute "age" and has the comparison type ">".

The use of the comparator "rule" denotes that a coded predicate expression exists for comparing values for that attribute e.g. the predicates "name" or "address".

The use of synonyms such as "resident_at" instead of "living_in" is described in section 6.5.2.

The explicit definition of these connections also helps in the understanding of the scope of the query. Refer back to the initial example query:

> What are the names of all people living in London who are older than 27 and are personal tutees of Dr Smith.

We are able to deduce that the "people" being referred to all belong to the subset under_grad as in the view definition it is only under_grads who satisfy the implied connection "personal_tutees_of".

As under_grad is a subset of student we also know that the entity being referenced is a student.

From knowing the entity of reference is a subset of both student and person, the system is able to deduce that two possible interpretations of the implied connection "living_in" exist. The conflict in understanding the user's intended meaning is caused by the system not knowing whether the implied connection refers to the attribute person.address or the attribute student.home_address. The system having identified that this conflict exists then proceeds to resolve it by further interaction with the user.

6.3.3 Representing Composite Object Expressions

It is important for a system to recognise the partial expression of attributes. In the above example query a reference is made to the attribute ADDRESS via the intra-entity connection "living_in". However, the attribute ADDRESS is a structured attribute which is composed of several attributes. Thus if it were to attempt to satisfy the constraint

Address = London

it would almost certainly fail to match with any address that is stored. In order to match such partial expressions we require a means of specifying acceptable partial expressions of composite attributes. The equals comparator should not be limited to a literal

equivalence, instead it should allow for a semantic match. For the attribute ADDRESS this semantic equivalence could be defined as:

```
Address == House / Street / Town / County / Country
        or       Town / County
        or       Town
        or       County
        or       Country

"/" means concatenates with using a space separator
```

In order to perform semantic marching we first need to identify such composite attributes and specify which attributes they are composed of. For example:

```
composite(name,[title,firstname,surname]).
composite(address,[street,town,county,country]).
```

Thus NAME is defined as a composite object which is composed of the attributes TITLE, FIRSTNAME and SURNAME. Having identified these attributes we then need to define the rules to allow for the evaluation of acceptable combinations e.g. SMITH MR is not an acceptable form of NAME.

The rules to define the acceptability of partial attribute expressions are stated as logic predicates. The following rules, which are written in VM/PROLOG, define the acceptable forms that the composite attribute name can take.

```
namerule(St,Sf,Ss,S) <- stconc(St,' ',S1) &
                        stconc(Sf,' ',S2) &
                        stconc(S1,S2,S3) &
                        stconc(S3,Ss,S).
namerule(St,Sf,Ss,S) <- stconc(St,' ',S1) &
                        stconc(S1,Ss,S).
namerule(St,Sf,Ss,S) <- stconc(Sf,' ',S1) &
                        stconc(S1,Ss,S).
namerule(St,Sf,S,S).
```

Thus the following four predicates will succeed

```
namerule('MR','JOHN','SMITH','SMITH').
namerule('MR','JOHN','SMITH','MR SMITH').
namerule('MR','JOHN','SMITH','JOHN SMITH').
namerule('MR','JOHN','SMITH','MR JOHN SMITH').
```

However the predicates

```
namerule('MR','JOHN','SMITH','SMITH MR').
namerule('MR','JOHN','SMITH','SMITH JOHN').
```

will fail to match.

A similar predicate "addressrule" is used to match values for the attribute "address".

6.4 - REPRESENTING THE DATABASE MODEL

Having outlined a representation for the user view we can now proceed to consider the second knowledge component, the database model (figure 6.2).

The relational model is already formally defined in the database system's internal schema so representing it in the interface shell is a fairly trivial process.

6.4.1 - Representing relations

The basic definition of the data model requires a description of all of the relations, their key fields and a list of all of the attribute fields. This is achieved using the relation statement.

```
relation(person,[num],[num,title,firstname,surname,
        date_of_birth,street,town,county,country,sex]).
relation(student,[num],[num,year,h_street,h_town,
        h_county,h_country,nationality,attendance_type]).
relation(under_grad,[num],[num,personal_tutor]).
relation(course,[code],[code,session,level,teacher]).
relation(attend,[student,course],[student,course]).
```

The first predicate "relation" defines the relation "person" to have the key field "num" and attributes num, title, firstname, surname, date_of_birth, street, town, county, country and sex. The following table (figure 6.5) further illustrates the use of the "relation" predicate.

RELATION STATEMENT		
NAME	KEY	ATTRIBUTE LIST
person	num	num,title,firstname,surname,sex, date_of_birth,street,town, county,country
under_grad	num	num,personal_tutor
course	code	code,session,level,teacher

figure 6.5

The key elements are duplicated in the attribute list primarily to improve system performance. The improvement in performance is

achieved by eliminating the need to append the lists everytime the full attribute list is required.

To allow the user to refer to structured attributes in the same way that they refer to non-structured attributes we also specify an "image_relation". Obviously, this is only required for relations which have composite attributes. Thus:

```
image_relation(person,[num],[num,name,date_of_birth,
                             address,sex]).
image_relation(student,[num],[num,year,home_address,
                             nationality,attendance_type]).
```

6.4.2 - Attribute definition

In order that the system is able to recognise a certain attribute by its value, we need to define the format that the attribute values may take. e.g. consider the query

> Fetch the names of all people who are male

It is obvious from our knowledge of the world that "male" refers to the attribute "sex", although no explicit reference is made to the attribute. Such format information is made explicit by the use of an attribute format statement.

```
attribute(num,integer,6).
attribute(name,composite,namerule).
attribute(date_of_birth,fchar,8).
attribute(address,composite,addressrule).
attribute(sex,one_of,[male,female]).
attribute(year,one_of,[1,2,3,4]).
attribute(personal_tutor,reference,teachers).
attribute(nationality,one_of,['UK','EEC',rest_of_world]).
attribute(session,one_of,[first,second,third]).
attribute(level,one_of,[one,two]).
attribute(teacher,reference,teachers).
```

Such attribute descriptions can be summarised as follows.

ATTRIBUTE STATEMENT		
ATTRIBUTE	ATTRIBUTE TYPE	TYPE DESCRIPTION
name	composite	namerule
sex	one_of	male, female
personal_tutor	reference	teachers

figure 6.6

The interpretation of the TYPE DESCRIPTION depends on the ATTRIBUTE TYPE. See Appendix G for a full list of attribute types and the corresponding interpretation of the type description information.

The attribute description information gives both a syntactic and pragmatic description of the attributes in the data model. For a semantic description of the whole database model, we need to examine the way the user interprets or translates the structures into his user view.

6.5 - REPRESENTING THE DOMAIN TRANSLATION SEMANTICS

The third knowledge component of the interface shell (figure 6.2) involves explicitly expressing the process of translation between the user view and database model. In section 6.3 when defining the user view, it was stated that the user perceives the existence of a connection "personal_tutor_to" between the entities teachers and under_grads. When defining the data model it was stated that the relation under_grad has an attribute personal_tutor and that this attribute has type reference. However, the system has not been given an explicit definition of how to translate and hence satisfy such user perceived connections in the terms of the database model.

6.5.1 - Translating implied connections

In order that the interface system can understand the implied connections, we have explicitly stated them using a logic representation. We thus define the personal tutor connection as:

```
personal_tutor_to(Teacher_num,Student_num) <-
                    under_grad(Student_num,Teacher_num).
```

Thus the interface is now able to resolve the user's perceived connection personal tutor without requiring the user to navigate through the database model, i.e. the system can simply call the defined predicate "personal_tutor_to".

Other links are similarly defined

```
lectured_by(Course,Teacher) <- courses(Course,*,*,Teacher).

teacher_to(Teacher,Student) <- attend(Student,Course) &
                               lectured_by(Course,Teacher).
```

The connection "attends" is already represented as a relation and so does not need to be represented as a connection. The connection "teacher_to" uses the previously defined connection taught_by. Thus

> a teacher is a teacher_to a student if the student attends a course which is taught_by the teacher.

Using such explicit representations the system is able to code the user's implied query without direct reference to the data model.

6.5.2 - Recognising object descriptions

Included in the translation information is a minimal language translation information component. This includes phrase descriptions of database terms such as:

```
database_term('university employees',university_employees)
database_term('personal tutor to',personal_tutor_to).
database_term('lectured by',lectured_by).
```

Also in this language section there is definition of plurals e.g.

```
plural(student,students).
plural(address,addresses).
plural(sex,sexes).
plural(grad,grads).
plural(graduate,graduates).
plural(street,streets).
plural(person,people).
plural(country,countries).
```

and synonym definitions to match with the database model naming conventions.

```
synonym(department,dept).
synonym(departments,depts).
synonym(tutees,under_grads).
synonym(tutors,teachers).
synonym(people,persons).
```

This language translation information is used to translate the phrases of the user query into the terms of the database model, whereas the connection translation information is used to translate the user concepts into the relational structures of the database model.

6.6 - Query formulation

As we previously stated, the query formulation information and the interface mechanism can be considered as a single module as their functions are so tightly linked.

Using the bound Prolog - SQL/DS link, which we described in chapter five, section 5.10, the interface shell is able to extract tuples from the database using predicates to represent relations. Thus the query

Fetch the person named Mr Smith

would be coded as:

```
person(Num,'MR',Firstname,'SMITH',Date_of_birth,
                    Street,Town,County,Country,Sex).
```

where the predicate "person" is defined as:

```
person(Num,Title,Firstname,Surname,Date_of_birth,Street,Town,County,
       Country,Sex) <-
                    bsql('select * from person',
                         [Num,Title,Firstname,Surname,Sex,
                         Date_of_birth,Street,Town,County,Country]
                                                        ,Error).
```

The simple specification of entities as logic predicates leads to poorly specified database queries. The problems caused by such poorly specified queries are outlined in section 5.9. However, by using the binding method outlined in section 5.10 we are able to reduce the effects of this problem. It is the function of the query formulation component (figure 6.2) to perform any bindings. Thus the previous query would result in the following bsql call

```
bsql('select * from person',[Num,'MR',Firstname,
          'SMITH',Date_of_birth,Street,Town,County,
              Country,Sex],Er).
```

From the information in this call the query formulation component will be able to formulate the following constrained SQL command

```
select * from person
where title='MR'
      and surname='SMITH'
```

It is then the task of the interface mechanism to pass this query to the database management system using the sql predicate call.

Tuples which satisfy the retrieval request can then be passed back into the interface shell. These tuples can then be displayed to the user via the interface manager (figure 6.2).

6.7 - SUMMARY

This chapter has described the internal architecture of the generalised interface shell and the specific knowledge components which make up the shell. A representation language to be used by the components of the interface shell has been specified (see Appendix F). When defining the system we have drawn on many of the ideas outlined in previous chapters, such as the logic representation of the user view and the design of interfacing mechanism. The practical implementation of such a shell type of interface has enforced these ideas of using logic languages such as Prolog for an interface specification language. In Appendix I a full listing of the interface code is given. The specific code for the Course_lecturer_Student model is given in the module ALTER. This module is obviously altered for different domain situations. Although the types of representation language predicate remain constant.

CHAPTER - 7

THE APPLICATION OF THE INTERFACE SHELL

In early chapters we considered the deficiencies of current database systems for naive users. We have outlined how interfaces of the expert system type could help alleviate these difficulties. In order to provide such interfaces we have described the physical external architecture for connection to a database management system. We have also proposed a relational design method which simplifies the tasks that the interface is required to perform. In the previous chapter, chapter six, we described the internal structure of the interface shell which we have developed. In this chapter we wish to combine all of the proposals we have made and justify them by demonstrating the operation of the resulting interface shell during extracts from several actual interactive sessions with a user. This will enable us to highlight many of the features which we believe greatly simplify and hence improve the interactive process for naive users. Appendix H shows an interactive session in full i.e. from the initial user query through to the user receiving a reply. We acknowledge that to create a complete system, the user domain interpretations which we have described, would require greater initial syntactic language parsing.

7.1 Domain querying

Most queries seek to elicit specific facts, but some seek more general information. In many cases a naive user will use the same

form to express both types of query. In the first mode a typical exchange would be:

Question: What is John Smith doing on Mondays at 10 am?

Answer: Course 02CS

The second mode is a meta-level query or domain query i.e. a query about the domain rather than about specific facts. e.g.

Question: What is John Smith doing on Mondays at 10 am?

Answer: Attending a lecture

It is most important that naive users should have the ability to query what is actually being represented in the database since, by definition, they may be unfamiliar with its structure. This information is coded in the domain information components defined in chapter six. For such initial querying, when a user wishes to confirm or ascertain the system's representation of the modelled domain, the simple structure and operation of a prompted-input-type dialogue system is most appropriate. We believe the requirement for simplicity outweighs the problem of inflexibility associated with such systems (see section 2.4.3). We therefore decided to use this approach to implement queries about the domain so that users can be guided by the system towards the domain knowledge which has been stored.

When executed, the domain querying environment provides the user with a choice of primary entities about which the user is able to

make further enquiries. By pursuing an enquiry, the user can ascertain all subsets of an entity that the system recognises, and all perceived connections which the system is aware of for that entity. The attribute definitions can also be queried allowing the user to find out the format of the values stored in an attribute. In this way a user can gain an overview of the domain and if required obtain more detailed information regarding the system's representation of the domain.

7.2 - Database querying

Although domain queries are important, they are relatively infrequent. Hence the alternative mode, database querying, is the default mode of querying for the interface shell. Database queries have a limited language type of interaction. Our experience has shown that the vast number of possible queries that a user can request make the simpler prompted input or menu-based system unusable for this type of query. Prompted input systems are suitable only for initial interaction or situations where the number of possible queries is manageable.

As outlined in section 6.2 the process of assisting naive users to perform database querying can be divided into five basic tasks. By considering examples of user queries we will consider possible ways in which these tasks could be performed and demonstrate the specific way in which they are performed by the interface shell we have developed.

7.2.1 - User Request Analysis

When a user inputs a database query the system attempts first to identify the terms and phrases of the query which it understands. Consider the following example query

QUERY: Get the names and ages of all people who attend 01CS or teach 01CS

After the removal of plurals and synonyms, and identification of the database terms, the analysis of the query gives the following information

COMPOSITE_ATTRIBUTE	NAME
CONNECTOR	AND
ATTRIBUTE	AGE
ENTITY	PEOPLE
CONNECTION	ATTEND
UNKNOWN	01CS
CONNECTOR	OR
CONNECTION	TEACH
UNKNOWN	01CS

From this description the system attempts to associate attributes with entities to form objects. In order to form such objects the system must first define areas of context. Context is important in understanding a user query as it defines which entity is currently being referred to implicitly.

		CONTEXT
COMPOSITE_ATTRIBUTE CONNECTOR ATTRIBUTE ENTITY	NAME AND AGE PEOPLE	 PERSON
CONNECTION UNKNOWN CONNECTOR	ATTEND 01CS OR	STUDENT COURSE
CONNECTION UNKNOWN	TEACH 01CS	TEACHER COURSE

figure - 7.1

From the user view definition of inter-entity connections, such as connect(attend,...) (see section 6.3.2), the system is able to recognise that the user perceived connection ATTEND, connects the entities STUDENT and COURSE. From its user view information the system also recognises that STUDENT is a subset of PERSON. Thus the system is able to deduce that the entity PERSON is "in context" for the query prior to the reference to the inter-entity connection ATTEND, and that after ATTEND the entity COURSE is the "in context" entity.

From the above example we can see that when a sub-set entity is the "in context" entity then all of its super-set entities are also in context. Thus when STUDENT is in context PERSON is also in context and when TEACHER is in context PERSON is still in context. This factor of sub-entity context facilitates the concept and

implementation of sub entity inheritance. This allows the system to associate the subset entities with the attributes and connections of their super-set entities.

Having determined the context phases of the query and associated the attributes with these context entities, thereby forming the objects, the system is able to deduce the subject of the query (i.e. the retrieval list). The language forms of the above query are fairly simple and it is easy to obtain the subject for this query as being the objects PERSON.NAME and PERSON.AGE. This is obviously a trivial example but adequately shows the process of context evaluation.

7.2.2 - Condition / Restriction identification

Having obtained information about the objects that the user wants retrieved,the system proceeds to ascertain on what conditions the retrieval is based. The precedence of the "or" operator splits the constraints into two sections. Each of the two sections corresponds to a section of a UNION type select statement in SQL.

If the first section of the constraint list is:

```
PERSON IS A STUDENT   AND
STUDENT ATTEND '01CS'
```

then the initial condition that

PERSON IS A STUDENT

is a subset enforcement constraint. Such constraints are necessary as they not only produce the correct evaluation of the query but also vastly reduce the search space by constraining the query. As described in section 5.9, when using systems which incorporate some form of inferential search technique it is important to reduce the search as much as possible and as soon as possible.

In the example query the specific term '01CS' is unknown to the interface system. It may be a stored item in the database. If it is a stored value then the system must match it against the attribute which it is stored in. Obviously, it would not be possible for the system to check every unknown term against the entire contents of the database. Such a matching process to ascertain the associated attribute would be not only exceedingly time consuming but also un-informative if a user were asking about an instance of an object which does not exist. For example, if a user asks if there is any information about course XYZ, and course XYZ does not exist, then the system would have no means of matching XYZ to an attribute. It is therefore up to the system to query the user as to whether the user **believes** that the term is stored in the database. Having ascertained that the user does believe the item is stored, then its type is changed from "unknown" to "stored" (see table 7.1). When attempting to understand the intended meaning of a query it is irrelevant whether or not an item which the user believes is stored in the database is actually stored. Values which the user believes

are stored need to matched against the corresponding attributes which they are believed to be stored in.

When matching a "believed" stored value to an attribute the system first attempts to match the stored term against the key attributes for the entity which is in context. Having explicitly defined the format types of attribute values with the attribute statement, (see section 6.4.2) the system is able to test the unknown item to see if it is of the same type. For example, to the system the following three queries appear similar.

QUERY1 Get the names of all people who are older than 20
QUERY2 Get the names of all people who are older than John Smith
QUERY3 Get the names of all people who are older than 2347193

The system has to match the formats of the three believed stored items "20", "John Smith" and "2347193" with possible attribute formats. Query1 is easily matched to the AGE attribute. However Query2 and Query3 require a join to via PERSON and then match to the attributes NAME and ID_NUMBER respectively. (For a full listing of the type matching process see Appendix I module TM).

In the example query in section 7.2.1,

QUERY Get the names and ages of all people who attend 01CS or teach 01CS

the system would find a direct match with the key attribute CODE of COURSE. The CODE attribute is defined as a four character fixed length string, which matches the format of the item. If the format type of the CODE had failed to match with key elements then we would have had to consider the other possible attributes of the entity in context.

Thus this query has been transformed into the query

```
Fetch  PERSON.NAME and PERSON.AGE
subject to
    person is a student and student attend course and
    course's code is equal to 01CS
 or
    person is a teacher
and
    teacher teach course
and
    course's code is equal to 01CS
```

It is only this representation of the query that the user sees. All of the other representations of the query are internal to the interface manager, figure 6.2, which shields the user from seeing them.

A user may reject a parse if the displayed representation does not conform to the one intended by his initial query. When a parse is rejected the system will back-track and re-evaluate the assumptions which have been drawn during the initial query parse.

7.3 - Query Formulation

Having divided the query into its two parts, the retrieval part and the conditional part, we can now proceed to specify the query as a rule or a logic query_predicate. As with all predicates the query predicate has a head and a body. The head specifies the retrieval list and the body the constraint list.

7.3.1 - Forming the query predicate head

Using the previously formed retrieval list we first decompose all composite attributes into their base attributes. In our example, the attribute NAME is a composite attribute composed from the base attributes

title, firstname, surname

Thus our revised query list becomes

[title,firstname,surname,age]

This list is then processed to change the elements into Prolog variables and a numeric value is appended to each of them to make them unique. Thus the query predicate head becomes

```
query_pred([Title0,Firstname0,Surname0,Age0]) <-
```

The requirement for uniqueness of variables is needed as another entity, or in our example another person, may be referred to in the query. For example, consider the following query.

Fetch the names and ages of all people older than Norma Smith

The NAME specified in the query retrieval list is certainly not to be matched with the implied NAME "NORMA SMITH". The use of numeric values allows the system to distinguish between different occurrences of an entity. Thus in this case we would have the same predicate head

```
query_pred([Title0,Firstname0,Surname0,Age0]) <-
```

i.e. retrieve name and age of person0

while the constraint list would be defined as:

```
    age of person0 > age of person1
  and
    person1 named 'Norma Smith'
```

Such entity distinction is imperative for the successful evaluation of the query rule.

7.3.2 - Forming the query predicate body

The predicate body defines the constraints which the retrieved information must satisfy if it is to conform to the user's initial query. The predicates which form the body must represent the clauses of the constraint list.

Consider the constraint list for the query outlined in section 7.2.

```
    person is a student
  and
    student attend course
  and
    course.code = '01CS'
```

The first clause of this query defines the entity, for which values are to be retrieved as belonging to a subset. To represent this concept in logic we specify the two relations STUDENT and PERSON as predicates and the key fields can then be matched. Thus the clause

person is a student

can be represented as:

```
student(Num0,Home_street0,Home_town0,Home_county0,Home_country0,
        Nationality0,Sponsor0,Attendance_type0,Year_of_study0)
      &
person(Num0,Age0,Title0,Firstname0,Surname0,
          Street0,Town0,County0,Country0,Sex0).
```

The system matches Num0 so that only the entity members of the relation person who are defined as "student" will be returned.

The subset entity is specified first as it will always have the same or a smaller search space. By knowing the value of the Num attribute the system is able to to perform an indexed look-up and so optimise the search of the larger predicate "person".

In section 6.5.1 we described how the translation of user perceptions could be explicitly represented as predicates. For inter-entity connections we simply specify these as the connecting predicate with arguments that are the key fields of the entities that are being connected. Thus the defined connection

STUDENT ATTEND COURSE

is represented as

attend(Num0,Code1)

Where NUM is the key attribute of STUDENT and CODE the key attribute of COURSE.

To resolve intra_entity connections the system needs to consult the user view information, which is defined in section 6.3.2. This information tells the system how to resolve the connections. For example given the following query

Fetch names of people older than 50

the intra-entity connection information expresses the hidden knowledge that "older than" refers to the attribute AGE and that it is of comparison type "greater than". Thus it enables the system to represent the query as

```
query_pred([Title0,Firstname0,Surname0]) <-
        person(Num0,Age0,Title0,Firstname0,Surname0,
                Street0,Town0,County0,Country0,Sex0)
        &
        gt(Age0,50).
```

If the intra-entity connection had been between two entities, instead of having a literal value, e.g.

Fetch names of people older than person with id num 100001

then the system would still require the access to the definition of "older than". The only difference would be the change of entity occurrence over the intra-entity connection, i.e.

```
query_pred([Title0,Firstname0,Surname0]) <-
                person(Num0,Age0,Title0,Firstname0,Surname0,
                              Street0,Town0,County0,Country0,Sex0)
               &
                gt(Age0,Age1)
               &
                person(Num1,Age1,Title1,Firstname1,Surname1,
                              Street1,Town1,County1,Country1,Sex1)
               &
                eq(Num1,100001).
```

A slightly different representation is required for intra-entity connections which have a comparator defined as a "rule". This occurs for connections relating to composite objects and hence requiring partial expression evaluation, as described in section 6.3.3. Consider the following query

> Fetch the names and ages of all people older than Norma Smith

Firstly the system has to recognise that a new entity is being specified. From an initial query parse the system realises that the string "Norma Smith" does not satisfy the format specification of the attribute AGE (see section 6.4.2). From this the system infers that the intra-entity connection may be between two entities. It then proceeds to parse the query under this assumption as follows

```
person0   older_than     person1   Norma Smith
Entity    Intra-Entity   Entity    Unknown
```

As AGE is an attribute of PERSON the system tries to match the "unknown" to an attribute of PERSON to identify the entity occurrence. Matching the string format to the attributes of PERSON we have two possible matches, NAME or ADDRESS. The system chooses the NAME attribute. However, if it had chosen the ADDRESS attribute

the user would have rejected the parse and the system would then re-try using the other possible attribute.

From the successful parse the system is able to augment the query specification to include the following information:

```
person0 Age0  older_than   Age1  person1  Name1  Norma Smith
Entity  Att   Intra-Entity Att   Entity   Att    Unknown
        ----------------------                  ----------------
            condition ">"                        condition "rule"
```

It can then represent the query in the following logical form:

```
query_pred([Title0,Firstname0,Surname0]) <-
                person(Num0,Age0,Title0,Firstname0,Surname0,
                               Street0,Town0,County0,Country0,Sex0)
               &
                gt(Age0,Age1)
               &
                person(Num1,Age1,Title1,Firstname1,Surname1,
                               Street1,Town1,County1,Country1,Sex1)
               &
                namerule(Title1,Firstname1,Surname1,'NORMA SMITH').
```

Here "namerule" is a predefined predicate which matches partial expressions of the composite attribute NAME. The arguments for such predicates conform to a fixed pattern, with the front variables corresponding to the base attributes of the composite attribute, and the last argument being a string representing the value to matched

Many of the conditional comparisons specified by the user during the interaction will not be predefined, but must still be recognised and represented by the system. For example in the query

the system must not only recognise that the unknown value "34CS" is an identifier of the entity COURSE but it must also match it to the attribute CODE of COURSE. Once such conditions have been recognised and the literal values have been matched to their respective objects, the system then has to represent the conditions. When matching identification attributes which have no specified condition comparison type then they default to "equals" and so the built in predicate "eq" is used.

The previous query can be defined in the predicate form as:

```
query_pred([Num0]) <-
            under_grad(Num0,Personal_tutor0)
          &
            person(Num0,Age0,Title0,Firstname0,Surname0,
                        Street0,Town0,County0,Country0,Sex0)
          &
            student(Num0,Home_street0,Home_town0,Home_county0,
                        Home_country0,Nationality0,Sponsor0,
                            Attendance_type0,Year_of_study0)
          &
           person(Num0,Age0,Title0,Firstname0,Surname0,
                        Street0,Town0,County0,Country0,Sex0)
          &
           attend(Num0,Code1)
          &
           course(Code1,Session1,Level1,Teacher1)
          &
           eq(Code1,'34CS').
```

To execute the query we merely assert the predicate query_pred then call it as

```
query_pred(X).
```

The retrieved tuple's attributes are matched against the retrieval list before they are passed to the user.

The time cost of executing this logic query is not as bad as it first appears. By using the binding methods outlined in section 6.10, the system performs only one linear search of the relation under_grad. All of the remaining calls to the database are indexed key attribute look-ups which have query cost 1 in the SQL/DS system.

The inefficiencies which do exist in the query specification are caused by the distance between the retrieval list and the condition list. i.e. in the example there is no need to call the predicate STUDENT as none of the attributes of STUDENT are used. However, if the initial query had been:

Get names and home addresses of all under graduates who attend 34CS

Then the only difference would have been in the retrieval list and not in the constraint list.

For the above query the query head would be:

```
query_pred([Title0,Firstname0,Surname0,Home_street0,
            Home_town0,Home_county0,Home_country0])   <-
```

The lack of alteration that is needed to transform the query high-lights the way in which the query body models the user's own perceptions of the query and not the database model's. As the user's conditions of retrieval have not changed it is only the objects to

be retrieved which have been altered and so it is only the head of the query predicate which is affected.

7.4 - QUERY EVALUATION

Once the query predicate has been formed all previously asserted queries are retracted and this new query is then asserted. To execute the query all that the system is required to do is call the predicate query_pred(X). The tuples satisfying the query body are then returned tuple by tuple. When all satisfying tuples have been returned the query predicate fails.

When executing a query the system first checks the scale of the retrieval before it gives the user any retrieved information. This involves the system performing a COUNT operation on the successes of the query predicate. Often the user would require only the first tuple which satisfies the predicate and it would be very wasteful to retrieve many unwanted tuples. Therefore the system performs a count and does not exceed ten. Having tested the scale of the retrieval the system then informs the user that there are 0-10 or over 10 tuples which satisfy the given query conditions. The user can then either view each individual tuple one after the other or quit and respecify a new or more selective query.

A complete example of a user interaction is given in appendix H. This includes the corresponding logic representation of the query and for comparison a SQL translation of the query.

7.5 - THE NEED FOR MIXED DIALOGUE

The use of mixed dialogue is most important in interactive systems which deal with ambiguity. The ability for the system to stop and confirm with the user that it has made the correct assumptions is most advantageous as it avoids much of the wasted time in pursuing incorrect inferences (see section 3.2.4). A dialogue facility is useful when trying to understand terms which are unknown to the system (see Appendix H). However, it is important that the user dialogue is not used to excess as it may alienate the user and may also be excessively costly in respect of time. The over use of such a facility is particularly likely in systems which rely on back-tracking.

7.5.1 - THE USER PROFILE

In order to minimise the occurrence of excessive user dialogue, we have introduced a session record, which records the answers given by the users. By examining this record before consulting the user the system eliminates the need to ask the same question repeatedly. The session record is known as a user profile. The user profile builds up a picture of an individual user's terminology for the domain.

Thus if the system does not recognise the words used by the user it will ask for synonyms until it finds one it recognises or the user gives up. The user profile also stores the literal values which the user has specified in his query requests in the belief that they are stored in the database. The user profile can also be used to store any or all of the knowledge components described in chapter six.

It is possible to save and load the user profiles from one session to the next so as to maintain a permanent record of a user's queries, and eliminate much of the mundane dialogue needed for term verification.

The user profile gives the interface system a degree of user adaptivity. This adaptivity allows each individual user's profile to be modified during the interactive session. Basic user profiles can be provided for new users, and these can be specifically tailored for different types of user groups. Such tailoring can be beneficial as it can be used to reduce the expected domain of interest and

create a logical horizon for a user group. By reducing the expected domain of interest we reduce the available terminology and correspondingly reduce the search space for the interface assistant system. Reducing the search space greatly improves the system's performance.

Most of the user dialogue is related to attempts to understand terms which the system does not recognise. In the majority of cases these will be words which the user believes are stored in the database. As previously stated in section 7.2.2, when attempting to understand a user query it is of considerable benefit to know which terms the user believes to be stored in the database. When the system is made aware of such a term then it records this fact in the user profile. This eliminates the need for the system to ask the user repeatedly about the term. The need for the system to recognise all terms in the query helps reduce the risk of the system translating the query incorrectly.

7.6 - SUMMARY

The examples both in this chapter and in appendix H high-light many of the ways in which the encoded domain knowledge of the interface shell can be used to help translate user queries and so improve the usability of database systems for the inexperienced user. We have attempted to illustrate the user interaction with the system by showing the external responses a user receives and the internal processes performed by the interface shell. We have attempted to show how a user can phrase his request in the terms and concepts of his own personal user view, and the way the system attempts to maintain these conceptions in its logical representation of the query. The simplification of the queries due to the subset decomposition and the resulting inheritance capabilities justifies the advantages that can be gained from the relational design technique we described in chapter five. We have demonstrated the use of the domain translation knowledge, and the reason why it was specified in the form of directly executable logic predicates. The simple nature of the calls to the database relations, in the form of predicates, would be totally un-usable due to efficiency restrictions if it were not for the binding techniques we described in chapter six.

CHAPTER - 8

REVIEW AND PROPOSED DIRECTIONS FOR FUTURE WORK

In this chapter we summarise the work which we have carried out and high-light many of the important aspects. We also propose possible directions for future work. This future work includes both the practical work which could be carried out to improve the current version of the information retrieval system we have described in this thesis, and more generally the directions in which we feel the combination of information systems and logic programming should develop in the longer term.

8.1 - SUMMARY

8.1.1 - How an expert system can help

We have identified the problems which frustrate the use of existing database systems by naive users and also the ways in which expert systems, and in particular a logic representation, could be used to resolve these problems.

We have described the problems experienced by naive users who are required to specify their queries in a formal query language, and illustrated how these problems could be alleviated by using both a limited natural language interface and some form of inference to eliminate superfluous query specification.

Having identified the problems of navigating the relational model, we illustrated the simplicity with which Horn clause logic predicates could be used to specify user views. Such logical user views resemble the user's own concepts, rather than those of the relational structures. We demonstrated that, by having a representation which was conceptually closer to the user's own actual view, the task of translation could be simplified. Such findings confirmed Bundy's [1984] idea of distinct translation stages for intelligent front ends.

When reviewing the systems which have been proposed to assist in providing users with information, we stated the need for these systems to be interfaces to external database systems as opposed to systems with self contained databases. Such self contained systems failed to capture the potential benefits associated with the use of an existing database system.

Many of the early systems designed to assist naive users in retrieving information suffered from the problem of transportability. That is, the systems were specifically designed for a single domain. The use of predicate logic as a domain specification language enables the division of the knowledge requirements of the system into several components or modules. This allowed us to create a generalised "shell" structured interface which minimises the amount of effort required to specify an interface for a new domain. Simply by combining different modules we are able to expand the domain of knowledge of the system.

More recent retrieval systems have attempted to provide a degree of transportability. However, such systems have encountered great problems through the lack of representation of the domain semantics. Systems which rely entirely on language parsing have only a shallow understanding of the meaning of a user's request.

Having outlined the ways in which a logic representation of the user view of the domain could help, we needed a method to capture and specify the user view information. To achieve this we used an augmented version of the sub-entity modelling technique. This modelling technique enabled the modelling of the user's perceptions of inter-entity and intra-entity connections, the concepts of structured objects and the existence of subset entities and the inherited properties of such subset entities. We also outlined a method for producing a relational model from this user view which was closer to the users own conceptual view. This closeness simplified the task of translating between the representation of the user view and the relational model.

8.1.2 - Connecting an interface front end to a DBMS

Having specified the advantages of using a logic representation of a user view, we then proceeded to implement a system which provided a means of representing and using the logic of a user view to assist retrieval from a database system. This involved combining an intelligent front end system with a database system.

After considering several of the proposed design methods for coupling expert systems and databases [Vassiliou 1983] we decided on a combined form of architecture which would maintain the independence of both systems. This in turn required the provision of a communication link. Before developing such a link we outlined the following two rules which the communications link must satisfy

1. The communications link should enable the expert system to extract any item of data that is stored in the database which the interface user is allowed to access.

2. The use of the communication link should not impinge on the operation of either system so as to hamper the usage of either system in isolation or in combination.

For the system to take full advantage of the database facilities it was necessary to allow the dynamic specification of queries. To achieve this required a tight coupling of the two systems.

By implementing a tight coupling we exposed a system with an inferential search technique to vast volumes of data. In terms of performance this was potentially disastrous. However, by developing an appropriate binding technique between the two systems we vastly reduced the volume of communications traffic and so improved the system's overall performance.

8.1.3 - Specifying an expert system front end

When designing the user interface front end we pursued our ideal of a generalised interface. The interface architecture was designed as a shell which could be "filled" with separate distinct knowledge modules. The use of such modules simplified the process of transporting the system to a new domain or altering the current domain of interest. Such alterations could occur due to expansion of the represented domain or a change in one aspect of the domain, such as the underlying relational model.

In order to provide such modules we defined a generalised language which enabled the modules to represent the user concepts of the domain.

Having developed the interface shell and specified a domain representation for our typical example user view, we proceeded to demonstrate the ease with which a limited language understanding system, which used a simple phrase recognition technique, could code non trivial retrieval requests. This demonstrated the simplicity with which user requests could be translated into a predefined logical view and the ease with which such queries once specified in logic could be executed against a relational database.

8.2 - FUTURE IMPROVEMENTS OF THE INTERFACE SHELL

In this section we consider some aspects of the interface shell where it would be possible to improve the functionality of the current version.

8.2.1 - Improving the system's response

The shell currently answers queries by presenting the user with the results in a tabular form, which corresponds to the structure of the relational model. Such responses are abrupt and present the results in an artificial way. When implementing the interface shell, we were predominantly concerned with the understanding of what a user wants, so we have largely ignored the task of presentation of the results, which was the fifth of the five interface tasks, stated in section 6.2. The interface system's responses could be improved by interpreting the results back into the terms of the user model. To illustrate this consider the following example

QUERY: Who teaches 01CS ?

The system currently presents the reply to this query as:

Title	PROF
Firstname	ARTHUR
Surname	SMITH

Instead of this curt response the system could interpret the results back into the user model and, combining this with a paraphrased version of the original user query, the system could express the reply as

The name of the teacher who teaches the course which has code 01CS is
PROF ARTHUR SMITH

A form of reply such as this improves the user friendliness of the system and provides a degree of validity to the system's understanding of the original query.

8.2.2 - Improved meta-level querying

In chapter seven we explained how the shell allows querying of the domain information using a menu-driven dialogue. Our reason for using menus rather than limited language was that it gave a simple means of making initial queries about an unknown domain. For subsequent interactive queries of the domain a more in-depth form of querying may be required. In such circumstances it may prove beneficial to include an additional meta querying system which does allow limited language queries of the system's underlying semantic interpretation of the domain. To illustrate this consider the example from section 3.3.3 where we have defined the term holiday_entitlement for an employee database as:

```
holiday_ent(Name,Days) <-
                    employee(Name,*,Years_ser,*) &
                    calculate_holiday(Years_ser,Days).

calculate_holiday(Years_ser,30) <-
                    ge(Years_ser,10) & / .
calculate_holiday(Years_ser,Days) <-
                    Days := 20 + Years_ser.
```

Using this representation of the concept the system has the information to answer the following meta query

QUESTION: What is an employees holiday entitlement

ANSWER: 30 days if years service is greater than 10
or else
20 plus number of years service

The text for the answer is generated directly from the Prolog code representation of the concept. The representation is already stored in the system's domain information module. The problem of text generation from sources such as Prolog code is currently being investigated by Melish [1987].

8.2.3 - Providing semantic integrity

One of the major new problems in handling knowledge rather than data is the need for knowledge integrity or semantic integrity [Frost & Whittaker 1983]. This is in addition to the need for data integrity. As our logic specifications give us a semantic representation of the model we can use this representation to test for semantic integrity violations or contradictions.

Using Kowalski's [1978] logic definitions we can explicitly define a contradiction as an occurring set of conflicting events

```
contradiction( C ) :- Pred1 , Pred2 ,,, PredN.
```

where each predicate represents an event

To illustrate this consider the semantic concepts that a student is taught by a teacher, and that a course is taught by a teacher. Thus only teachers can teach whereas both students and courses can be taught. We can enforce this constraint as a contradiction, specifying that if there exists a situation where it is not a teacher who teaches then a contradiction has occurred. Similarly we can specify that if it is not a course or a student who is taught then a semantic contradiction has occurred. In logic we can simply specify this as follows

```
contradiction('student or course taught by non teacher') <-
                          taught_by(*,Teacher) &
                          not teacher(Teacher,*,*,*).
contradiction('non student or course taught by teacher') <-
                          taught_by(SC,Teacher) &
                          not ( student(SC,*,*,*,*,*)
                                or
                                 course(SC,*,*,*)   ).
```

Contradiction testing can be used to ensure the correct negation of objects (see section 3.4.4). Such contradiction predicates could be used to verify database updates. Semantic verification could easily be performed by the following expression:

```
contradiction(X) & write(X) & fail.
```

Obviously semantic testing would have to be run as a background task as it performs extensive searches of the entire database contents.

8.2.4 - Improved binding techniques

In chapter five we described a method for binding instantiated Prolog variables to constraints for database queries. This method of binding reduced the traffic between the two systems and so improved the performance of the overall system. In section 5.10.5, we stated the following rule for improving the specification of database look-ups in logic:

> All tests should be performed on the attributes of the relation to be retrieved, before any tuple is actually retrieved.

By adhering to this rule the system was able to use the binding technique for instantiated variables, thus:

```
Id_num = 12345 &
person(Id_num,Name,Address,Age).
```

allows the system to perform an indexed look up on the relation "person".

The binding method described was unable to handle any constraint other than equality. For example in standard Prolog [Clocksin 1981] it is not acceptable to perform a constraint on an uninstantiated variable. Thus

```
Id_num <= 12400 &
Id_num >  12345 &
person(Id_num,Name,Address,Age).
```

would not be allowed. This is because variables can not be instantiated to constraints such as "> 12345". Instead the query would have to be specified as:

```
person(Id_num,Name,Address,Age) &
Id_num <= 12400 &
Id_num >  12345 .
```

This does not adhere to the specification rule and so results in a costly linear search. This problem severely limited the initial binding technique.

However, we have recently developed a technique whereby numeric variables can be instantiated to constraints and such constrained variables can be used to further improve the traffic control and performance of the two systems.

To achieve this we have introduced the new comparators "gtn", "ltn", "gen", "len" and "eqn" (see Appendix I module QCON for a listing of these predicates). These comparators either perform a test or instantiate a variable to a numeric constraint. Rules have had to be defined to allow for the combining of such constraint operators. Using these operators we can specify the previous logic query as:

```
Id_num len 12400 &
Id_num gtn 12345 &
person(Id_num,Name,Address,Age) .
```

This new binding technique enables the specification of the constraints before the actual retrieval is performed and so allows the system to perform an indexed search of the relation. This

further reduces the volume of traffic and so vastly improves the combined systems' performance.

This new technique has not yet been incorporated into the interface mechanism for the generalised interface shell. Although the code for the conditional binding predicate csql is listed in Appendix I module CSQL.

8.3 - A NEW GENERATION OF EXPERT DATABASE SYSTEMS

In the previous section we looked at the facilities which could be added to the interface shell which we have developed. In this section we look further ahead to the many new opportunities which the inclusion of a logic semantic representation offers to the formation of Expert Database Systems (EDS).

8.3.1 - Semantic query optimisation

The semantic representation of the domain and relational models which we have provided can be used to improve the efficiency of storage and retrieval of the data. Such improvements in query optimisation can be achieved by the use of logical equivalence transformations, and also by the use of the additional semantic constraints available to the system. Chakravarthy [1986] describes how semantic query optimisation can help in two ways. The first is the elimination of the need for certain database calls, and the

second is an increase in the number of query constraints, which improves the efficiency of the retrieval operation by reducing the search space and allowing the database system to optimise the search query path. We can illustrate a simple example of query optimisation using the following model

SALES_STAFF

NAME	AGE	SALARY	DEPT	ADDRESS

DEPARTMENT

TITLE	LOCATION	MNGR.NAME

figure - 8.1

The first type of optimisation can be shown by the trivial query

Who works on the tenth floor

We can specify this as

department(Dept,10,*) & sales_staff(Name,*,*,Dept,*).

If we know there are only eight floors then we do not bother to consult the database and there is no need for a search of the department relation. Thus the semantic information can reduce the number of calls to the external database.

The second type of optimisation where semantic information improves the constraints of a query can be seen by considering the following query.

> Who works on the first floor and earns more than £ 40,000

We can specify this as

```
department(Dept,1,*) & sales_staff(Name,*,Salary,Dept,*) &
Salary > 40000.
```

If we know that the only department on the first floor is the furniture department, then we can specify this query without the need for a join. Thus the query could be specified as a single look-up and if we use the enhanced binding method described in section 8.2.4 then the retrieval can be performed as an indexed retrieval. The new specification of the logic query would be:

```
Salary gtn 40000 &
Dept = 'FURNITURE' &
sales_staff(Name,*,Salary,Dept,*).
```

Thus the domain information has simplified and hence improved the evaluation of the query executed by the external database.

8.3.2 - Fuzzy values / Intuitive queries

Another new facility which is made possible by the use of a domain semantic representation is the satisfying of intuitive queries. These queries are similar to the goal queries of Motro [1986]. In

order to satisfy intuitive queries we need some form of coding or representation of judgement. Consider the following intuitive query:

> What is the name of a person who can carry out a job which requires someone who is tall and heavy?

When it is applied to the following model:

PERSON

NAME	HEIGHT (cms)	WEIGHT (kg)
ADAMS	180	71
BLACK	175	73
CLARK	171	74
DAVIS	170	65
EVANS	160	70

figure - 8.2

We can define the concepts tall and heavy as

```
tall(X)  <- person(X,Height,_) & Height > 170 .
heavy(X) <- person(X,_,Weight) & Weight > 70 .
```

If the above goal wanted an optimised answer i.e. the best person for the job then we would need to represent some measure of 'distance' i.e. a means of quantifying the intuitive retrieval conditions "heavy" and "tall" and then assigning a weighting to the conditions to determine the relative importance of heavy or tall.

Depending on whether weight or height is the more important then there are two neighbourhood goal results and one optimised result for the above model

```
NEIGHBOURHOOD GOAL where weight is most important
          retrieved tuple          ('CLARK',171,74)

NEIGHBOURHOOD GOAL where height is most important
          retrieved tuple          ('ADAMS',180,71)

OPTIMUM NEIGHBOURHOOD GOAL
          retrieved tuple          ('BLACK',175,73)
```

The evaluation of intuitive queries is a major new area of database querying. The logic specification of semantic judgement concepts can be easily incorporated into our current domain representation. The understanding of such judgmental queries when regarded as logic views is similar to the understanding of exact concepts e.g. consider the following example queries

```
Fetch names of all people who are lecturers
Fetch names of all people who are tall
```

These can be represented in logic as:

```
person(Num,Name,*,*,*) &
lecturer(Num,*,*,*).

person(Num,Name,*,*,*) &
tall(Num).
```

The representation and execution of individual judgement goals is therefore relatively simple. However, considerable work is still required on the quantification of judgement goals in order that we may solve optimal neighbourhood goals and evaluate the overall optimal solution for the optimum neighbourhood goal.

8.3.3 - Data redundancy

The use of a logic representation of the domain can help alleviate the problems of redundancy, where stored data is dependent on, or can be calculated from, other items of stored data. Such redundancy causes difficulties both in wasted storage and in integrity control when updates are made. The use of an explicit representation of the domain can help resolve these problems. Consider as an example the following relational model

MANAGERS

Name	Age	Salary	Type
Adams	26	13000	Department Manager
Brown	46	23000	Group Manager
Clark	52	26000	Group Manager
Davis	40	20000	Section Manager

figure - 8.3

Suppose that we know that in this model the company's wage structure is such that:

Department Managers salary is less than or equal to 15000

Section Managers salary is between 15000 and 21000

Group Managers salary is greater than or equal to 21000

Then we can express this information in rules as expert data which defines the model domain e.g.

```
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary <= 15000 &
                        Type = 'Department Manager' / .
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary >= 21000 &
                        Type = 'Group Manager' & / .
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary > 15000 &
                        Salary < 21000 &
                        Type = 'Section Manager' .
```

Using these rules we are able to remove the need for explicit storage in the database of the attribute "TYPE".

In general such reduced relations can be specified as:

```
relation(A1,A2, ,,Ai,Aj,Ai+1, ,,AN) <-
           relation(A1,A2, ,,Ai,Ai+1, ,,AN) &
           constraint_function(A1,A2, ,,Ai,Ai+1, ,,AN) &
           instantiate(Aj).
```

The use of logic to represent data aids conventional databases as it reduces redundancy. However, using such semantic rules to represent the actual data requires special techniques for handling updates of both the data and the rules themselves.

8.3.4 - Inferring knowledge from data

The use of logic as described in section 8.3.3 to represent the actual data facts, can be extended to the description of generalised aspects of the data. Such a logic representation could be automatically derived from the actual data. Wiederhold [1986] identifies two main reasons for inferring such generalised data

descriptions. The first is the provision of "intelligent summaries of databases". This would help reduce the problem which he outlines of information overload. The second reason is that of extracting "new knowledge present implicitly in the data".

The use of a language which represents the semantics and concepts of the domain can help facilitate this task and provide a means of expressing any information that is inferred. To illustrate how such inferred knowledge may be used consider the previous relation of MANAGERS in a departmental store, where all the data is stored explicitly. We can attempt to infer the wage structure of the company, previously stated in section 8.3.3, by interpreting this data. Reconsidering the previous model, figure 8.3, the resulting inferred interpretation of the wage structure would be

```
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary = 13000 &
                        Type = 'Department Manager' / .
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary >= 23000 &
                        Type = 'Group Manager' & / .
manager(Name,Age,Salary,Type) <-
                        manager(Name,Age,Salary) &
                        Salary = 21000 &
                        Type = 'Section Manager' .
```

Obviously more data would improve our implied knowledge. However implied knowledge can be misleading as it is specific to the known facts at a given instant. Considering the example above we could infer that:

```
manager(Name,Age,Salary,Type) <-
                    manager(Name,Age,Salary) &
                    Salary := Age * 500 .
```

Clearly this rule, though correct at the present time, does not have the same permanent status as the previously expressed rules.

The inferred rule is true for the model at a precise instant of time. However, such rules may be purely coincidental and have no lasting validity.

The ability to infer knowledge from factual data is an important new area. To facilitate future work in this area is required a language which can represent both the inferred knowledge and the domain semantics, so that more in-depth inferences can be drawn. By the term in-depth we are primarily referring to inferences involving inter-entity connections within the domain, as in the above example inferences were all drawn from a single relation. Such domain semantics would be similar to those modelled in the domain representation of the interface shell.

8.3.5 - An improved expert system environment

In our attempt to simplify the process of data retrieval for naive users we have created an efficient link between a logic based environment and a database system. Using this link we have vastly improved the performance of the logic environment when accessing large volumes of data. As stated in section 5.11 there was a

performance improvement of over ninety percent when using the combined system as opposed to using the Prolog system in isolation. The efficient handling of large volumes of data is only one of the many advantages that can be acquired by expert systems when they are linked to database systems. We see this link as the essence of a complementary existence for both expert systems and database systems. As database systems themselves evolve so expert systems can benefit from their improvement. This process of complementary existence can currently be seen with the improvements in distributed databases. If expert systems are able to fully utilise this advance then they may provide a means by which expert systems can themselves operate in a distributed manner, thereby transparently sharing information and knowledge throughout a network of expert systems.

8.4 - Conclusion

In this thesis we have been predominantly concerned with the creation of a naive user interface for a relational database. In demonstrating how this can be achieved we have shown that many other benefits can arise from the use of a logic based semantic representation in conjunction with a relational database system. We believe that such a combination offers considerable promise for the future.

REFERENCES

ADDIS 1985

ADDIS T.R. 'Designing Knowledge-based Systems', Kogan Page Ltd 1985.

AL-ZOBAIDIE 1987

AL-ZOBAIDIE A. & GRIMSON J.B. 'Expert systems and database systems: how can they serve each other ?' - Expert Systems, Vol 4, No 1, Feb 1987.

A.I. CORP 1980

ARTIFICIAL INTELLIGENCE CORP. 'INTELLECT Query System User Guide' - 500 Fifth Ave, Waltham, Mass, USA; 1980.

BATES 1984

BATES M. MOSER M. & STALLARD D. 'The IRUS Transportable Natural Language Database Interface' - Proc First International Workshop on Expert Database Systems, South Carolina, Editor H. KERSCHBERG, pp 617-630, 1984.

BERMAN 1986

BERMAN S. 'A Semantic Data Model As the Basis for an Automated Database Design Tool' - Journal of Information systems, Vol 11, No 2, pp 149-165, Pergamon Press 1986.

BOGURAEV 1983

BOGURAEV B. & SPARCK JONES K. 'How to Derive a Database Front End Using General Semantic Information' - Proc of Conference on applied natural language processing, California, pp 81-88, Feb 1983.

BONTEMPO 1983

BONTEMPO C.J. 'Feature catalog of Query by Example' - Relational Database Systems Analysis and Comparison, Edited by J.W. Schmidt & M.L. Brodie, pp 410 - 435, Verlag-Springer 1983.

BRODIE 1984

BRODIE M.L. 'Knowledge Base Management Systems Discussion Group' - Proc First International Workshop on Expert Database Systems, South Carolina, Editor H. Kerschberg, 1984.

BUNDY 1984

BUNDY A. 'Intelligent front ends' - Proc of Fourth Conference of the BCS on Expert Systems, Cambridge University Press 1984.

CERI 1986

CERI S. GOTTLOB G. & WIEDERHOLD G. 'Interfacing Relational Databases and Prolog Efficiently' - Proc First International Conference on Expert Database Systems, pp 141-153, April 1986.

CHAKRAVANTHY 1986

CHAKRAVANTHY U.S. MINKER J. & GRANT J. 'Semantic Query Optimisation: Additional Constraints and Control Strategies' - Proc First International Conference on Expert Database Systems, pp 259-269, April 1986.

CHANG 1984

CHANG C.L. & WALKER A. 'PROSQL: A Prolog Programming Interface with SQL/DS' - Proc First International Workshop on Expert Database Systems, South Carolina, Editor H. KERSCHBERG, 1984.

CHEN 1976

CHEN P. 'The entity-relationship model-towards a unified view of data' - ACM Trans on Database Systems 1 (1), pp 9-36, 1976.

CLOCKSIN & MELLISH 1981

CLOCKSIN W.F. & MELLISH C.S. 'Programming in Prolog' - Springer-Verlag; 1981.

CODD 1972

CODD E.F. 'Relational completeness of database sublanguages' - Courant Computer Science Symp, Prentice Hall 1972.

CODD 1974

CODD E.F. 'Seven steps to Rendezvous with the casual user' - IFIP conf working conf on Database Management, Ed J.W. Klimbie, North-Holland, pp 179-200, 1974.

CODD 1979

Codd E. F. 'Extending the Database Relational Model to Capture More Meaning' - ACM TODS Vol 4 No 4 Dec 1979.

CODD 1982

CODD E.F. 'Relational Database: A Practical Foundation for Productivity' - CACM, Vol 25, No 2, pp 109-117, February 1982.

CODD 1983

CODD E.F. 'Relational Database Systems Analysis and Comparison' - Editor by SCHMIDT J.W. & BRODIE M.L. Springer-Verlag; 1983.

CUFF 1984

CUFF R.N. 'HERCULES: Database query using natural language fragments' - Proc Third British National Conference on Databases, Editor J Longstaff 1984.

D'AGAPEYEFF 1983

D'AGAPEYEFF A. 'Expert Systems, Fifth Generation and UK Suppliers' - NCC Manchester; 1983.

DAMERAU 1985

DAMERAU F.J. 'Problems and some solutions in customization of natural language database front ends' - ACM Transactions on Office Information Systems, Vol 1, No 2, pp 165-184, April 1985.

DATE 1986

DATE C.J. 'An Introduction to Database Systems' - Fourth Edition Systems programming series, Vol 1, Addison-Wesley; 1986.

DE BRA 1986

DE BRA P. 'Decomposition based on Functional Dependency Set Implications' - Proc Int Conference on Database Theory, Rome, Sept 1986.

EPSTEIN 1981

EPSTEIN B. et al 'INGRES Version 6.3 Reference Manual' Feb 1981.

FEIGENBAUM 1980

FEIGENBAUM E.A. 'Knowledge Engineering for the 1980s' - Computing Science Department Stanford University, 1982.

FROST & WHITTAKER 1983

FROST R.A. & WHITTAKER S. 'A Step towards the Automatic Maintenance of the Semantic Integrity of Databases' - The Computer Journal, Vol 26, No 2, pp 124-133, 1983.

GEVARTER 1983

GEVARTER W.B. 'An overveiw of computer-based natural language processing', NASA Tech Memo 85635, Washington D.C. 1983

GINSPARG 1983

GINSPARG J. M. 'A Robust Portable Natural Language Data Base Interface' - Proc Conference Applied Natural Language Processing, Santa Monica, California, pp 25-30, Feb 1983.

GOOD 1984

GOOD M.D. et al 'Building a user-derived interface' - CACM, Vol 27, No 10, pp 1032-1043, October 1984.

GRAY 1981

GRAY M.A. 'Implementing unknown and imprecise values in databases' - Proc First British National Conference on Databases, pp 139-150, Camb 1981.

GRAY 1984

GRAY P. 'Logic, Algebra and Databases' - Ellis Horwood Series, 1984.

GROSZ 1983

GROSZ B. J. 'TEAM: A Transportable Natural-Language Interface System' - Proc of Conference on Applied Natural Language, Santa Monica California, pp 39-45, Feb 1983.

GUIDA & TASSO 1983

GUIDA G. & TASSO C. 'IR-NLI An Expert Natural Language Interface to online databases'- Proc Conference on Applied Natural Language Processing, Santa Monica, California, pp 31-38, Feb 1983.

HARRIS 1984

HARRIS L.R. 'Experience with intellect: Artificial Intelligence Technology Transfer' - The A.I. Magazine, Vol 5, Part 2, pp 43-50, 1984.

IBM 1983

IBM 'INTELLECT General Information Manual' - Program Number 5796-PWA, First Edition, Sept 1983.

IBM 1984a

IBM 'SQL/Data System Terminal User's Guide for VM/SP' - Program Number SH24-5045-1, Release 3, Second Edition, Dec 1984.

IBM 1984b

IBM Systems Journal, Vol 23, No 2, 1984.

IBM 1985

IBM 'VM/Programming in Logic Program Description and Operations Manual' - Release 1 , Modification 0 , Program Number 5785-ABH, July 1985.

JARKE & VASSILIOU 1984

JARKE M. & VASSILIOU Y. 'Coupling expert systems with database management systems' - Artificial Intelligence application for business, pp 65-85, Editor W Reitman; 1984.

JOHNSON 1984

JOHNSON R.G. 'Intergrating Data and Metadata to Enhance the User Interface', Proc Third British National Conference on Databases, Editor J Longstaff, pp 31-39, 1984.

KELLY 1977

KELLY M.J. 'Limited Vocabulary Natural Language Dialogue' - Infotech 20, Man machine studies 1977.

KING et al 1977

KING M. DELL'ORCO P. & SPADAVECCHIA V. 'Catering for the experienced and the Naive User' - Proc Workshop on Natural Language For Interaction with Databases, International Institute for Applied Systems Analysis, pp 49-68, 1977, Infotech 20 Man machine studies 1977.

KOWALSKI 1978

KOWALSKI R 'Logic for Data description' - Logic and Databases, Editor Gallairer and Minker, Plenum Press, New York, pp 77-102, 1978.

KOWALSKI 1984

KOWALSKI R 'Logic as a Database language' - Proc Third British National Conference on Databases, Editor J Longstaff, 1984.

KUMMEL 1979

KUMMEL P 'Formalization of Natural Languages' - Series in Communication with Cybernetics, Springer-Verlag, 1979.

LAKOFF 1971

LAKOFF G. 'Linguistik und naturliche Logik' - Athenaum Vlg, Frankfurt 1971.

LARSON & WALLICK 1984

LARSON J. & WALLICK J. 'An Interface for Novice and Infrequent Database Management System Users' - AFIPS Conference Proc, NCC Vol 53. pp 523-529, 1984.

LOGICA 1984

LOGICA 'RAPPORT-4 RAPIDE' - Ref R4-RPD, Database Project Group, Logica Ltd 64 Newman St London, May 1984.

MELLISH 1987

MELLISH C. 'Computers and language generation' - Paper presented at Alvey Intelligent Interface Special Interest Group Workshop, May 1987.

MOTRO 1986a

MOTRO A. 'BAROQUE: A Browser for Relational Databases' - ACM Trans Office Information Systems, Vol 4, No 2, pp 164-181, April 1986.

MOTRO 1986b

MOTRO A. 'SUPPORTING GOAL QUERIES IN RELATIONAL DATABASES' - Proc First International Conference on Expert Database Systems, pp 85-96, April 1986.

NAQVI 1986

NAQVI S. 'Negative Queries in Horn Databases' - Proc First International Conference on Expert Database Systems, South Carolina, pp 75-82, April 1986.

PARKIN 1982

PARKIN A. 'Data Analysis and System Design by Entity-Relationship Modelling' - The Computer Journal, Vol 25, No 4, pp 401-409, 1982

PRENNER & ROWE 1978

PRENNER C.J. & ROWE L.A. 'Programming Languages for Relational Database Systems' - AFIPS NCC Conference Proc, pp 849-855, 1978.

RAJINIKANTH & BOSE 1986

RAJINIKANTH M. & BOSE P.K. 'A Semantic and Logical Front-end to a Database System' - Proc ACM/SIGART International Symposium on Methodologies for Intelligent Systems, Tennessee, pp 103-111, Oct 1986.

ROUSSOPOULOS & MARK 1985

ROUSSOPOULOS N. & MARK L. 'Schema Manipulation in Self-Describing and Self-Documenting Data Models' - International Journal of Computer and Information Sciences, Vol 14 No 1, pp 1-28, 1985

RUNCIMAN & THIMBLEBY 1986

RUNCIMAN C & THIMBLEBY H. 'Equal opportunity interactive systems' - International Journal Man-machine studies 25, pp 439-451, 1986.

SCIORE & WARREN 1984

SCIORE E. & WARREN D.S. 'Towards an integrated Database-Prolog System' - Proc First International Workshop on Expert Database Systems, pp 293-305, South Carolina, Editor H. KERSCHBERG, 1984.

SENKO 1976

SENKO M.E. 'DIAM II: The binary Infological level and Its Database Language - FORAL' - ACM SIGPLAN/SIGMOD Conf Proc, Salt Lake City 1976 ACM SIGPLAN Vol 11 pp 121-140 1976.

SHAVE 1981

SHAVE M.J.R. 'Entities, functions and binary relations; steps to a conceptual schema' - Computer Journal, Vol 24, No 1, 1981.

SHIPMAN 1976

SHIPMAN D. 'The Functional Data Model and the Data Language DAPLEX' - ACM Trans on Database Systems 6, pp 140-173, 1976.

SMITH 1981

SMITH J.M. FOX S. & LANDERS T.A. 'Reference Manual for ADAPLEX' - Computer Corporation of America, 1981.

SMITH 1984

SMITH J.M. 'Expert Database Systems: A Database Perspective' - Proc First International Workshop on Expert Database Systems, South Carolina, 1984.

STONEBRAKER 1980

STONEBRAKER M.R. 'Retrospection on a Data Base System' - ACM TODS, Vol 5, No 2, June 1980.

STOTT PARKER 1984

STOTT PARKER 'Logic Programming and Databases' - Proc First International Workshop on Expert Database Systems, South Carolina, Editor H. KERSCHBERG, 1984.

TENNANT 1981

TENNANT H. 'Natural Language Processing' - PBI Petrocelli book, 1981.

TENNANT 1984

TENNANT H. 'Menu-based Natural Language Understanding' - AFIPS Conference Proc, NCC, Vol 53, pp 629-635, 1984 .

TODD 1976

TODD S.J.P. 'The Peterlee Relational Test Vehicle - a system overview' - IBM Systems Journal, Vol 4, pp 285-308, 1976.

TOU 1982

TOU F.N. et al 'Rabbit An Intelligent Database Assistant' - Proc of Conference AAAI, pp 314-318, 1982.

VASSILIOU 1983

VASSILIOU Y. et al 'How does an expert system get its data?' - Proc Nineth International Conference on Very Large Databases, pp 70-72, 1983.

VASSILIOU 1985

VASSILIOU Y. et al 'Access to Specific Declarative Knowledge by Expert Systems: The impact of Logic Programing' - Decision Support Systems 1, pp 123-141, North-Holland 1985.

VERYARD 1984

VERYARD R. 'Pragmatic data analysis' - Blackwell Scientific Publications 1984

WALLACE 1984

WALLACE M. 'Communicating with databases in natural language' - Ellis Horwood, 1984.

WALLACE & WEST 1983

WALLACE M. & WEST V. 'QPROC: a natural language database enquiry system implemented in PROLOG' - ICL Technical Journal, pp 393-406, Nov 1983.

WALTZ 1978

WALTZ D.L. 'An English Language Question Answering System for a large Relational Database' - CACM, Vol 21, No 7, pp 526-539, July 1978.

WEIZENBAUM 1966

WEIZENBAUM J. 'ELIZA - A Computer program for the study of Natural Language between Man and Machine' - CACM, Vol 9, No 1, pp 36-45, Jan 1966.

WEST 1986

WEST V. 'Natural Language Database Enquiry' - ICL Technical Journal, pp 46-63, May 1986.

WIEDERHOLD 1986

WIEDERHOLD G. et al 'Acquisition of Knowledge from Data' - Proc ACM/SIGART International Symposium on Methodologies for Intelligent Systems, Tennessee, pp 103-111, Oct 1986.

WILKINS 1986

WILKINS M.W. 'Updating Logical Databases Containig Null Values' - Proc First International Conference on Database Theory, Rome, Sept 1986.

ZANIOLO 1986

ZANIOLO C. 'Prolog: A Database Query Language for all Seasons'- Proc First International Workshop on Expert Database Systems, South Carolina, Editor H. KERSCHBERG, pp 219-232, 1984.

ZLOOF 1977

ZLOOF M. 'Query By Example: A Database Language' - IBM systems journal, pp 324-343, No 4, 1977.

APPENDIX A

AN EXAMPLE OF A RELATIONAL MODEL

STUDENT

SNUM	STITLE	SFIRSTNAME	SSURNAME	SYEAR	SAGE	PTUTOR

ATTEND

STUDENT	COURSE

COURSE

CODE	LECTURER	SESSION	LEVEL

LECTURER

LNUM	LTITLE	LFIRSTNAME	LSURNAME	LROOM	LDEPT

APPENDIX B

ALTERNATE LOGIC TRANSLATION OF AMBIGUOUS QUERIES

Initial query:

Get the names of students who attend courses attended by the student named ADAMS or students who are older than 30

Parse 1 - explicit OR

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,*,Snum,*) &
            attends(Snum,Course) &
            attends(Snum1,Course) &
            student(*,Surname1,*,*,Age1,Snum1,*) &
             (  Surname1 = 'ADAMS' |
                gt(Age1,30)    ).
```

Parse 1 - implicit OR

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,*,Snum,*) &
            attends(Snum,Course) &
            attends(Snum1,Course) &
            student(*,'ADAMS',*,*,*,Snum1,*).
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,*,Snum,*) &
            attends(Snum,Course) &
            attends(Snum1,Course) &
            student(*,*,*,*,Age1,Snum1,*) &
            gt(Age1,30).
```

Parse 2 - explicit OR

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,Age,Snum,*) &
            (  attends(Snum,Course) &
               attends(Snum1,Course) &
               student(*,'ADAMS',*,*,*,Snum1,*) )|
            gt(Age,30).
```

Parse 2 - implicit OR

```
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,Age,Snum,*) &
            attends(Snum,Course) &
            attends(Snum1,Course) &
            student(*,'ADAMS',*,*,*,Snum1,*).
query(Title,Surname,Initial) <-
            student(Title,Surname,Initial,*,Age,Snum,*) &
            gt(Age,30).
```

APPENDIX C

COMPARING QUERIES PARSED AGAINST DIFFERENT DATABASE STRUCTURES

The following queries are displayed with the possible parses in SQL to satisfy the queries for both database structures.

QUERY 1

Fetch the names of teachers teaching courses in session 1

DATABASE STRUCTURE 1

```
select title, firstname, surname
from lecturer
where
  lecturer.id in select teacher_id
                 from course
                 where
                    course.session = 1 union
select title, firstname, surname
from senior_lecturer
where
  senior_lecturer.id in select teacher_id
                 from course
                 where
                    course.session = 1 union
select title, firstname, surname
from professor
where
  professor.id in select teacher_id
                  from course
                  where
                    course.session = 1
```

DATABASE STRUCTURE 2

```
select title, firstname, surname
from person
where
  person.id in select teacher_id
            from course
            where
              course.session = 1
```

QUERY 2

Fetch the names of senior lecturers teaching courses taught at level 2

DATABASE STRUCTURE 1

```
select title, firstname, surname
from senior_lecturer
where
  lecturer.id in select teacher_id
                 from course
                 where
                   course.level = 2
```

DATABASE STRUCTURE 2

```
select title, firstname, surname
from person
where
  person.id in ( select id
                 from senior_lecturer )
and
  person.id in ( select teacher_id
                 from course
                 where
                    course.level = 2 )
```

QUERY 3

fetch the names of people who earn over £ 20000

DATABASE STRUCTURE 1

```
 select title, firstname, surname
 from lecturer
 where
   lecturer.salary > 20000
union
 select title, firstname, surname
 from senior_lecturer
 where
   senior_lecturer.salary > 20000
union
 select title, firstname, surname
 from professor
 where
   professor.salary > 20000
union
 select title, firstname, surname
 from dept_secretary
 where
   dept_secretary.salary > 20000
union
 select title, firstname, surname
 from personal_secretary
 where
   personal_secretary.salary > 20000
```

DATABASE STRUCTURE 2

```
 select title, firstname, surname
 from person
 where
   person.id in ( select id
                  from university_employee
                  where
                    salary > 20000 )
```

APPENDIX D

COMMENTED EXTRACT OF THE BSQL MODULE

```
/* bsql - if this is not a select command or it has
/*        got a "where" part or an "order by" part or
/*        a "group by" part; or the retrieval list is
/*        a variable or the retrieval list is a list of
/*        variables then use the normal sql predicate
/*

bsql(C,T,Er) <- (   ( ~ substring(C,'select',0,6) ) ;
                    ( substring(C,' where ',S,7) );
                    ( substring(C,' order by ',P,10) );
                    ( substring(C,' group by ',Q,10) );
                   var(T);
                    varl(T)  ) &
                / &
                sql(C,T,Er).

/* bsql - the previous predicate has failed so the
/*        command is a select command without a where
/*        part and the retrieval list is partly assigned

bsql(C,T,Er) <- stconc('describe ',C,DesC) &
                sql(DesC,TD,Errd) &
                ge(Errd,0) &
                create(TD,T,New,' where ') &
                stconc(C,New,NewC) &
                / &
                sql(NewC,T,Er).
bsql(C,T,Er) <- sql(C,T,Er).

/* create - given the describe list and the retrieval
/*          list we can create a where part to constrain
/*          the DBMS access

create([*![*![*!TT]]],[H!T],R,W) <- var(H) &
                                    create(TT,T,R,W) & / .
create([HT![*![*!TT]]],[H!T],R,W) <- stconc(W,HT,S1) &
                                     stconc(S1,' = ',S2) &
                                     stt_to_att(SH,H) &
                                     stconc(S2,SH,R1) & / &
                                     create(TT,T,R2,' and ')
                                        &
                                     stconc(R1,R2,R) & /.
create([],[],'',Any).
```

```
/* stt_to_att this predicate converts a string to an atom
/*            if the atom is already a string then it places /*
it in quotes

stt_to_att(SH,H) <- stringp(H) & / &
                    stconc('''',H,S1) &
                    stconc(S1,'''',SH) & /. stt_to_att(SH,H) <-
st_to_at(SH,H).

/* varl succeeds if the list is a list of variables

varl([H!T]) <- var(H) & varl(T) & / .
varl([]).
```

APPENDIX E

SUMMARY OF TEST RESULTS

The following table lists the queries which were analysed.

TEST NUM	SQL QUERY
1	sql('select * from student where number = 6',[A,B,C,D,E],Er).
2	sql('select * from student',[A,6,C,D,E],Er).
3	sql('select * from student',[A,B,C,D,E],Er) & B = 6.
4	sql('select name,number from student where number=6',[A,B],Er).
5	sql('select name,number from student where name=''FABER''',[A,B],Er).
6	sql('select name,number from student',[A,B],Er) & A = 'FABER'.
7	sql('select * from student where number=6',[A,B,*,*,*],Er).
8	sql('select * from student',[A,6,*,*,*],Er).
9	sql('describe select * from student',A,Er).
10	sql('describe select name,number from student where number=6',A,Er).
11	sql('select name,number from student where number = 6',A,Er).
12	sql('select * from student where age = 54',[A,B,C,D,E],Er).
13	sql('describe select name,number from student where number=6',Des,Err) & sql('select * from student where number = 6',[A,B,C,D,E],Er).

RESULTS SUMMARY

7 tuple relation mean CPU time in milliseconds for a 1000 iterative loop. Test 0 is a control test of the iterative loop.

GROUP	TEST	NORMAL	BOUND	DIFFN	%DIFF
GROUP 1	9	1428	1503	-75	-5.25
	10	1396	1462	-66	-4.73
GROUP 2	1	2618	2848	-230	-8.79
	4	2506	2744	-238	-9.09
	5	2511	2746	-235	-9.36
	7	2644	2846	-202	-7.64
	11	2511	2730	-219	-8.72
	12	2663	2850	-187	-7.02
GROUP 3	2	5328	4465	863	16.20
	3	5634	6034	-400	-7.10
	6	5248	5632	-384	-7.32
	8	5369	4479	890	16.58
ROGUE	0	20	20	0	
	13	3987	4270	-283	

The following test lists a second set of test queries

TEST NUM	SQL QUERY
*1	sql('select * from qpcat.catsingles where scatno = 3598', [A,B,C,D,E,F,G,H,I],Er).
*2	sql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,3598,I],Er).
*3	sql('describe select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],Er).
*4	sql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],Er) & H = 3598.

RESULTS SUMMARY

3000 tuple relation mean CPU time in milliseconds for a 10 iterative loop. Test 0 is a control test of the iterative loop.

GROUP	TEST	NORMAL	BOUND	DIFFN	%DIFF
	*0	0	0	0	
2	*1	28	30	-2	-7.14
3	*2	15185	50	15135	99.67
1	*3	14	15	-1	-7.14
3	*4	16772	17041	-269	-1.60

APPENDIX F

EXAMPLES OF REPRESENTATION LANGUAGE STATEMENTS

USER VIEW INFORMATION

```
entity ENTITY.

ENTITY1 direct_subset_of ENTITY2.

connect( CONNECTION, ENTITY1, ENTITY2, N, M, INVERSE_CONNECTION).

intra_entity( CONNECTION, ATTRIBUTE, COMPARATOR).

composite( COMPOSITE_ATTRIBUTE ,ATTRIBUTE_LIST).
```

DATABASE MODEL INFORMATION

```
relation( RELATION, KEY_FIELDS, ATTRIBUTE_LIST).

image_relation( RELATION, KEY_FIELDS, IMAGE_ATTRIBUTE_LIST).

attribute( ATTRIBUTE, ATTRIBUTE_TYPE, TYPE_DESCRIPTION).
```

TRANSLATION INFORMATION

```
database_term( USER_PHRASE, DATABASE_TERM).

plural( SINGULAR, PLURAL).

synonym( USER_TERM, DATABASE_TERM).
```

QUERY FORMULATION

```
bsql( SQL_SELECT_STATEMENT,
      ATTRIBUTE_RETRIEVAL_LIST,
      ERROR_RETURN_CODE).
```

APPENDIX G

ATTRIBUTE STATEMENT DESCRIPTION TABLE

ATTRIBUTE_TYPE	DESCRIPTION	INTERPRETATION
integer	N	N digit integer
real	[N,M]	real number of N digits M digits after decimal point
fchar	N	character string of fixed length N
vchar	N	character string of variable length N
composite	RULENAME	composite attribute values acceptable combinations defined by predicate RULENAME(*)
one_of	[LIST]	LIST of all possible values attribute can take
reference	RELATION	attribute is a reference to a tuple in the relation RELATION. It is therefore of the form of the key attribute(s) of the RELATION

APPENDIX H

COMMENTED EXAMPLE OF AN INTERACTIVE SESSION WITH THE SYSTEM

To show the flexibility and capability of the system we now consider a non trivial query which is traced from the initial input of the user query through to the user receiving a reply.

LIST THE NAMES OF ALL STUDENTS WHO ARE FEMALE AND ARE LECTURED BY PROF SMITH

Given this initial input the system parses it, then queries the unknown terms. If two unknown terms are separated by only a space then the system initially assumes them to be perceived as a single item. The system fails to recognise the words LIST, PROF and SMITH. The system responds with:

Do you believe that the term LIST is stored in the database y/n

To which the user would respond: N

The system then replies:

I do not understand the term LIST please supply an alternative or type Q to quit and respecify the entire query

To which the user replies: FETCH

The system recognises this term and so proceeds to ask:

Do you believe that the term PROF SMITH is stored in the database y/n

To which the user replies: Y

The system then proceeds to parse the query to confirm its understanding it asks:

The objects you wish retrieved are: Person Name

To which the user replies: Y

The system then proceeds to state its assumptions:

This retrieval is subject to:

person is a student and student has sex ='FEMALE' and
student lectured by teacher and teacher is a person and
person's name is according to a rule 'PROF SMITH'

To which the user replies: Y

The system then codes the query and then informs the user:

There is 1 tuple satisfying the request hit return to view or q to quit.

If the user types return he is presented with the following list

title	MRS
firstname	DORIS
surname	SMITH

INTERNAL LOGIC REPRESENTATION AND ALTERNATIVE SQL TRANSLATION

From the previous interactive session the system will have generated the following internal representation which it could then execute.

Initial Query,

LIST THE NAMES ALL OF STUDENTS WHO ARE FEMALE AND ARE LECTURED BY PROF SMITH

Logic coding of query generated by system

```
pred_query([Title0,Firstname0,Surname0])  <-
      person(Num0,Age0,Title0,Firstname0,Surname0,
                Street0,Town0,County0,Country0,Sex0 )  &
      student(Num0,Home_street0,Home_town0,Home_county0,
                Home_country0,Nationality0,Sponsor0,
                Attendance_type0,Year_of_study0 )  &
      eq(Sex0 ,'FEMALE' ) &
      lectured_by(Id_num0, Id_num1 ) &
      teacher(Id_num1,Room1 )  &
      person(Id_num1,Age1,Title1,Firstname1,Surname1,
                Street1,Town1,County1,Country1,Sex1 )  &
      namerule(Title1,Firstname1,Surname1,'PROF SMITH').
```

Possible Optimal Logic coding of query (hand generated)

```
pred_query([Title0,Firstname0,Surname0])  <-
      person(Num0,*,Title0,Firstname0,Surname0,*,*,*,*,'FEMALE')  &
      lectured_by(Id_num0, Id_num1 ) &
      person(Id_num1,*,'PROF',*,'SMITH',*,*,*,*,* ).
```

SQL coding of same query (hand generated)

```
select person.title, person.firstname, person.surname
from person
where
 person.num in ( select num
                 from student)
and
 person.sex ='FEMALE'
and
 person.num in (select attend.student
                from attend
                where
                 course in (select code
                            from course
                            where
                             lecturer = (select num
                                         from person
                                         where
                                            title='PROF'
                                            and
                                            surname='SMITH' )))
```

APPENDIX I

SOURCE CODE LISTING OF ENTIRE INTERFACE SYSTEM

MODULE LIST

Alphabetical listing of all program modules

PROBOOT	EXEC
STARTUP	EXEC
AGGREGAT	PROLOG
ALTER	PROLOG
ANALYSE	PROLOG
APOST	PROLOG
APPEND	PROLOG
ATTFIND	PROLOG
BSQL	PROLOG
CHECK	PROLOG
COMPOSE	PROLOG
CONDITIO	PROLOG
CONNECTO	PROLOG
CONUSER	PROLOG
CSQL	PROLOG
CUSER	PROLOG
DATA	PROLOG
DBTERMS	PROLOG
DISPLAY	PROLOG
DOMAIN	PROLOG
DOUBLE_D	PROLOG
EQUIVAL	PROLOG
EXPLAIN	PROLOG
FC	PROLOG
FILL_REM	PROLOG
FILLER	PROLOG
FIND	PROLOG
FIND_DEF	PROLOG
GETFILE	PROLOG
GETSCREE	PROLOG
ID	PROLOG
INTRO	PROLOG
ISA	PROLOG
LOGIC	PROLOG
MEMBER	PROLOG
NEWUSER	PROLOG
OUT	PROLOG
OUTQ	PROLOG
PLURAL	PROLOG
PRINTL	PROLOG
PROCESS	PROLOG

QCON	PROLOG
QUOTED	PROLOG
RATI	PROLOG
READIN	PROLOG
REVERSE	PROLOG
RL	PROLOG
ROUL	PROLOG
RUN	PROLOG
SLEAD	PROLOG
STR_LIST	PROLOG
SYNONYM	PROLOG
TM	PROLOG
TOP	PROLOG
TRANSL	PROLOG
UC	PROLOG
UNIT_UN	PROLOG
VERIFY	PROLOG

APPENDIX I continued

PREDICATE CROSS REFERENCE LIST

PREDICATE NAME
[MODULE NAME WHERE PREDICATE DEFINED]
BRIEF PREDICATE DESCRIPTION

a
[qcon]
test predicate used to deliver integers in the range 1 to 8

a_condition
[fc]
defines the types of possible conditional units

act_on
[conuser]
predicate takes appropriate action depending on user response to question about nature of an unknown term in a user query

actual_relation
[alter]
relational attributes as actually stored, compare with image_relation which stroes composite attributes

add_entity
[cuser]
adds entity name to list of attribute to form retrieval list

addressrule
[alter]
model specific composite attribute comparison rule

add_subset
[rl]
add a subset clause to the conditional clause list of the query

add_term_stored
[conuser]
record that the user believes a term is stored

add_to_term_database
[conuser]
terms composed of several words must be stored so that they may be identified in the future

a_function
[process]
specifies all recognised system functions

aggregates
[aggregat]
lists for model aggregates and their type (0 to 5)

analyse
[analyse]
analyse query input to form system recognised objects

apostro
[apost]
eliminates single apostrophe from a word and notes the possibile existence of a possession verb

append
[append]
appends two lists to form a third

ask_about
[cuser]
asks the user whether an assumption is correct, this is assuming it has not already been asked, the user response is then stored

attend
[data]
model relation predicate

attended_by
[alter]
model specific connection specification

att_inf
[domain]
displays further information when requested on an attribute during the querying of the domain information

attributes
[alter]
attribute definition statement

attributes_of_interest
[attfind]
given a relation name finds all attributes of it or its supersets

atributes_of_interest
[cuser]
given a relation name finds all attributes of it or its supersets and records all entity names in a list

belongs_to
[data]
model relation predicate

break_printlist
[explain]
prints out the query condition list with suitable line breaks to make the assumptions more readable

bsql
[bsql]
binds instantiated varibles into the call to the SQL/DS database

capital
[outq]
captialise first element of a list of characters

caseshift
[str_list]
turn a string from lower case to upper case

ccat
[out]
csql test predicate

change_eqn
[qcon]
tests current status of variable then correctly changes it, if possible, to eq N

change_gen
[qcon]
tests current status of variable then amends it, if possible, to be constrained by ge N

change_gtn
[qcon]
tests current status of variable then amends it it if possible to be constrained by gt N

change_len
[qcon]
tests current status of variable then amends it if possible to be constrained by le N

change_ltn
[qcon]
tests current status of variable then amends it if possible to be constrained by lt N

check
[double_d]
checks integrity of loaded Prolog code

check_addax
[qcon]
check to see if a variable contstrained by >= and <= conicide so making the only relevant constraint =

check_asked_condition
[check]
checks to see if the user has already been asked a question if he has not then new question is recorded as having been asked - can also be used to clear asked storage area

check_for_double_definition
[double_d]
checks to see if the same predicate, head and body, has been defined more than once

check_for_split_clause_definition
[find_def]
checks to see if a predicate head has been defined in more than one module

check_user
[cuser]
checks with user to ensure that the deductions made by the system about the query are correct

combine
[fc]
combines conditional objects to form more complete conditional units in the conditional clause list

comp
[csql]
forms compound conditional clauses by appending "and" to the front of any following conditional clauses

compose
[compose]
composes the recognised query terms into a meaning internal representation

composite
[alter]
composite attribute definition statement

con
[attfind]
locates all connections for a given entity

cond
[csql]
matches condition names to prolog conditional predicates

conditionals
[conditio]
states types of conditions

condition_output
[out]
output a conditional clause in a more easily readable form for confirmation checking by the user

con_inf
[domain]
displays information on a connection when requested during querying of the domain information

connect
[alter]
model connection definition statement

connection_of_interest
[attfind]
identifies for a given entity all connections it has with other entities or it inherits from any superset

connectors
[connecto]
states the three types of connectors "and" "or" "not"

constraint
[qcon]
adds a constraint on to the predicate evaluation stack

constraint_con
[csql]
forms conditional clause of a SQL/DS query if the variable has been instantiated to a constraint condition.

consult_user_for_unknown
[conuser]
dialogue with user to detremine which of the unknown terms the user believes to be stored in the database

copy_till_dash
[str_list]
copy a string enclosed in quotes until closing quote

count
[run]
count number of tuples retrieved by the logic query predicate, up to a maximum of 10

count1
[csql]
csql test predicate

course
[data]
model relation predicate

coutput
[qcon]
writes out a constraint clause

create
[csql,bsql]
creates the conditional clauses of a SQL/DS call

crite
[logic]
writes a condition as a prolog predicate

csql
[csql]
binds all conditional statements into a SQL/DS call

ct
[csql]
predicate csql test call

ct2
[csql]
predicate csql test call

ct3
[csql]
predicate csql test call

cw
[domain]
writes out a connection

cwrt
[qcon]
writes out a prolog conditional predicate as a conditional operator

database_term
[alter]
model phrase definition statement

db_terms
[dbterms]
joins words in query word list to form recognised object terms

department
[data]
model relation predicate

display
[display]
displays current query and assumptions made

dimension
[alter]
attribute dimension definition statement

domain_attributes
[domain]
identifies all attributes for an entity including those inherited from any superset entities

domain_connections
[domain]
identifies all connections for an entity including those inherited from any superset entities

domain_inf
[domain]
displays information on an entity when requested during querying of the domain information

domain_ssets
[domain]
locates all subsets for a given domain entity and the subsets of its subset entities

entity
[alter]
entity definition statement

eqn
[qcon]
equals constraint for constraing a variable to be equal to a number N

equi
[equival]
defines equivalance of two different operators anded together

equivalance
[equival]
defines equivalance of any two operators anded together

explain
[explain]
when a predicate query fails to retrieve any tuples an explanation of the conditions on which the query was based is output

extfind
[find_def]
finds names of all prolog files on system

faculty
[data]
model relation predicate

fee
[data]
model relation predicate

fget_query
[getfile]
using a set of query got from a test file run the system

filler
[filler]
defines all words considered to be filler words

find
[find_def]
finds module(s) name where predicate is defined

find_aggregate
[id]
attempts to find aggregate clauses in a query

find_aggregates
[id]
searches entire query sentence to find aggregate clauses

findall
[find_def]
specifies all predicates currently loaded

find_combination
[id]
given a list of conditions attempts to match them to form more succinct queries

find_combinations
[id]
searches an entire query sentence to find all possible combinations or combinations of combinations

find_condition
[id]
given a list of term types attempts to match them to identify possible conditional units

find_conditions
[id]
searches list to find all conditions

find_entity
[uc]
searches for an entity as an atom or the first term of an object list structure

find_object
[rl]
finds an entity attribute pair within a list

find_others
[find_def]
finds name of other module(s) if predicate defined in more than one module

find_others_similar
[find_def]
find names of other modules where predicates with a name containing a given string are defined

findout
[find]
finds for a given predicate body all predicates which it calls

find_similar
[find_def]
find names of modules for predicates with a given string contained in their name

find_subset
[rati]
given query clause list locate the subset clauses within it

formalise_condition
[fc]
form conditional objects into conditional clauses

formalise_conditions
[fc]
for the entire conditional list form conditional objects into conditional clauses

form_condition
[csql]
forms conditional part of SQL\DS query

form_condition
[uc]
form the predicate query conditions on which the retrieval is based

form_object
[rl]
combine an entity and attribute to form an object

function_check
[domain]
checks whether a user has input a system function 'domain', 'load', 'new', 'save'

g
[getscree]
initiates running the system to parse the user query once parsed it then asks the user to confirm the execution of the query

gen
[qcon]
greater than or equal to constraint for constraing a variable to a conditional value

get_attribute
[fc]
get an attribute appropriate for a conditional unit, "any" if any attribute appropriate

get_comparator
[uc]
identify comparison operator for a conditional clause

get_condition
[fc]
get conditions embedded in a list structure

get_context
[uc]
identify entity in context which matches current attribute

get_entity
[rl]
get an entity to match with an attribute

get_inter_entity
[uc]
get new in context entity after an inter entity connection condition

get_objects
[uc]
group remainig entity objects to form retrieval list

get_query
[getscree]
reads in a user query from the terminal then initiates running the system to understand the query

gf
[getfile]
test run of system reading queries from file "simple data a"

gtn
[qcon]
greater than constraint for constraing a variable to a conditional value

identify_clauses
[id]
identifies the constraint clauses of a query first identifying the conditional units then attempting to combine them

identify_quoted_terms
[quoted]
given list of words removes quotation marks from quoted terms

ifrule
[out]
outputs when writing out conditional clauses that a condition is subject to a rule

image_relation
[alter]
relation definition statement includes imagined composite attributes

increm
[outq]
returns the increment of the input variable

ins
[data]
insert values into the SQL/DS database

intra_entity
[conditio]
states intra-entity conditions for a given model

intro
[intro]
Initiates system running display a welcome screen

is
[isa]
predicate defined to specify the type of a word in a query sentence

lastline
[readin]
checks to see if line read from the terminal is the last line or a continuation line

lastlinef
[readin]
checks to see if line read from a file is the last line or a continuation line

lectured_by
[alter]
model specific connection specification

lecturer
[data]
model relation predicate

lecturer_to
[alter]
model specific connection specification

len
[qcon]
less than or equal to constraint for constraing a variable to a conditional value

level
[data]
model relation predicate

list_attributes
[attfind]
prints out attribute list three to a row

list_domains
[attfind]
prints out all entity names

list_eqn
[csql]
makes elements of one list equal constrained to elements of another list

listst
[data]
convert list elements into string separated by ","

loaduser
[newuser]
loads the assumptions and data stored in a user profile

loc
[dbterms]
matches a list to the start of another list delivering the remainder of the list

locate
[dbterms]
locates a list of elements in another list

ltn
[qcon]
less than constraint for constraing a variable to a conditional value

maintain_query
[check]
forms predicate to maintain query has been asked store

make_retrieval_list
[uc]
form retrieval list from list of remaining objects

make_variable
[outq]
turns an atom or composite attribute into a variable or list of variables by capitalising the first character

match
[id]
matches types of conditions with terms in a query sentence

match
[tm]
match acceptability of a value to the defined attribute format

match_isa
[uc]
match a condition with the in context entity or one of its super/sub sets

members
[member]
finds whether an element is a member of a list

namerule
[alter]
model specific composite attribute comparison

nationality_classify
[data]
model relation predicate

nct2
[csql]
predicate csql test call

nct
[csql]
predicate csql test call

ncat
[csql]
predicate csql test call

negate_inter_ent
[outq]
write out a not clause for the query predicate tail

new_append
[attfind]
appends the elements of one list to another eliminating any duplicate elements

new_con
[qcon]
sets up a new conditional constraint number for a variable

new_context
[uc]
defines possible new context entity

new_query
[newuser]
clears any previously asserted queries

new_user
[newuser]
clears user profile area for new user removing all assumptions made during any prior session

no_composite
[logic]
ensures an attribute list has no members which are composite attributes

not_attend
[alter]
model specific connection specification

not_attended_by
[alter]
model specific connection specification

num_writes
[run]
write out number of possible tuples to be retrieved

nw_append
[cuser]
appends pairs of list objects as long as every second element of the pair is unique

obtain
[apost]
obtains conditional type of inter-entity condition

obtain_context
[rl]
find entity which is currently in context

obtain_objects
[rl]
search a list to find all entity attribute objects

obtain_retrie_list
[rl]
form a list of objects which will constitute the retrieval list

obtain_retrieval_list
[rl]
initiate process of attempting to find required retrieval list allowing backtracking from any wrong assumption

o_clause
[find_def]
output the clause's module name and calling predicate list for each predicate

o_pred
[find_def]
outputs predicate list as long as it have not already been printed

out
[newuser]
opens a new file or oldfile if it already exists to write out a user profile

out_att_list
[logic]
outputs the attribute list as a list of variables to form the query predicate

out_cond
[uc]
write out condition to the condition list

out_cond
[logic]
outputs a condition as a prolog condition as part of the logic specification of a query predicate

out_dim
[run]
write out for an attribute its dimension if known

output_atts
[domain]
output a list of attributes for an entity and initiates further querying of attributes

output_condition
[logic]
outputs a conditional clause as part of the logic specification of a query

output_connector
[logic]
outputs a connector as part of the logic specification of a query predicate

output_cons
[domain]
output a list of connections for an entity and initiates further querying of these connections

output_inter_entity
[logic]
outputs an inter entity relation as part of the logic specification of a query predicate

output_pred
[find_def]
output a list of predicates with the modules where they are declared and a list of predicates which call them

output_profile
[newuser]
writes out the assumptions made during a session to a user profile file

output_relation
[logic]
outputs a relation as part of the logic specification of a query predicate

output_retrieval
[run]
write out a retrieved tuple

output_ssets
[domain]
output a list of subsets for an entity and initiates further querying of these subsets if required

output_subset
[logic]
outputs a subset condition as part of the logic representation of the query

out_ret
[run]
write an attribute value for a retrieved tuple

out_ret_list
[outq]
writes out the logic query predicate list of attributes to be retrieved

out_query
[outq]
writes out the logic predicate representation of the original query

paid
[alter]
model specific connection specification

paid_to
[alter]
model specific connection specification

person
[data]
model relation predicate

personal_tutee_of
[alter]
model specific connection specification

personal_tutor_to
[alter]
model specific connection specification

plural
[alter]
vocabulary plural definition statement

post_grad
[data]
model relation predicate

primary_entity_list
[domain]
identifies all entities which are primary i.e. have no supersets

printlist
[printl]
print a list of elements as a string separated by spaces

printlistf
[printl]
print to a file, a list of elements as a string separated by spaces

process_query
[process]
controls the parsing of the initial user query

professor
[data]
model relation predicate

professor_sec
[data]
model relation predicate

prt_oblst
[domain]
prints outs a list of objects as string separated by spaces

putwrittail
[outq]
test predicate used to debug predicate query tail output

q
[attfind]
Queries for all known entities their attributes, connections

qst_to_qli
[str_list]
turns a string of words separated by spaces into a list

qt_to_ql
[str_list]
locates query terms to form a query list

query
[attfind]
Queries an entity specifying all known facts about the domain of interest i.e. atrributes, connections, subsets

quoted
[quoted]
returns string within quotation marks

rationalise
[rati]
removes unwanted/repetative subset conditions

readline
[readin]
reads a line from the terminal checks to see if it is a continuation line

readlinef
[readin]
reads line from a file and test to see if it is the last line

recall_query
[check]
recalls all queries that have been stored as "been asked of the user"

reduce
[verify]
reduces a retrieval list by eliminating relation clauses if an object from that relation is being retrieved on its own

relation
[alter]
relation definition statement

remove_default
[outq]
remove default context entity names from required attributes list

remove_filler
[fill_rem]
eliminates all words with type filler from a list

remove_previous_queries
[outq]
removes the instatiation of any previous queries

re_order_not
[process]
correct specification of NOT next to actual condition and away from any assumption clauses

retrieve_output
[cuser]
forms output list of entity attributes to retrieved

reverse
[reverse]
reverses the order of elements in a list

reverse_get_inter_entity
[uc]
get inter entity connection for an inversely defined connection to the one specified in the connect statement

rites
[domain]
writes out connection name and degree of connection based on description of connection in connect statement

rules
[find]
returns all predicates called by a given predicate

run_query
[run]
load query predicate and execute it displaying the results if required

salary
[data]
model relation predicate

saveuser
[newuser]
saves the assumptions made during the current session in a user profile file

same_entity
[uc]
a new context entity is the same as the current one

same_key
[logic]
tests whether two entities have the same key

sc
[qcon]
search and write out all conditional constraints

secretary
[data]
model relation predicate

separate
[str_list]
specifies possible word separtors

session
[data]
model relation predicate

setup
[top]
loads all system modules

shift
[str_list]
converts lower case to upper case unless word is quoted

single_strop
[apost]
If a word only has a single apostrophe then the word before the apostrophe is returned

sing_plural
[domain]
Converts word from plural to single if the plural is known

specific_conditionals
[fc]
ascertains whether a conditional unit refers to a specific attribute

space_separate
[str_list]
seperate all word separtion characters by spaces

split_object
[out]
splits an object pair back into its entity attribute parts

sponsor
[data]
model relation predicate

stprst
[run]
writes out an atom or print a string

strip_leading_terms
[slead]
strip any leading terms from the list of conditional clauses which are not conditions

strip_object_default
[rl]
remove default context entities from the retrieval list

strip_spaces
[str_list]
removes spaces from start and finish of a string

stt_to_att
[csql,bsql]
converts an atom to a string if it is already enclosed in quotes then the resulting string is also placed in quotes

student
[data]
model relation predicate

sub_cond
[logic]
outputs a sub condition as part of the logic specification of a query predicate

sub_condition
[out]
writes out a sub condition, used to make reading conditional clauses easier when a user is confirming a parse

sub_sep
[str_list]
substitute a word separation character with a string made up of itself with a space either side

subset
[alter]
entity direct subset definition statement

subsets
[alter]
entity direct and indirect subset definition statement

supervise
[data]
model relation predicate

supervised_by
[alter]
model specific connection specification

supervisor_to
[alter]
model specific connection specification

swap_plurals
[plural]
for list of words swap plurals for singular

swap_synonyms
[synonym]
swaps any word which has a more usual synonym

synonym
[synonym]
defines all acceptable synonyms

synonyms
[synonym]
defines a synonym as a synonym of a synonym
taught_by
[alter]
teacher
[data]
model relation predicate
teacher_for
[alter]
model specific connection specification
test
[qcon]
test to see whether a constraint actually further constrains a variable
test_db_term
[conuser]
tests whether a term is stored as a database term
translate
[transl]
translates a list of words into there recognised word type
type_of_aggregate
[id]
defines sequence of terms which constitute an aggregate clause
type_of_combination
[id]
defines sequence of terms and conditional clause which constitute a combined conditional clause
type_of_condition
[id]
defines a sequence of terms which constitute a conditional clause
type_of_unknown
[unit_un]
defines the two types of unknown values to be unknown or believed stored
under_grad
[data]
model relation predicate
unit_co
[uc]
unites conditionals with their objects and rules maintaining the in context entity
unite_conditions
[uc]
unites conditionals with their objects and rules
unite_unknown
[unit_un]
forms all types of consecutive unknown terms into a single unknown term

university_employee
[data]
model relation predicate

uppercase_data
[quoted]
convert unknown data items to uppercase unless they are quoted

ushift
[str_list]
converts a list of characters from lower case to upper case

varl
[csql,bsql]
determines whether a list is a list of uninstantiated variables

verify_retrieval
[verify]
verifies that in a retrieval list a relation and selected objects from that relation are not being retrieved together

wc
[domain]
writes out a connection

wct2
[csql]
predicate csql test call

write_connect
[domain]
writes out the information about an entity connection in a pretty form

write_head
[outq]
writes out the head of the logic query predicate

write_line
[domain]
writes out a line for a connection depending on whether it is a 1 to 1 or 1 to many connection

writes_att
[domain]
writes out an attribute definition based on the attribute statement

write_tail
[outq]
write out the query predicate tail

writtail
[outq]
write out the clauses and clause connections of the query predicate tail

wrt_tail
[outq]
write out a clause of the query predicate tail

```
/* start up vm prolog with prolog file top  */
EXEC PROBOOT
EXEC VMPROLOG DROP WS MIXED NODISPLAY
```

```
&BEGSTACK
reconsult(top).
intro.
&END
```

```
&BEGSTACK
reconsult(top).
intro.
&END
```

```
aggregates(average,0).
aggregates(count,3,any).
aggregates(least,2,any).
aggregates(maximum,1,any).
aggregates(minimum,2,any).
aggregates(most,1,any).
aggregates(sum,4,any).
aggregates(total,3,any).
aggregates(oldest,1,age).
aggregates(youngest,2,age).
```

```
not_attend(I,C) <- ¬ var(I) & person(I,*,*,*,*,*,*,*,*,*,'UNIV_EMP') & /.
not_attend(I,C) <- person(I,*,*,*,*,*,*,*,*,*,*) &
                   course(C,*,*,*) &
                   ¬ attend(I,C) .
attended_by(Code, Id ) <- attend(Id,Code).
not_attended_by(Code, Id ) <- not_attend(Id,Code).

taught_by(Course,Teacher) <- course(Course,*,*,Teacher).
teacher_for(Teacher,Course) <- course(Course,*,*,Teacher).

personal_tutor_to(Teacher,Under_grad) <- under_grad(Under_grad,Teacher).
personal_tutee_of(Under_grad,Teacher) <- under_grad(Under_grad,Teacher).

supervised_by(Post_grad,Teacher) <- supervise(Teacher,Post_grad).
supervisor_to(Teacher, Post_grad) <- supervise(Teacher,Post_grad).

lecturer_to(Teacher, Student) <- attend(Student,Course) &
                                 taught_by(Course,Teacher).
lectured_by(Student,Teacher) <- lecturer_to(Teacher,Student) .

addressrule(St,To,Co,Cr,To).
addressrule(St,To,Co,Cr,Co).
addressrule(St,To,Co,Cr,Cr).
addressrule(St,To,Co,Cr,S) <- stconc(St,' ',S1) &
                              stconc(To,' ',S2) &
                              stconc(Co,' ',S3) &
                              stconc(S1,S2,S4) &
                              stconc(S4,S3,S5) &
                              stconc(S5,Cr,S).
addressrule(St,To,Co,Cr,S) <- stconc(To,' ',S2) &
                              stconc(Co,' ',S3) &
                              stconc(S2,S3,S5) &
                              stconc(S5,Cr,S).
addressrule(St,To,Co,Cr,S) <- stconc(To,' ',S2) &
                              stconc(S2,Co,S) .
addressrule(St,To,Co,Cr,S) <- stconc(To,' ',S2) &
                              stconc(S2,Cr,S) .
addressrule(St,To,Co,Cr,S) <- stconc(Co,' ',S2) &
                              stconc(S2,Cr,S) .


namerule(St,Sf,Ss,S) <- stconc(St,' ',S1) &
                         stconc(Sf,' ',S2) &
                         stconc(S1,S2,S3) &
                         stconc(S3,Ss,S).
namerule(St,Sf,Ss,S) <- stconc(St,' ',S1) &
                         stconc(S1,Ss,S).
namerule(St,Sf,Ss,S) <- stconc(Sf,' ',S1) &
                         stconc(S1,Ss,S).
namerule(St,Sf,S,S).

attributes(level_id,one_of,["ONE","TWO"]).
attributes(session_id,one_of,["FIRST","SECOND","THIRD"]).
relation(professor,[id_num],[id_num]).
relation(lecturer,[id_num],[id_num]).
relation(senior_lecturer,[id_num],[id_num]).
relation(post_grad,[id_num],[id_num]).
relation(session,[session],[session_id]).
relation(level,[level],[level_id]).
relation(salary,[salary],[salary]).
entity(session).
entity(level).
```

```
entity(salary).

/* SYSTEM SET-UP FOR LIST NOTATION
<- pragma(list,1).


/* A SPECIFICATION OF THE CODED RELATIONAL MODEL FOR THE
/* COURSE-LECTURER-STUDENT DOMAIN
/*

/* ENTITIES OR OBJECTS IN THE DOMAIN
/*
entity(course).
entity(post_grad).
entity(under_grad).
entity(departmental_sec).
entity(professor_sec).
entity(fee).
entity(professor).
entity(lecturer).
entity(senior_lecturer).
entity(sponsor).
entity(department).
entity(faculty).


/* THE RELATIONS REPRESENTING THE ENTITIES WITH THIER ATTRIBUTES
/* (Attribute which are entity references have been high-lighted)


image_relation(course,[code],[code,session,level,teacher]).

image_relation(person,[id_num],[person_type,id_num,age,sex,
                                                name,address]).
actual_relation(person,[id_num],[id_num,age,title,firstname,surname,
                          street,town,county,country,sex,person_type]).
composite(name,[title,firstname,surname]).
composite(address,[street,town,county,country]).

image_relation(student,[id_num],[id_num,home_address,nationality,sponsor,
                    attendance_type,year_of_study,student_type]).
actual_relation(student,[id_num],[id_num,home_street,home_town,
home_county,
                   home_country,nationality,sponsor,
                   attendance_type,year_of_study,student_type]).
composite(home_address,[home_street,home_town,home_county,home_country]).

image_relation(under_grad,[id_num],[id_num,personal_tutor]).

image_relation(university_employee,[id_num],
                          [id_num,salary,department_id,
                          ni,tax_code,university_employee_type]).

image_relation(secretary,[id_num],[id_num,secretary_type]).

image_relation(teacher,[id_num],[id_num,room,teacher_type]).

image_relation(professor_sec,[id_num],[id_num,professor]).

image_relation(fee,[attendance_type,nationality_classify,student_type],
            [attendance_type,nationality_classify,student_type,value]).

image_relation(sponsor,[sponsor_name],[sponsor_name,address]).
```

```
image_relation(department,[department_code],[department_code,
                                        department_name,faculty]).

image_relation(faculty,[faculty_name],[faculty_name]).


/* EXPLICIT DEFINITION OF ENTITY RELATIONSHIPS REQUIRED FOR MANY TO MANY
/* RELATIONSHIPS
/*
image_relation(attend,[student_id,course_id],[student_id,course_id]).

image_relation(belongs_to,[student_id],[student_id,department_id]).

image_relation(supervise,[teacher],[teacher,post_grad]).


/* ROGUE RELATION ????
/*
image_relation(nationality_classify,[nationality],
                                [nationality,classification]).

relation(X,Y,Z) <- image_relation(X,Y,Z).
relation(X,Y,Z) <- actual_relation(X,Y,Z).



/* ADDITIONAL INFORMATION FOR USER CONCEPTUALISED MODEL
/*

/* SUBSET DEFINITION
/* subset(person,student)  defines  person   has a subset student
/*                     or inversely  student  is a person
/*
subset(person,university_employee).
subset(person,student).
subset(university_employee,teacher).
subset(university_employee,secretary).
subset(student,post_grad).
subset(student,under_grad).
subset(secretary,departmental_sec).
subset(secretary,professor_sec).
subset(teacher,professor).
subset(teacher,senior_lecturer).
subset(teacher,lecturer).

subsets(X,Y) <- subset(X,Y).
subsets(X,Y) <- subset(X,Z) & subsets(Z,Y).

/* A SUPERSET IS ALSO AN ENTITY SO WE NEED TO EXPAND THE ENTITY DEFINTION
/*
entity(person).
entity(student).
entity(university_employee).
entity(teacher).
entity(secretary).


/* WE ALSO NEED TO EXPLICITLY DEFINE ALL PERCIEVED RELATIONSHIPS BETWEEN
/* ENTITIES WITH THEIR CONNECTIONS DEFINTION.
/* THIS CAN BE ACHIEVED USING THE FOLLOWING DEFINITION FORMAT

/* CONN  NAME        BETWEEN                          TYPE    INVERSE_NAME
```

```
/*   1   PAID        UNIVERSITY_EMP, SALARY      1:M     PAID TO
/*
/* read as;-
/*
/*      university employee paid a salary
/*      salarty paid to a university employee
/*
/*      a university employee has one and only one salary
/*      the same salary may be paid to many university employees
/*
/* translates
/*     paid(Person,Salary) <- university_employee(Person,*,*,Salary,*,*).
/*     paid_to(Salary,Person) <- paid(Person,Salary).
/*
/*     dimension( paid in # ).
/*
/*  coded
/*       connect(paid,university_employee,salary,1,m,'paid to').
/*
/*  synonym
/*        paid - earn


connect( paid, university_employee, salary, 1, m,paid_to).

     paid(Person,Salary) <- university_employee(Person,Salary,*,*,*,*).
     paid_to(Salary,Person) <- paid(Person,Salary).

     dimension( age,'years' ).
     dimension( value,'pounds' ).
     dimension( salary,'pounds' ).

/*   synonym(paid,earns).

connect( aged, person, age, 1, m, '').
connect( named, person, name, 1, m, '').
connect( resident_at,student, home_address, 1, m, '' ).
connect( resident_at,person, address, 1, m, '' ).
/*connect( have_as, unknown, unknown, n, m, '').
connect( located_at, sponsor, address, 1, m, '').
connect( called, sponsor, name, 1, 1, '').
connect( sponsor_to, sponsor, student, n, 1, sponsored_by).
connect( lecturer_to, teacher, student, n, m, lectured_by).
connect( supervisor_to, teacher, post_grad, n, m, supervised_by).
connect( personal_tutor_to, teacher, under_grad, n, 1,
                                                  personal_tutee_of).
connect( secretary_to, professor_sec, professor, 1, 1, in_charge_of).
connect( work_for, departmental_sec, teacher, n, m, has_working_for).
connect( member_of, student, department, n, m, belongs_to_by).
connect( attend, student, course, n, m, attended_by).
connect( in, student, year_of_study, 1, m, '').
connect( working_in, university_employee, department, 1, m,
                has_working_in_it).
connect( belongs_to, person, faculty, 1, m, belonged_to).
connect( taught_by, course, teacher, 1, m, teacher_for).
connect( identified_by,person, ident_number, 1, 1, '').
connect( has_gender, person, sex, 1, m, '').
connect( classified_as, nationality, type_of_nat, 1, m, '').
connect( pays, sponsor, fee, n, m, paid_by).
connect( dependent_on, fee, attendance, n, m, '').
connect( has_a, student, nationality, 1, m, '').
connect( type_of, student, attendance, 1, m, '').
connect( depends_on, fee, type_of_nat, n, m, '').
```

```
connect( charged, student, fee, 1, m, charged_to).
connect( member_to, department, faculty, 1, m, made_up_of).
connect( taught_in, course, session, 1, m, '').
connect( taught_at, course, level, 1, m, '').
/* connect( called, course, code, 1, 1, '').


/* SPECIFICATION OF INDIVIDUAL ATTRIBUTES
/*
attributes(code,fchar,4).
attributes(session,one_of,["FIRST","SECOND","THIRD"]).
attributes(level,one_of,["ONE","TWO"]).
attributes(teacher,reference,teacher).
attributes(room,vchar,4).
attributes(id_num,integer,[100000,999999]).
attributes(age,integer,[0,120]).
/* attributes(age,composite,agerl).
/* attributes(date_of_birth,fchar,8).
/*  attributes(date_of_birth,composite,dbrule).
attributes(sex,one_of,["MALE","FEMALE"]).
attributes(person_type,one_of,[student,university_employee]).
attributes(student_type,one_of,[post_grad,under_grad]).
attributes(university_employee_type,one_of,[secretary,teacher]).
attributes(secretary_type,one_of,[departmental_sec,professor_sec]).
attributes(teacher_type,one_of,[professor,senior_lecturer,lecturer]).
attributes(nationality,vchar,10).
attributes(sponsor,reference,sponsor).
attributes(attendance_type,one_of,["FULL TIME","PART TIME"]).
attributes(year_of_study,one_of,[1,2,3,4]).
attributes(personal_tutor,reference,teacher).
attributes(salary,real,[5,2]).
attributes(department_id,reference,department).
attributes(ni,fchar,9).
attributes(tax_code,fchar,9).
attributes(professor,reference,professor).
attributes(nationality_classify,one_of,["UK","EEC","OTHER"]).
attributes(value,real,[4,2]).
attributes(department_code,vchar,4).
attributes(department_name,vchar,40).
attribute(faculty,reference,faculty).
attributes(faculty_name,one_of,["SCIENCE","ARTS",
                                "MEDICINE","LAW","ENGINEERING"] ).
attributes(student_id,reference,student).
attributes(course_id,reference,course).
attributes(post_grad,reference,post_grad).
attributes(name,composite,namerule).
attributes(title,one_of,["MR","MS","MISS","MRS","DR","PROF"]).
attributes(firstname,vchar,15).
attributes(surname,vchar,15).
attributes(sponsor_name,composite,sponsor_namerule).
attributes(address,composite,addressrule).
attributes(home_address,composite,addressrule).
attributes(street,vchar,15).
attributes(town,vchar,15).
attributes(county,vchar,15).
attributes(country,vchar,15).
attributes(home_street,vchar,15).
attributes(home_town,vchar,15).
attributes(home_county,vchar,15).
attributes(home_country,vchar,15).


database_term(departmental_sec,[departmental,sec]).
```

```
database_term(departmental_sec,[department,sec]).
database_term(id_num,[id,num]).
database_term(date_of_birth,[date,of,birth]).
database_term(attendance_type,[attendance,type]).
database_term(student_type,[student,type]).
database_term(department_id,[department,id]).
database_term(course_id,[course,id]).
database_term(belongs_to_by,[belongs,to,by]).
database_term(belonged_to,[belonged,to]).
database_term(student_id,[student,id]).
database_term(belongs_to,[belongs,to]).
database_term(under_grad,[under,grad]).
database_term(personal_tutor,[personal,tutor]).
database_term(personal_tutor_to,[personal,tutor,to]).
database_term(post_grad,[post,grad]).
database_term(sponsor_name,[sponsor,name]).
database_term(nationality_classify,[nationality,classify]).
database_term(professor_sec,[professors,sec]).
database_term(university_employee,[university,employee]).
database_term(university_employee,[university,emp]).
database_term(department_name,[department,name]).
database_term(faculty_name,[faculty,name]).
database_term(senior_lecturer,[senior,lecturer]).
database_term(tax_code,[tax,code]).
database_term(home_address,[home,address]).
database_term(home_street,[home,street]).
database_term(home_county,[home,county]).
database_term(home_country,[home,country]).
database_term(home_town,[home,town]).
database_term(resident_at,[resident,at]).
database_term(taught_at,[taught,at]).
database_term(type_of,[type,of]).
database_term(has_gender,[has,gender]).
database_term(depends_on,[depends,on]).
database_term(type_of_nat,[type,of,nat]).
database_term(attended_by,[attended,by]).
database_term(charged_to,[charged,to]).
database_term(classified_as,[classified,as]).
database_term(has_member,[has,member]).
database_term(taught_in,[taught,in]).
database_term(paid_by,[paid,by]).
database_term(paid_to,[paid,to]).
database_term(work_in,[work,in]).
database_term(has_working_in_it,[has_working,in,it]).
database_term(working_in_it,[working,in,it]).
database_term(working_in,[working,in]).
database_term(dependent_on,[dependent,on]).
database_term(sponsor_to,[sponsor,to]).
database_term(sponsored_by,[sponsored,by]).
database_term(personal_tutee_of,[personal,tutee,of]).
database_term(personal_tutor_of,[personal,tutor,of]).
database_term(taught_by,[taught,by]).
database_term(teacher_for,[teacher,for]).
database_term(identified_by,[identified,by]).
database_term(work_under,[work,under]).
database_term(has_working_for,[has,working,for]).
database_term(has_a,[has,a]).
/*database_term(have_as,[have,as]).
database_term(lectured_by,[lectured,by]).
database_term(work_for,[work,for]).
database_term(in_charge_of,[in,charge,of]).
database_term(member_of,[member,of]).
database_term(member_to,[member,to]).
```

```
database_term(made_up_of,[made,up,of]).
database_term(supervised_by,[supervised,by]).
database_term(supervisor_to,[supervisor,to]).
database_term(secretary_to,[secretary,to]).
database_term(lecturer_to,[lecturer,to]).
database_term(located_at,[located,at]).

  database_term(otherthan,[other,than]).
  database_term(lessthan,[less,than]).
  database_term(olderthan,[older,than]).
  database_term(youngerthan,[younger,than]).



/* plural(singular,plurals)
 plural(post_grad,post_grads).
 plural(tax_code,tax_codes).
 plural(under_grad,under_grads).
 plural(department_name,department_names).
 plural(departmental_sec,departmental_secs).
 plural(faculty_name,faculty_names).
 plural(id_num,id_nums).
 plural(professor_sec,professor_secs).
 plural(date_of_birth,date_of_births).
 plural(sponsor_name,sponsor_names).
 plural(senior_lecturer,senior_lecturers).
 plural(person_type,person_types).
 plural(student_type,student_types).
 plural(university_employee_types,university_employee_types).
 plural(secretary_type,secretary_types).
 plural(teacher_type,teacher_types).
 plural(home_address,home_addresses).
 plural(university_employee,university_employees).
 plural(attendance_type,attendance_types).
 plural(year_of_study,year_of_studies).
plural(personal_tutor,personal_tutors).
plural(course,courses).
plural(fee,fees).
plural(age,ages).
plural(give,gives).
plural(take,takes).
plural(room,rooms).
plural(professor,professors).
plural(lecturer,lecturers).
plural(department,departments).
plural(faculty,faculties).
plural(person,people).
plural(student,students).
plural(teacher,teachers).
plural(teach,teaches).
plural(secretary,secretaries).
plural(code,codes).
plural(session,sesions).
plural(lecture,lectures).
plural(level,levels).
plural(name,names).
plural(title,titles).
plural(attend,attends).
plural(firstname,firstnames).
plural(secondname,secondnames).
plural(surname,surnames).
plural(address,addresses).
plural(sex,sexes).
```

```
plural(nationality,nationalities).
plural(sponsor,sponsors).
plural(salary,salaries).
plural(ni,nis).
plural(value,values).
plural(work,works).
plural(grad,grads).
plural(graduate,graduates).
plural(street,streets).
plural(county,counties).
plural(country,countries).
plural(town,towns).
```

```
/* Analyses the query words in terms of the database model

analyse(Str,Lu,Ru) <- qst_to_qli(Str,List) &
                      / & label(lanalyse) &
                      swap_plurals(List,Listplural) &
                      swap_synonyms(Listplural,Listsynon) &
                      db_terms(Listsynon,Listdbterm1) &
                      swap_synonyms(Listdbterm1,Listdbterm2) &
                      translate(Listdbterm2,Listtran) &
                      uppercase_data(Listdbterm2,Listtran,Listupp) &
                      identify_quoted_terms(Listupp,Listquote) &
                      apostro(Listquote,Listtran,Aposquery,Apostran) &
                      unite_unknown(Apostran,Aposquery,Lf,Lrf) &
                      remove_filler(Lf,Lrf,Lu,Ru).
```

```
/*********************************************************/
/*                                                       */
/*   Eliminate words with a single apostrophe            */
/*   as having an implied connection                     */
/*                                                       */
/*********************************************************/
apostro([Hqin!Tqin],[Htin!Ttin],[Hqout![dummy!Tqout]],
                          [Htout![inter_relation!Ttout]]) <-
                                        single_strop(Hqin,Hqout) & / &
                                        obtain(Hqout,Htin,Htout) &
                                        apostro(Tqin,Ttin,Tqout,Ttout).
apostro([Hqin!Tqin],[Htin!Ttin],[Hqin!Tqout],[Htin!Ttout])  <-
                                        apostro(Tqin,Ttin,Tqout,Ttout).
apostro([],[],[],[]).


single_strop(Hqin,Hqout) <-  st_to_at(Shqin,Hqin) &
                             substring(Shqin,'''',S1,1) & / &
                             stlen(Shqin,L) &
                             S3 := S1 + 1 &
                             S2 := L - S3 &
                             substring(Shqin,Sh2,S3,S2) &
                             ¬ substring(Sh2,'''',S4,1) &
                             substring(Shqin,Shqout,0,S1) &
                             st_to_at(Shqout,Hqout).


obtain(Hqout,unknown,Htout) <- Hqout is Htout & /.
obtain(Hqout,Htin,Htin).
```

```
/* append(L1,L2,L3) appends list1 and list2 giving list3
append([],L,L).
append([X!T1],T2,[X!T3]) <- append(T1,T2,T3).
```

```
query(D) <- repeat & nl &
          prst('Domain of interest:') & nl &
          writes(D) &
          ( entity(D) ; list_domains ) &
          / & nl &
          prst('Attributes of interest:') & nl &
          attributes_of_interest(D,A) &
          list_attributes(A,10) & nl &
          prst('Connection of interest:') & nl &
          connection_of_interest(D,C,[]) &
          list_attributes(C,10) & nl &
          st_to_at(Sd,D) &
          stconc(Sd,'_type',S) &
          st_to_at(S,As) &
          (
           ( attributes(As,one_of,Tl) &
             prst('Types of domain:   ') & prst(D) & nl &
             list_attributes(Tl,10) & nl ) ; true ).
q <-
          entity(D) & nl & nl &
          prst('Domain of interset:    ') &
          prst(D) &
          nl &
          prst('Attributes of interest:') & nl &
          attributes_of_interest(D,A) &
          list_attributes(A,10) & nl &
          prst('Connection of interest:') & nl &
          connection_of_interest(D,C,[]) &
          list_attributes(C,10) & nl &
          st_to_at(Sd,D) &
          stconc(Sd,'_type',S) &
          st_to_at(S,As) &
          (
           ( attributes(As,one_of,Tl) &
             prst('Types of domain:   ') & prst(D) & nl &
             list_attributes(Tl,10) & nl ) ; true ) &
          fail .

list_domains <- prst('Domains of interest are') & nl &
                entity(X) &
                prst('     ') &
                st_to_at(S,X) &
                stconc(S,'                                                    ',C) &
                substring(C,S2,0,34) &
                prst(S2) &
                fail.

list_attributes([],M) <- nl & / .
list_attributes(T,N) <- gt(N,60) & nl & list_attributes(T,10) & / .
list_attributes([H!T],N) <-
                tab(N) &
                M := N + 25 &
                prst(H) &
                list_attributes(T,M) .

attributes_of_interest(D,L) <- relation(D,N,C) &
                               subset(X,D) & / &
                               attributes_of_interest(X,L1) & / &
                               new_append(L1,C,L) & /.
attributes_of_interest(D,L) <- relation(D,N,L) & /.
attributes_of_interest(D,B) <- subset(M,D) &
                                  attributes_of_interest(M,B) .
```

```
/* append(L1,L2,L3) appends list1 and list2 giving list3
new_append([],L,L).
new_append([X!T1],T2,T3) <- member(X,T2) & new_append(T1,T2,T3).
new_append([X!T1],T2,[X!T3]) <- new_append(T1,T2,T3).

<- reconsult( member ).

connection_of_interest(D,L,H) <- con(D,C,[]) &
                                 subset(X,D) & / &
                                 connection_of_interest(X,L1,H) & / &
                                 new_append(C,L1,L) & /.
connection_of_interest(D,C,[]) <- con(D,C,[]) .

con(B,[A!L],R) <- connect(A,B,C,D,E,F) &
                  ( ¬ member(A,R) ) &
                  new_append([A],R,P) &
                  con(B,L,P) & / .
con(C,[A!L],R) <- connect(A,B,C,D,E,'') &
                  ( ¬ member(A,R) ) &
                  new_append([A],R,P) &
                  con(C,L,P) & / .
con(C,[F!L],R) <- connect(A,B,C,D,E,F) &
                  ( ¬ member(F,R) ) &
                  new_append([F],R,P) &
                  con(C,L,P) & / .
con(B,[],R).
```

```
/* succeeds if the list is a list of variables
varl([H!T]) <- var(H) & varl(T) & / .
varl([]).

bsql(C,T,Er) <-  ( ( ¬ substring(C,'select',0,6) ) ;
                  ( substring(C,' where ',S,7) );
                  ( substring(C,' order by ',P,10) );
                  ( substring(C,' group by ',Q,10) );
                  var(T);
                  varl(T) ) &
                   / &
                  sql(C,T,Er).
bsql(C,T,Er) <- stconc('describe ',C,DesC) &
                sql(DesC,TD,Errd) &
                ge(Errd,0) &
                create(TD,T,New,' where ') &
                stconc(C,New,NewC) &
                / &
                sql(NewC,T,Er).
bsql(C,T,Er) <- sql(C,T,Er).

create([A![B![C!TT]]],[H!T],R,W) <- var(H) & create(TT,T,R,W) & / .
create([HA![B![C!TT]]],[H!T],R,W) <- stconc(HA,' = ',S1) &
                                     stconc(W,S1,S2) &
                                     stt_to_att(SH,H) &
                                     stconc(S2,SH,R1) & / &
                                     create(TT,T,R2,' and ') &
                                     stconc(R1,R2,R) & /.
create([],[],'',G).


stt_to_att(SH,H) <- stringp(H) & / &
                    stconc('''',H,S1) &
                    stconc(S1,'''',SH) & /.
stt_to_att(SH,H) <- st_to_at(SH,H).
```

```
/*****************************************************************/
/*                                                               */
/*  Erase user asked parses                                      */
/*                                                               */
/*****************************************************************/
check_asked_condition(clear) <- axn(previously_checked,B,D,[],M) &
                                delax(D) &
                                fail.
check_asked_condition(clear).

check_asked_condition(In,Type,found) <-
                              previously_checked(In,Type,found) & / .
check_asked_condition(In,Type,assert) <-
                              P =..[previously_checked,In,Type,found] &
                              addax(P).
check_asked_condition(Ret,Type,check) <-
                              previously_checked(Ret,Type,Con) & / & fail.
check_asked_condition(Ret,Type,check) <-
                              ¬ previously_checked(Ret,Type,Con) &
                              P =..[previously_checked,Ret,Type,check] &
                              addax(P).


maintain_query(Q) <- P =..[previously_checked,Q,query,check] &
                     addax(P).

recall_query(Q) <- previously_checked(Q,query,check).
```

```
/*********************************************************/
/*                                                       */
/*  Compose the query units into a meaningful            */
/* concatenation                                         */
/*                                                       */
/*********************************************************/
compose(Ret,R12,Rep3) <- obtain_retrieval_list(R1i,Ret,Repa) &
                         verify_retrieval(R1i,R1,Rree) &
                         append(Rree,Repa,Rep) &
                         strip_leading_terms(Rep,Rep1) &
                         rationalise(Rep1,Rep2) &
                         strip_leading_terms(Rep2,Rep3) &
                         verify_context_flow(R1,Rep3,Out) &
                         check_user(R1,Rep3,R12,Rp2) .
```

```
conditionals("/=").
conditionals("<").
conditionals("<=").
conditionals("=").
conditionals(">").
conditionals(">=").
conditionals(rule).
conditionals(after).
conditionals(before).

intra_entity(olderthan,age,">").
intra_entity(youngerthan,age,"<").
intra_entity(older_than,age,">").
intra_entity(younger_than,age,"<").
intra_entity(aged,age,"=").
intra_entity(named,name,"=").
intra_entity(called,name,"=").
intra_entity(resident_at,address,rule).
intra_entity(resident_at,home_address,rule).
intra_entity(born,date_of_birth,"=").
```

```
connectors(and).
connectors(not).
connectors(or).
```

```
/*****************************************************************
/*                                                               */
/*  Consult user to make sure that the terms which we            */
/*  do not know the meaning of are in fact items                 */
/*  stored in the database                                       */
/*                                                               */
/*  If the user has not been previously asked and                */
/*  The term is not already specified as database term           */
/*  Then record the user has been asked and                      */
/*  Ask the user                                                 */
/*  read in the reply and                                        */
/*  act accordingly                                              */
/*  then if answer was yes try to re-analyse the query           */
/*                                                               */
/*****************************************************************/
consult_user_for_unknown([Hq!Tq],[unknown!Tp]) <-
                                ¬ previously_asked(Hq) &
                                ¬ stored_in_database(Hq) &
                                Askedterm =..[previously_asked,Hq] &
                                addax(Askedterm) & nl & nl &
                                prst('Do you believe that ') &
                                writes(Hq) &
                                prst(' is stored as a single ') &
                                prst('entity in the database') &
                                nl & nl &
                                readline(Reply) &
                                act_on(Reply,Hq) &
                                retry(lanalyse).
consult_user_for_unknown([Hq!Tq],[Hp!Tp]) <- ¬ Hp = unknown &
                                consult_user_for_unknown(Tq,Tp).
consult_user_for_unknown([],[]).


/*****************************************************************
/*                                                               */
/* Act on reply if answer was y or yes then add the              */
/* term to the user model as a stored item of data               */
/* if the term is made up of more than one word then             */
/* add to database of terms.                                     */
/*                                                               */
/*****************************************************************
/*                                                               */
/* Act on reply if answer was no then                            */
/* test to see if the term single is a word                      */
/* In which case fail reporting that the term is unknown         */
/* Otherwise fail and backtrack to the single terms              */
/*                                                               */
/*****************************************************************
act_on('y',Term) <- act_on('yes',Term).
act_on('yes',Term) <- add_term_stored(Term) .
act_on(No,Term) <- st_to_at(Sterm,Term) &
                   qst_to_qli(Sterm,Lterm) &
                   Lterm = [Dummy] &
                   test_db_term(Term) &
                   / & fail.
act_on(No,Term) <- st_to_at(Sterm,Term) &
                   qst_to_qli(Sterm,Lterm) &
                   Lterm = [Dummy] &
                   prst('I am unable to parse your query as I do') &
                   prst(' not understand the term ') &
                   writes(Term) & nl &
                   prst('Please enter a synonym for the term ') &
                   writes(Term) & nl &
```

```
                    prst('or Q to quit') &
                    nl & nl &
                    caseshift(Sterm,Slterm) &
                    readline(Sy) &
                    ¬ ( Sy = 'q'  ; Sy = 'Q' ) &
                    st_to_at(Sy,Sya) &
                    st_to_at(Slterm,Alterm) &
                    Synon =..[synonym,Alterm,Sya] &
                    addax(Synon) &
                    retry(lanalyse).

/*****************************************************************
/*                                                               */
/* A term has been defined as being made up of several           */
/* words in order to recognise it in the future we need          */
/* add the term to our known composite database terms            */
/*                                                               */
/*****************************************************************
add_to_term_database(Term) <- st_to_at(Sterm,Term) &
                              qst_to_qli(Sterm,Lterm) &
                              ¬ Lterm = [Dummy] &
                              Db_term =..[database_term,Term,Lterm] &
                              addax(Db_term).
add_to_term_database(Term).


/*****************************************************************
/*                                                               */
/* A term has been defined as beleived to be stored in           */
/* the database it is therefore asserted as a stored             */
/* database term.                                                */
/*                                                               */
/*****************************************************************
add_term_stored(Term) <- st_to_at(Sterm,Term) &
                         caseshift(Sterm,Slterm) &
                         st_to_at(Slterm,Lterm) &
                         Stored_term =..[stored_in_database,Term]  &
                         addax(Stored_term) &
                         add_to_term_database(Term) &
                         Sltored_term =..[stored_in_database,Lterm]  &
                         addax(Sltored_term) &
                         add_to_term_database(Lterm) .

test_db_term(Term) <-
                      st_to_at(Sterm,Term) &
                      caseshift(Sterm,Slterm) &
                      st_to_at(Slterm,Lterm) &
                      database_term(C,D) &
                      member(Lterm,D).
```

```
<- pragma(list,1).

/* succeeds if the list is a list of variables
varl([H!T]) <- var(H) & varl(T) & / .
varl([]).

csql(C,T,Er) <-  ( ( ¬ substring(C,'select',0,6) ) ;
                   ( substring(C,' where ',S,7) );
                   ( substring(C,' order by ',P,10) );
                   ( substring(C,' group by ',Q,10) );
                   var(T);
                   varl(T) ) &
                    / &
                  sql(C,T,Er).
csql(C,T,Er) <- stconc('describe ',C,DesC) &
                sql(DesC,TD,Errd) &
                ge(Errd,0) &
                create(TD,T,New,' where ') &
                stconc(C,New,NewC) &
                / &
                sql(NewC,TT,Er) &
                list_eqn(T,TT).
csql(C,T,Er) <- sql(C,T,Er).

create([A![B![C!TT]]],[H!T],R,W) <- var(H) & create(TT,T,R,W) & / .
create([HA![B![C!TT]]],[H!T],R,W) <- ¬ H = constraint(N) & / &
                                     stt_to_att(SH,H) &
                                     form_condition(HA,W,SH,R1,' = ') &
                                     / &
                                     create(TT,T,R2,' and ') &
                                     stconc(R1,R2,R) & /.
create([HA![B![C!TT]]],[constraint(N)!T],R,W) <-
                                     constraint_con(R1,W,N,HA) &
                                     / &
                                     create(TT,T,R2,' and ') &
                                     stconc(R1,R2,R) & /.
create([],[],'',G).

form_condition(A,W,H,R,C) <- ¬ C = null &
                             stconc(A,C,S1) &
                             stconc(W,S1,S2) &
                             stconc(S2,H,R).
form_condition(A,W,H,'',null).

constraint_con(S,W,N,A) <- constraint(N,Sy1,N1,Sy2,N2) &
                           cond(Sy1,C1) &
                           st_to_at(N1S,N1) &
                           form_condition(A,W,N1S,S1,C1) &
                           cond(Sy2,C2) &
                           comp(W,S1,W2) &
                           st_to_at(N2S,N2) &
                           form_condition(A,W2,N2S,S2,C2) &
                           stconc(S1,S2,S) & / .

comp(W,'',W).
comp(W,X,' and ') <- ¬ X = ''.

cond(null,null).
cond(eq,' = ').
cond(gt,' > ').
cond(lt,' < ').
cond(le,' <= ').
cond(ge,' >= ').
```

```
stt_to_att(SH,H) <- stringp(H) & / &
                    stconc('''',H,S1) &
                    stconc(S1,'''',SH) & /.
stt_to_att(SH,H) <- st_to_at(SH,H).

list_eqn([H!T],[HH!TT]) <- H eqn HH &
                           list_eqn(T,TT).
list_eqn([],[]).

ct(Rn,Bn) <-
      sql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],N) &
      gt(H,Rn) &
      lt(H,Bn) &
      writes([A,B,C,D,E,F,G,H,I]) & nl.
nct(Rn,Bn) <- H gtn Rn &
           H ltn Bn &
           csql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],N) &
           coutput(H,Ho) &
           writes([A,B,C,D,E,F,G,Ho,I]) & nl.
ct3 <-
     csql('select * from qpcat.catsingles',
                      [A1,B1,C1,'DITC16/83',E1,F1,G1,H1,I1],N1) &
     csql('select * from qpcat.catsingles',
                      [A2,B2,C2,'DITC1/82',E2,F2,G2,H2,I2],N2) &
      sql('select * from qpcat.catsingles',
                      [A,B,C,D,E,F,G,H,I],N) &
      gt(H,H1) &
      lt(H,H2) &
      writes([A,B,C,D,E,F,G,H,I]) & nl.
ct2 <-
      sql('select * from qpcat.catsingles',
                      [A1,B1,C1,'DITC16/83',E1,F1,G1,H1,I1],N1) &
      sql('select * from qpcat.catsingles',
                      [A2,B2,C2,'DITC1/82',E2,F2,G2,H2,I2],N2) &
      sql('select * from qpcat.catsingles',
                      [A,B,C,D,E,F,G,H,I],N) &
      gt(H,H1) &
      lt(H,H2) &
      writes([A,B,C,D,E,F,G,H,I]) & nl.

wct2 <-
system('query time',cms) &
      sql('select * from qpcat.catsingles',
                      [A1,B1,C1,D1,E1,F1,G1,H1,I1],N1) &
      D1 = 'DITC16/83' & / &
system('query time',cms) &
      sql('select * from qpcat.catsingles',
                      [A2,B2,C2,D2,E2,F2,G2,H2,I2],N2) &
 nl &
system('query time',cms) &
      sql('select * from qpcat.catsingles',
                      [A,B,C,D,E,F,G,H,I],N) &
      D2 = 'DITC1/82' &
      gt(H,H1) &
      lt(H,H2) &
nl &
system('query time',cms) &
nl &
      writes([A,B,C,D,E,F,G,H,I]) & nl.

nct2 <-
     csql('select * from qpcat.catsingles',
```

```
                    [A1,B1,C1,'DITC16/83',E1,F1,G1,H1,I1],N1) &
     csql('select * from qpcat.catsingles',
                    [A2,B2,C2,'DITC1/82',E2,F2,G2,H2,I2],N2) &
     H gtn H1 &
     H ltn H2 &
     csql('select * from qpcat.catsingles',[A,B,C,D,E,F,G,H,I],N) &
     coutput(H,Ho) &
     writes([A,B,C,D,E,F,G,Ho,I]) & nl.

ccat([A2,B2,C2,D2,E2,F2,G2,H2,I2]) <-
     sql('select * from qpcat.catsingles where author = ''DEC''',
                    [A2,B2,C2,D2,E2,F2,G2,H2,I2],N2).
ncat([A2,B2,C2,D2,E2,F2,G2,H2,I2]) <-
     csql('select * from qpcat.catsingles',
                    [A2,B2,C2,D2,E2,F2,G2,H2,I2],N2).


count1 <-
     addax(c(0)) &
     csql('select * from qpcat.catsingles',
                    [A1,B1,C1,D1,E1,F1,G1,H1,I1],N1) &
     c(N) &
     M := N + 1 &
     delax(c(N)) &
     addax(c(M)) &
     D1= 'AAAAAAA2' &
     c(L) & write(L) & nl .
```

```
check_user(R1,Rep,R12,Rep2) <-
                              retrieve_output(R1,R12) &
                              ask_about('OBJECTS TO BE FETCHED',ret,R12) &
                              condition_output(Rep,Rep2) &
                              ask_about('SUBJECT TO',con,Rep2).

ask_about(String,Type,Ra) <- check_asked_condition(Rb,Type,found) & / &
                              Rb = Ra & /.

ask_about(String,Type,[]) <- check_asked_condition(Ra,Type,check) &
                                display &
                              nl &
                              prst(String) & nl & nl &
                              prst('   UN-CONSTRAINED RETREIVAL ') & nl &
                              nl &
                              prst('is this what you want for the ') &
                              prst(String) & prst(' clause ? y/n') & nl &
                              readli(Answer) &
                              ( Answer ='y';
                                Answer ='yes' ) &
                              check_asked_condition([],Type,assert) & / .
ask_about(String,Type,Ra) <- check_asked_condition(Ra,Type,check) &
                                display &
                              prst(String) & nl & nl &
                              printlist(Ra) & nl & nl &
                              prst('is this what you want for the ') &
                              prst(String) & prst(' clause ? y/n') & nl &
                              readli(Answer) &
                              ( Answer ='y';
                                Answer ='yes' ) &
                              check_asked_condition(Ra,Type,assert) & / .

retrieve_output([relation![[R]]],S) <- / &
                                       atributes_of_interest(R,S).
retrieve_output([[object![[A,B]]]!T],[[A![B![',']]]!Y]) <- / &
                                                  retrieve_output(T,Y).
retrieve_output([H!T],[A!B]) <-
                                 retrieve_output(H,A) &
                                 retrieve_output(T,B).
retrieve_output([],[]).

atributes_of_interest(D,L) <- relation(D,N,C) &
                              add_entity(C,D,C1) &
                              subsets(X,D) & / &
                              atributes_of_interest(X,L1) & / &
                              nw_append(L1,C1,L) & /.
atributes_of_interest(D,L1) <- relation(D,N,L) & / &
                               add_entity(L,D,L1).
atributes_of_interest(D,B) <- subsets(M,D) &
                              atributes_of_interest(M,B) .

/* append(L1,L2,L3) appends list1 and list2 giving list3
nw_append([],L,L).
nw_append([E![X![',']!T1]]],T2,T3) <- member(X,T2) & nw_append(T1,T2,T3).
nw_append([E![X![',']!T1]]],T2,[E![X![',']!T3]]]) <- nw_append(T1,T2,T3).

add_entity([Sh!St],R,[R![Sh![',']!Y]]]) <- add_entity(St,R,Y).
add_entity([],R,[]).
```

```
professor(100001 ).
lecturer(100002 ).
lecturer(100003 ).

salary(X) <- var(X) & /.
salary(X) <- numb(X).

student(100004,'HOME STREET A','TOWN A','HOME COUNTY A','HOME COUNTRY A',
        'NATIONALITY A','SPONSOR A','FT',1,'UNDER_GRAD').
student(100005,'HOME STREET A','TOWN A','HOME COUNTY A','HOME COUNTRY A',
        'NATIONALITY A','SPONSOR A','FT',3,'UNDER_GRAD').
student(100006,'HOME STREET T','TOWN T','HOME COUNTY T','HOME COUNTRY T',
        'NATIONALITY A','SPONSOR A','FT',3,'UNDER_GRAD').
student(100007,'HOME STREET Y','TOWN Y','HOME COUNTY Y','HOME COUNTRY Y',
        'NATIONALITY X','SPONSOR A','PT',2,'POST_GRAD').
student(100008,'HOME STREET Y','TOWN Y','HOME COUNTY Y','HOME COUNTRY Y',
        'NATIONALITY Q','SPONSOR A','FT',1,'POST_GRAD').
student(100012,'HOME STREET T','TOWN T','HOME COUNTY T','HOME COUNTRY T',
        'NATIONALITY A','SPONSOR B','FT',3,'UNDER_GRAD').
student(100013,'HOME STREET Y','TOWN Y','HOME COUNTY Y','HOME COUNTRY Y',
        'NATIONALITY X','SPONSOR B','PT',2,'POST_GRAD').
student(100014,'HOME STREET Y','TOWN Y','HOME COUNTY Y','HOME COUNTRY Y',
        'NATIONALITY Q','SPONSOR C','FT',1,'POST_GRAD').

under_grad(100004,100001).
under_grad(100005,100002).
under_grad(100006,100001).
under_grad(100012,100001).

university_employee(100001,20730.23,'CS',
                    'CZ253476Q',R23,'TEACHER').
university_employee(100002,15000.43,'CS',
                    'CZ253476Q',R23,'TEACHER').
university_employee(100003,16500.21,'CS',
                    'CZ253476Q',R23,'TEACHER').
university_employee(100009,11000.13,'CS',
                    'CZ253476Q',R23,'SECRETARY').
university_employee(100010,12340.45,'CS',
                    'CZ253476Q',R23,'SECRETARY').
university_employee(100011,15000.23,'CS',
                    'CZ253476Q',R23,'SECRETARY').

secretary(100009,'DEPARTMENTAL').
secretary(100010,'DEPARTMENTAL').
secretary(100011,'PROFESSOR').

teacher(100001,'G1','PROFESSOR').
teacher(100002,'G1','LECTURER').
teacher(100003,'G1','LECTURER').

professor_sec(100011,100001).

fee(attendance_type,nationality_classify,student_type,value).


sponsor('SPONSOR A','SPON STREET','SPON TOWN','SP COUNTY','SP COUNTRY').
sponsor('SPONSOR B','SPNB STREET','SPNB TOWN','SPB COUNTY','SP COUNTRY').

department('CS','COMPUTER SIENCE','SCIENCE').

faculty('SCIENCE').

attend(100004,'01CS').
```

```
attend(100005,'01CS').
attend(100006,'02CS').
attend(100004,'02CS').
attend(100004,'03CS').
attend(100005,'03CS').
attend(100006,'03CS').
attend(100004,'66CS').
attend(100007,'03CS').
attend(100007,'34CS').
attend(100008,'32CS').
attend(100008,'07CS').
attend(100007,'07CS').
attend(100006,'07CS').
attend(100005,'07CS').
attend(100004,'07CS').

course('01CS','FIRST','ONE',100001).
course('02CS','SECOND','ONE',100002).
course('03CS','FIRST','TWO',100003).
course('07CS','SECOND','TWO',100003).
course('66CS','SECOND','ONE',100002).
course('32CS','THIRD','ONE',100002).
course('34CS','FIRST','TWO',100001).
session('FIRST').
session('SECOND').
session('THIRD').
level('ONE').
level('TWO').
ins <- person(A,B,C,D,E,F,G,H,I,J,K) &
       listst([A,B,C,D,E,F,G,H,I,K,J],Str) &
       stconc('insert into testp values(',Str,Str1) &
       stconc(Str1,' ) ',Str2) &
       sql(Str2,ERR) & fail.

listst([A],S2) <- stringp(A) & / &stconc('''',A,S1) & stconc(S1,'''',S2).
listst([A],S) <- / & st_to_at(S,A).
listst([A!B],Str) <- listst([A],S1) &
                     listst(B,S2) &
                     stconc(S1,' , ',S3) &
                     stconc(S3,S2,Str) .

person(100001,46,'PROF','ARTHUR','SMITH','12 HIGH ST',
       'LIVERPOOL','MERSEYSIDE','ENGLAND','MALE','UNIV_EMP').
person(100002,57,'MR','BARRY','SMITH','12 HIGH ST',
       'NEWCASTLE','TYNESIDE','ENGLAND','MALE','UNIV_EMP').
person(100003,32,'MRS','CLARE','SMITH','12 HIGH ST',
       'EPPING','ESSEX','ENGLAND','FEMALE','UNIV_EMP').
person(100004,23,'MRS','DENNIS','SMITH','12 HIGH ST',
       'TORQUAY','CORNWALL','ENGLAND','FEMALE','STUDENT').
person(100005,18,'MR','ERIC','SMITH','12 HIGH ST',
       'NOTTINGHAM','NOTTS','ENGLAND','MALE','STUDENT').
person(100006,21,'MR','FRANK','SMITH','12 HIGH ST',
       'NOTTINGHAM','NOTTS','ENGLAND','MALE','STUDENT').
person(100007,20,'MR','GEORGE','SMITH','12 HIGH ST',
       'TOWNF','COUNTYF','ENGLAND','MALE','STUDENT').
person(100008,20,'MR','HARRY','SMITH','12 HIGH ST',
       'TORQUAY','CORNWALL','ENGLAND','MALE','STUDENT').
person(100009,18,'MISS','INGRID','SMITH','12 HIGH ST',
       'LIVERPOOL','MERSEYSIDE','ENGLAND','FEMALE','UNIV_EMP').
person(100010,19,'MR','JOHN','SMITH','12 HIGH ST',
       'LIVERPOOL','MERSEYSIDE','ENGLAND','MALE','UNIV_EMP').
person(100011,23,'MR','KEN','SMITH','12 HIGH ST',
       'LIVERPOOL','MERSEYSIDE','ENGLAND','MALE','UNIV_EMP').
```

```
person(100012,22,'MR','LARRY','SMITH','12 HIGH ST',
        'NOTTINGHAM','NOTTS','ENGLAND','MALE','STUDENT').
person(100013,33,'MR','MORRIS','JONES','12 HIGH ST',
        'TOWNF','COUNTYF','COUNTRYF','MALE','STUDENT').
person(100014,31,'MRS','NORMA','BROWN','12 HIGH ST',
        'TOWNF','COUNTYF','COUNTRYF','FEMALE','STUDENT').


supervise(100003,100007).
supervise(100001,100007).
supervise(100002,100008).
supervise(100003,100013).
supervise(100003,100014).


post_grad(100007).
post_grad(100008).
post_grad(100013).
post_grad(100014).

nationality_classify(nationality,classification).

belongs_to(100004,'CS').
belongs_to(100005,'CS').
belongs_to(100006,'CS').
belongs_to(100007,'CS').
belongs_to(100008,'CS').
belongs_to(100008,'PHY').
belongs_to(100012,'CS').
belongs_to(100013,'CS').
belongs_to(100014,'CS').
```

```
/*
db_terms(A,B) <- database_term(C,D) &
                 locate(D,A,Before,After) &
                 append(Before,[C],E) &
                 append(E,After,R) &
                 db_terms(R,B).
db_terms(A,A).


locate([H!T],[H!After],[],A) <- loc(T,After,A) .
locate([H!T],[Ha!After],[Ha!Before],A) <- locate([H!T],After,Before,A).
locate([],After,[],After).

loc([H!T],[H!After],A) <- loc(T,After,A).
loc([],After,After).
```

```
display <-
          system('clear') &
          recall_query(Q) &
          prst('CURRENT QUERY') & nl & nl &
          prst(Q) & nl & nl &
          prst('----------------------------------------') & nl &
          check_asked_condition(Ra,ret,found) &
          prst('FETCH') & nl & nl &
          printlist(Ra) & nl & nl &
          prst('----------------------------------------') & nl &
          check_asked_condition(Rc,con,found) &
          prst('SUBJECT TO') & nl & nl &
          printlist(Rc) & nl & nl &
          prst('----------------------------------------') & nl &
          nl & / .
display.
```

```
function_check('domain') <- / &
                repeat &
                system('clear') & nl &
                prst('THE FOLLOWING ENTITIES ARE THE PRIMARY DOMAINS ') &
                prst('OF INTEREST') & nl & nl & nl &
                ¬ primary_entity_list &
                nl & nl & nl &
                prst('ENTER THE NAME OF A DOMAIN TO OBTAIN FURTHER ') &
                prst('INFORMATION') & nl & nl &
                prst('ALTERNATIVLY HIT RETURN FOR QUERYING MODE') & nl &
                readline(D) &
                domain_inf(D) &
                D = '' & / &
                fail.
function_check('load') <- nl & loaduser & / & fail.
function_check('save') <- nl & saveuser & / & fail.
function_check('new') <-  nl & new_user & / & fail.
function_check(A).

domain_inf('') <- / .
domain_inf(Ds) <- st_to_at(Ds,D) &
                  entity(D) &
                  domain_ssets(D,[],S) &
                  output_ssets(D,S,Reps) &
                  domain_connections(D,C,[]) &
                  output_cons(D,C,Repc) &
                  domain_attributes(D,A) &
                  output_atts(D,A,Repa) .

primary_entity_list <- entity(X) &
                       ¬ subsets(Y,X) & prst('    ') &
                       writes(X) & fail.


domain_attributes(D,L) <- relation(D,N,C) &
                          subset(X,D) & / &
                          domain_attributes(X,L1) & / &
                          new_append(L1,C,L) & /.
domain_attributes(D,L) <- relation(D,N,L) & /.
domain_attributes(D,B) <- subset(M,D) &
                          domain_attributes(M,B) .


domain_connections(D,L,H) <- con(D,C,[]) &
                             subset(X,D) & / &
                             domain_connections(X,L1,H) & / &
                             new_append(C,L1,L) & /.
domain_connections(D,C,[]) <- con(D,C,[]) .

domain_ssets(D,R,G) <- subset(D,S) &
                       ¬ member(S,R) &
                       new_append(R,[S],G1) & / &
                       domain_ssets(D,G1,G).
domain_ssets(D,R,R).

output_atts(D,A,Repa) <-
                    repeat &
                    system('clear') &
                    nl &
                    prst('ATTRIBUTES FOR ENTITY - ') &
                    writes(D) & nl & nl &
                    prt_oblst(A) & nl & nl &
                    prst('TYPE ATTRIBUTE NAME FOR FURTHER INFOMATION') &
```

```
                    nl &
                    prst('OR RETURN TO QUIT') & nl &
                    readline(Repa) &
                    att_inf(Repa) &
                    Repa = '' & / & nl .

att_inf('').
att_inf(Att) <- nl & prst('INFORMATION ON ATTRIBUTE - ') &
                st_to_at(Att,Aatt) &
                writes(Aatt) & nl &
                attributes(Aatt,Type,M) & nl &
                prst('    ') &
                writes_att(Type,M) &
                nl & nl &
                prst('HIT RETURN TO CONTINUE') & nl &
                readline(Return).

output_ssets(D,[],Reps) <- /.
output_ssets(D,S,Reps) <-
                    system('clear') &
                        nl & prst('SUBSETS FOR ENTITY - ') &
                    writes(D) & nl & nl &
                    prt_oblst(S) & nl & nl &
                    prst('TYPE SUBSET NAME FOR FURTHER INFOMATION') &
                    nl &
                    prst('OR RETURN TO CONTINUE') & nl &
                    readline(Sub) &
                    domain_inf(Sub).

output_cons(D,C,Repc) <-
                    repeat &
                    system('clear') &
                    nl & prst('CONNECTIONS FOR ENTITY - ') &
                    writes(D) & nl & nl &
                    prt_oblst(C) & nl & nl &
                    prst('TYPE CONNECTION NAME FOR FURTHER INFOMATION') &
                    nl &
                    prst('OR RETURN TO CONTINUE') & nl &
                    readline(Repc) &
                    con_inf(Repc) & Repc = '' & / .


con_inf('').
con_inf(Con) <-
                st_to_at(Con,Acon) &
                connect( Acon, E1, E2, N1, N2, Incon) & / &
                write_connect( Acon, E1, E2, N1, N2, Incon).
con_inf(Con) <-
                st_to_at(Con,Acon) &
                connect( Incon, E1, E2, N1, N2, Acon) &
                write_connect( Acon, E2, E1, N2, N1, Incon).

write_connect( Acon, E1, E2, N1, N2, Incon) <-
                nl & prst('INFORMATION ON CONNECTION - ') &
                rites(Acon) & nl & nl &
                prst(' CONNECTION OF THE FORM ') &
                nl & nl &
                write_line(Acon,E1,E2,N1) & nl &
                prst(' AND ') &  nl &
                write_line(Incon,E2,E1,N2) &
                nl & nl &
                prst('HIT RETURN TO CONTINUE') & nl &
                readline(Return).
```

```
rites(1) <- prst(' a ') & /.
rites(m) <- prst(' many ') & /.
rites(n) <- prst(' many ') & /.
rites('') <- prst(' had by ') & /.
rites(N) <- var(N) & prst(' unspecified ') & /.
rites(N) <- database_term(N,L) & printlist(L) & /.
rites(N) <- writes(N).


/*****************************************************************/
/*                                                               */
/*  prints out a list as a string of elements separated          */
/*  by a space. output directed to the console                   */
/*                                                               */
/*****************************************************************/
prt_oblst([]) <- /.
prt_oblst([H!T]) <- atom(H) & / &
                    st_to_at(Hs,H) &
                    prst(Hs) &
                    prst('    ') &
                    prt_oblst(T) & /.
prt_oblst([H!T]) <- numb(H) & / &
                    prst(H) &
                    prst('    ') &
                    prt_oblst(T) & /.
prt_oblst([H!T]) <- stringp(H) & / &
                    prst(H) &
                    prst('    ') &
                    prt_oblst(T) & /.
prt_oblst([H!T]) <- prt_oblst(H) &
                    prst('    ') &
                    prt_oblst(T) & /.

wc <-
               connect( Acon, E1, E2, N1, N2, Incon) &
               write_connect( Acon, E1, E2, N1, N2, Incon).
cw <-
               connect( Incon, E1, E2, N1, N2, Acon) &
               write_connect( Acon, E2, E1, N2, N1, Incon).

write_line(Acon,E1,E2,1) <- / &
                            prst('   A ') &
                            rites(E1) & prst(' is ') &
                            rites(Acon) & prst(' only one ') &
                            rites(E2).
write_line(Acon,E1,E2,m) <- / &
                            prst('   The same ') &
                            rites(E1) & prst(' may be ') &
                            rites(Acon) & prst(' many ') &
                            sing_plural(E2,P2) & rites(P2).
write_line(Acon,E1,E2,n) <- write_line(Acon,E1,E2,m).


writes_att(one_of,M) <- prst('Attribute value is one of') & nl &
                        prst('          ') & prt_oblst(M).
writes_att(integer,M) <- prst('Attribute is an integer in ') &
                         prst('the range') & nl &
                         prst('          ') & prt_oblst(M).
writes_att(real,[B,A]) <- prst('Attribute is a real with ') &
                          writes(B) & prst(' places before the') &
                          prst(' decimal point') & nl &
                          prst('          and ') &
```

```
                        writes(A) & prst(' places after').
writes_att(fchar,A) <- prst('Attribute is a character string of ') &
                       prst('fixed length ') & writes(A).
writes_att(vchar,A) <- prst('Attribute is a character string of ') &
                       prst('variable length up to ') & writes(A).
writes_att(reference,A) <- prst('Attribute is a reference to the ') &
                           prst('relation ') & writes(A).
writes_att(composite,A) <- prst('Attribute is a composite object ') &
                           prst('defined by the rule ') & writes(A).

sing_plural(E2,P2) <- plural(E2,P2) & /.
sing_plural(E2,E2) .
```

```
check(D,S) <-  check_for_double_definition(D) & nl &
               check_for_split_clause_definition(S).
/*****************************************************/
/*                                                   */
/*  CHECK FOR DOUBLE DEFINITION OF MODEL DATA        */
/*                                                   */
/*****************************************************/
check_for_double_definition(Pn) <- ax(Pn,D1,CN1) &
                                   ax(Pn,D2,CN2,I2) &
                                   D1 == D2 &
                                   ¬ I2 = 1 &
                                   ¬ CN1 = debug &
                                   ax(Pn,D2,CN2,I3) &
                                   ¬ I3 = 1 &
                                   nl &
                                   prst('Double definition of clause ') &
                                   nl & nl &
                                   writes(D1) & nl & nl &
                                   prst('In file(s) ') &
                                   writes(CN1) &
                                   ( ( ¬ CN1 = CN2 &
                                     prst('    ') &
                                     writes(CN2) & / ) ; true ) & nl & nl.
check_for_double_definition(null).
```

```
equivalance(X,X,X).
equivalance(X,Y,Z) <- equi(Y,X,Z).
equivalance(X,Y,Z) <- equi(X,Y,Z).

equi("=",">",">=").
equi("=","<","<=").
equi("=",">=",">=").
equi("=","<=","<=").
equi("<",">","/=").
equi("/=",X,"/=").
equi("=","/=",any).
```

```
explain(A,B,f) <- system('clear',cms) & nl &
                  remove_default(A,And) &
                  reverse(And,Rand) &
                  condition_output(Rand,Ra2) & / &
                  prst('We can conclude that no tuples ') &
                  prst('exist which satisfy the conditional clause ') &
                  nl & nl &
                  break_printlist(Ra2) & nl &
                  readli(JUNK).

break_printlist([]) <- /.
break_printlist([and!T]) <- / & nl & prst('   and') & nl &
                            break_printlist(T) & /.
break_printlist([or!T])  <- / & nl & prst('   or') & nl &
                            break_printlist(T) & /.
break_printlist([not!T]) <- / & nl & prst('   not') & nl &
                            break_printlist(T) & /.
break_printlist([H!T]) <- atom(H) & / &
                          st_to_at(Hs,H) &
                          prst(Hs) &
                          prst(' ') &
                          break_printlist(T) & /.
break_printlist([H!T]) <- numb(H) & / &
                          prst(H) &
                          prst(' ') &
                          break_printlist(T) & /.
break_printlist([H!T]) <- stringp(H) & / &
                          prst(H) &
                          prst(' ') &
                          break_printlist(T) & /.
break_printlist([H!T]) <- break_printlist(H) &
                          prst(' ') &
                          break_printlist(T) & /.
```

```
formalise_conditions(A,B,A,B) <- ¬ member([combine!T],A) & / .
formalise_conditions(A,B,C,D) <- formalise_condition(A,B,C1,D1) &
                                 ( ¬ A = C1 ; ¬ B = D1 ) & / &
                                 formalise_conditions(C1,D1,C,D).
formalise_conditions(A,B,A,B).

formalise_condition([[combine![Qcond]]!Tq],
                    [[combine![Tcond]]!Tt],Nq,Nt) <-
/*                  / &
/* write(['QCOND',Qcond]) &
/* write(['TCOND',Tcond]) &
                    combine(Qcond,Tcond,Nqc,Ntc) &
                    formalise_condition(Tq,Tt,Nq1,Nt1) &
                    append(Nqc,Nq1,Nq) &
                    append(Ntc,Nt1,Nt) .
/*formalise_condition([Hq!Tq], [Ht!Tt],[[Hq!Qr]], [[Ht!Tr]]) <-
/*                                  write('here0') &
/*                                  write(['Hq',Hq]) &
/*                                  write(['Tq',Tq]) &
/*                                  ¬ Tq = [] &
/*                                  formalise_condition(Tq,Tt,Qr,Tr).
formalise_condition([Hq!Tq], [Ht!Tt],[Hq!Qr], [Ht!Tr]) <-
/*                                   Tq = [] &
                                  formalise_condition(Tq,Tt,Qr,Tr).
formalise_condition([],[],[],[]).


combine([[combine![R]]!Qr],
        [[C![Tc]]!Tr],Q1,T1) <-
/*       write('here1') &
        combine(R,Tc,[Qa!Qb],[Ta!Tb]) &
/*write(['**Q2',Qa]) &
/*write(['**T2',Ta]) &
/*write(['**Qb',Qb]) &
/*write(['**Tb',Tb]) &
        ( Tb = [] &
         append([Qa!Qb],Qr,Q) &
         append([Ta!Tb],Tr,T) &
         combine(Q,T,Q1,T1) ) ;
         (
         ¬ Tb = [] &
/*         write(['**********************************Yureeka',Tb]) &
         append([Qa],Qr,Q) &
         append([Ta],Tr,T) &
/*         write(['**Q',[Q]]) &
/*         write(['**T',[T]]) &
/*         write(['**********************Yureeka',Tb]) &
         combine(Q,T,Q3,T3) &
/*         write(['**Q3',Q3]) &
/*         write(['**T3',T3]) &
         append(Q3,Qb,Q1) &
         append(T3,Tb,T1) ).
/*         write(['**T1',T1]) ).



combine([Qc1![or![[condition![[Qc2!R]]]]]],
        [Con1![connector![[conditional![[Con2!N]]]]]],
        [Newq],[Newt]) <-
                    a_condition(Con1) &
                    a_condition(Con2) &
                    specific_conditionals(Qc1,Att1,T1) &
```

```
                        specific_conditionals(Qc2,Att2,T2) &
                        get_attribute(Qatt,Tatt,Att1,Att2) &
                        ¬ Tatt = [] &
                        equivalance(T1,T2,Newcond) &
                        append([Newcond],Qatt,Qn) &
                        append(Qn,R,Qqn) &
                        append([condition],[Qqn],Newq) &
                        append([conditional],[Tatt],Tn) &
                        append(Tn,N,Ttn) &
                        append([conditional],[Ttn],Newt) .

combine([Qc1![or![[condition![[Qc2!R]]]]]],
        [Con1![connector![[conditional![[Con2!N]]]]]],
        [Newq],[Newt]) <-
                        a_condition(Con1) &
                        a_condition(Con2) &
                        specific_conditionals(Qc1,Att1,T1) &
                        specific_conditionals(Qc2,Att2,T2) &
                        get_attribute(Qatt,Tatt,Att1,Att2) &
                        Tatt = [] &
                        equivalance(T1,T2,Newcond) &
                        append([Newcond],Qatt,Qn) &
                        append(Qn,R,Qqn) &
                        append([condition],[Qqn],Newq) &

                        append([conditional],Tatt,Tn) &
                        append(Tn,N,Ttn) &
                        append([conditional],[Ttn],Newt) .

combine([[condition![[R![T!Tr]]]]!Qs],
        [[conditional![[N![NT!Ntt]]]]!D],
        [[condition  ![[R!Ns]]]],
        [[conditional![[N!Ny]]]]) <-
                              member([conditional!Qa],[N![NT]]) &
/* write('here6') &
/* write(['R',R]) &
/* write(['R!T',[R!T]]) &
                         get_condition([N![NT]],[N2],[[R]![T]],[R2]) &
/* write(['R2',R2]) &
/* write(['Qs',Qs]) &
                              combine([R2!Qs],[N2!D],S,Y) &
                              append(S,Tr,Ns) &
                              append(Y,Ntt,Ny) .
/* write(['S',S]) &
/* write(['Y',Y]) .


combine([[condition![[Qc2![R1!R2]]]]![Co!Qs]],
        [[conditional![[Tn![N1!N2]]]]![connector!D]],
        [[condition![[Qc2![R1!R2]]]]![Co![[condition!
        [[Qc2!Rt]]]]]],
        [[conditional![[Tn![N1!N2]]]]![connector!
        [[conditional![[Tn!Dt]]]]]  ) <-
         (  (R2 = [] & Rt = Qs & Dt = D ) ;
           (¬R2 = [] & append([R1],Qs,Rt)  & append([N1],D,Dt) ) ) .
/*        write('here2') &
/*        write(['here2R1',R1]) &
/*        write(['here2R2',R2]) .

combine([[condition![[Qc2![R1!R2]]]]!Qs],
        [[conditional![[Tn![N1!N2]]]]!D],
        [[condition![[Qc2![R1!R2]]]]![or![[condition!
        [[Qc2!Rt]]]]]],
```

```
        [[conditional![[Tn![N1!N2]]]]![connector!
        [[conditional![[Tn!Dt]]]]]  ) <-
         (  (R2 = [] & Rt = Qs & Dt = D ) ;
           (¬R2 = [] & append([R1],Qs,Rt)  & append([N1],D,Dt) ) ) .
/*        write('here3') .

combine([[condition![Qc2![Con![R]]]]!Qs],
        [[conditional![T1![Co![T2]]]]!Tr],
        [Qc2![Con!A]],
        [T1![connector!B]]  ) <-
/*        write('here4') &
        combine([R!Qs],[T2!Tr],A,B) .

get_attribute([],[],any,any) <- /.
get_attribute([Att],attribute,Att,Att) <- /.
get_attribute([Att],attribute,Att,any) <- /.
get_attribute([Att],attribute,any,Att) <- /.
get_attribute([Att],attribute,Any,Att).
get_attribute([Att],attribute,Att,Any).

a_condition(conditional).
a_condition(intra_entity).

specific_conditionals(Q,A,T) <- intra_entity(Q,A,T).
specific_conditionals(Q,any,Q) <- conditionals(Q).

get_condition([[conditional!R]],[[conditional!R]],Q,Q) <- /.
get_condition([H!T],R,[Qh!Qt],Q) <- get_condition(T,R,Qt,Q).
```

```
/* remove all filler words
remove_filler([],[],[],[]) <- /.
remove_filler([filler!X],[H1!T],Tt,Tw) <-
                                    / &
                                    remove_filler(X,T,Tt,Tw).
remove_filler([H1!T1],[H2!T2],[H1!T3],[H2!T4]) <-
                                    remove_filler(T1,T2,T3,T4).
```

```
  filler("*").
  filler(filler).
  filler(please).
  filler(in).
  filler(a).
  filler(which).
  filler(whose).
  filler(have).
  filler(has).
  filler(how).
  filler(much).
  filler(get).
  filler(do).
  filler(every).
  filler(all).
  filler(information).
  filler(about).
  filler(their).
  filler(as).
/*  filler(by).
  filler(the).
  filler(of).
  filler(who).
  filler(are).
  filler(but).
  filler(than).
  filler(they).
  filler(whom).
  filler(if).
  filler(them).
  filler(to).
  filler(where).
  filler(print).
  filler(number).
  filler(retrieve).
  filler(retreive).
  filler(fetch).
  filler(is).
  filler(specify).
  filler(any).
  filler(on).
  filler(with).

filler(".").
filler(",").
filler("!").
filler("?").
filler(":").
filler(";").
```

```
rules(A,M) <- axn(A,B,C,D,E) &
              C =..["<-",A1,B1] & / &
              findout(B1,M).
rules(A,[]).

findout(A,F) <- A=..["&",M1,M2] & / &
                findout(M1,F1) &
                findout(M2,F2) &
                new_append(F1,F2,F).
findout(A,F) <- A=..[";",M1,M2] & / &
                findout(M1,F1) &
                findout(M2,F2) &
                new_append(F1,F2,F).
findout(A,[M1]) <- A =..[M1!M2].
```

```
/*********************************************************/
/*                                                       */
/*  find clause names asociated with predicate Pn        */
/*                                                       */
/*********************************************************/
find(Pn,L) <- axn(Pn,A,R,CN,I) &
              / &
              find_others(Pn,[CN],L).
find(Pn,L) <- nl & prst('No such rule has been loaded') & nl & nl &
              findall([],L) & prst('Loaded clauses are') & nl & nl &
              printlist(L) & nl.

/*********************************************************/
/*                                                       */
/* find all clause names for loaded predicates           */
/*                                                       */
/*********************************************************/
findall(L,R) <- axn(Pn,A,B,CN,I) &
                ¬ member(CN,L) & / &
                append(L,[CN],L2) &
                findall(L2,R).
findall(L,L) .

/*********************************************************/
/*                                                       */
/* find other clause names for loaded predicate Pn       */
/* '                                                     */
/*********************************************************/
find_others(Pn,L,Lf) <- axn(Pn,A,R,CN,I) &
                        ¬ member(CN,L) & / &
                        append(L,[CN],L2) &
                        find_others(Pn,L2,Lf).
find_others(Pn,L,L).

/*********************************************************/
/*                                                       */
/* find file names for all files filetype prolog         */
/*                                                       */
/*********************************************************/
extfind(Pn,L) <- system('listfile * prolog ( exec',cms) &
                 dcio(in,input,file,cms,exec,a) &
                 readat(J1,in,1)  & write(J1) & fail &
                 readat(J2,in)  &
                 readat(J3,in)  &
                 readat(J4,in)  &
                 readat(File,in)  &
                 readat(J5,in) &
                 readat(J6,in) &
                 write(File) .

/*********************************************************/
/*                                                       */
/* find clause names where predicate defined in          */
/* more than one clause                                  */
/*                                                       */
/*********************************************************/
check_for_split_clause_definition(P) <- axn(P,A,R,CN,I) &
                                ¬ P = op &
                                find_others(P,[CN],Clause_list) &
                                ¬ Clause_list = [H] &
                                nl &
                prst('Predicate definition found in more than one clause') &
                                nl & nl &
```

```
                        writes(P) & nl & nl &
                        prst('In clauses ') &
                        printlist(Clause_list) & nl.

/*******************************************************/
/*                                                     */
/* find clause names asociated with predicate Pn */
/*                                                     */
/*******************************************************/
find_similar(Pred,Clause) <- axn(Pn,A,R,CN,I) &
                             atom(Pn) &
                             st_to_at(Sp,Pn) &
                             st_to_at(Sd,Pred) &
                             substring(Sp,Sd,Q,N) &
                             / &
                             find_others_similar(Pred,[Pn],Clause).

/*******************************************************/
/*                                                     */
/* find other clause names for loaded predicate Pn  */
/*                                                     */
/*******************************************************/
find_others_similar(Pred,L,Lf) <- axn(Pn,A,R,CN,I) &
                                  atom(Pn) &
                                  st_to_at(Sp,Pn) &
                                  st_to_at(Sd,Pred) &
                                  substring(Sp,Sd,Q,N) &
                                  ¬ member(Pn,L) & / &
                                 append(L,[Pn],L2) &
                                 find_others_similar(Pred,L2,Lf).
find_others_similar(Pn,L,L).

output_pred <- dcio(out,output,file,pred,list,a,v,80) &
               o_pred([debug,built],out) &
               dcio(close,out).
o_pred(Cl,Out) <-  axn(Pnnn,Ann,Rnn,CN,Inn) &
                   ¬ member(CN,Cl) &
                   new_append([CN],Cl,Cl2) &  / &
                   o_clause(CN,[],Out) &
                   o_pred(Cl2,Out).
o_clause(CN,Pl,Out) <- axn(Pn,A,R,CN,I) &
                       ¬ member(Pn,Pl) & / &
                   find(Pn,L) &
                   writes(Pn,Out) & nl(Out) &
                   tab(15,Out) & writes(L,Out) &
                   write(Pn) &
                   nl(Out) & tab(15,Out) &
                   rules(Pn,M) & writes(M,Out) &
                   nl(Out) &
                   new_append([Pn],Pl,Pl2) & / &
                   o_clause(CN,Pl2,Out).
o_clause(CN,Pl,Out).
```

```
/* get a query from a file test for bye to terminate */
fget_query(Query,Alist,File) <-
                              readlinef(Str,File) &
                              caseshift(Str,Query) & / & nl &
                              check_asked_condition(clear) &
                              maintain_query(Query) &
                              ( Query='bye' ) ;
                              ( process_query(Query,R1,Rep3) &
                                out_query(R1,Rep3,Alist) ) & / .

gf <- nl & dcio(in,input,file,simple,data,a) &
      repeat &
          fget_query(Query,Alist,in) &
          ( ¬ Query = 'bye' &
            display & nl &
            prst('HIT RETURN TO CONTINUE') & nl &
            readline(Return) &
            run_query(Alist,R) &
            fail ) ;
          Query = 'bye' & / &
          dcio(in,close).
```

```
get_query(Query,Rep3,Alist) <-
               system('clear') &
               prst('Enter Your Query or Command ( DOMAIN, LOAD, NEW,') &
               prst(' SAVE, STOP )') & nl & nl &
               readline(Str) &
               caseshift(Str,Query) & / &
               check_asked_condition(clear) &
               maintain_query(Query) &
               (  Query='stop' ) ;
               (  process_query(Query,R1,Rep3) &
                  out_query(R1,Rep3,Alist) ) &
               /.

g <-  repeat &
      get_query(Query,Ret,Alist) &
          ( ¬ Query = 'stop' &
            display & nl &
/*            prst('CHECK THE QUERY THEN HIT RETURN TO CONFIRM') & nl &
/*            prst('TYPE Q TO QUIT') & nl &
/*            readline(Return) &
/*            (¬ Return = 'q' &  ¬ Return = 'Q' ) &
            run_query(Alist,R) &
            explain(Ret,Alist,R) &
            fail                ) ;
         Query = 'stop' &  /.
```

```
/*****************************************************************/
/*                                                               */
/* identify conditional parts of a query                         */
/*  Qlst    - Query list                                         */
/*  Trlst   - translation list                                   */
/*  NewQ    - new query list                                     */
/*  NewTr   - new translation list                               */
/*                                                               */
/*****************************************************************/
identify_clauses(Q,T,Ret) <-
                              find_conditions(Q,T,Tempq,Tempt) &
/*write(['*Q ',Tempq]) &
/*write(['*T ',Tempt]) &
                              find_combinations(Tempq,Tempt,Intq,Intt) &
/*write(['*Q2 ',Intq]) &
/*write(['*T2 ',Intt]) &
                                 ( ( member(aggregate,Intt) ) |
                                 ( ¬ member(inter_entity,Intt) &
                                   ¬ member(intra_entity,Intt) &
                                   ¬ member(conditional,Intt) &
                                   ¬ member(stored,Intt) ) ) &
              formalise_conditions(Intq,Intt,Fntq,Fntt) &
/*  Intq = Fntq & Intt = Fntt &
                              find_aggregates(Fntq,Fntt,Nnq,Nnt) &
/*write(['*Q ',Nnq]) &
/*write(['*T ',Nnt]) &
                    unite_conditions(Nnq,Nnt,Ret) .
/*write(['*R ',Nnq]) .

find_conditions(Qin,Tin,Qout,Tout) <-
/* ( member(conditional,Tin) |
/*                                      member(inter_entity,Tin) ;
/*                                       member(intra_entity,Tin) ) &
                                  find_condition(Qin,Tin,Qtout,Ttout) &
                                  ( ¬ Qin = Qtout | ¬ Tin = Ttout ) &
/*                                / &
                                  find_conditions(Qtout,Ttout,Qout,Tout).
find_conditions(Qin,Tin,Qin,Tin).

find_condition([Hq!Tq],[Ht!Tt],Nq,[[conditional![[Ht!Nt2]]]!Nt1]) <-
                                 type_of_condition([Ht!Tc]) &
/*    add                           match(Tc,Tt,Tq,Remt,Remq,Q) &
                                 match(Tc,Tt,Tq,Nt1,Nq1,Q) &
                                 find_condition(Q,Tc,Nq2,Nt2) &
/*                                 find_condition(Remq,Remt,Nq1,Nt1) &
                                 append([[condition![[Hq!Nq2]]]],Nq1,Nq).
find_condition([Hq!Tq],[Ht!Tt],[Hq!Qr],[Ht!Tr]) <-
                                 find_condition(Tq,Tt,Qr,Tr).
find_condition([],[],[],[]).


find_aggregates(Qin,Tin,Qout,Tout) <- member(aggregate,Tin) & / &
                                      find_aggregate(Qin,Tin,Qout,Tout) &
                                      ¬ member(aggregate,Tout) .
find_aggregates(Qin,Tin,Qin,Tin).

find_aggregate([Hq!Tq],[Ht!Tt],Nq,[[aggregate![[Ht!Nt2]]]!Nt1]) <-
                                 type_of_aggregate([Ht!Tc]) &
                                 match(Tc,Tt,Tq,Remt,Remq,Q) &
                                 find_aggregate(Q,Tc,Nq2,Nt2) &
                                 find_aggregate(Remq,Remt,Nq1,Nt1) &
                                 append([[aggregate![[Hq!Nq2]]]],Nq1,Nq).
find_aggregate([Hq!Tq],[Ht!Tt],[Hq!Qr],[Ht!Tr]) <-
```

```
                                    find_aggregate(Tq,Tt,Qr,Tr).
find_aggregate([],[],[],[]).


find_combinations(A,B,C,D) <- find_combination(A,B,C1,D1) &
                              ( ¬ A = C1 | ¬ B = D1 ) &
                              / &
                              find_combinations(C1,D1,C,D).
find_combinations(A,B,A,B).

find_combination([Hq!Tq],[Ht!Tt],Nq,[[combine![[Ht!Tc]]]!Nt1]) <-
                                    type_of_combination([Ht!Tc]) &
                                    match(Tc,Tt,Tq,Remt,Remq,Q) &
                                    find_combination(Remq,Remt,Nq1,Nt1) &
                                    append([[combine![[Hq!Q]]]],Nq1,Nq).
find_combination([Hq!Tq],[Ht!Tt],[Hq!Qr],[Ht!Tr]) <-
                                    find_combination(Tq,Tt,Qr,Tr).
find_combination([],[],[],[]).


match([H!T1],[H!T2],[Hq!Tq],Remt,Remq,[Hq!Q]) <-
                                  match(T1,T2,Tq,Remt,Remq,Q).
match([],Remt,Remq,Remt,Remq,[]).


/* type_of_combination([conditional,connector,conditional]).
type_of_combination([[conditional!R],connector,stored]).
type_of_combination([[conditional!R],stored]).
type_of_combination([[combine!R],connector,stored]).
type_of_combination([[combine!R],stored]).
type_of_combination([conditional,connector,[conditional!R]]).
type_of_combination([conditional,connector,[combine!R]]).
type_of_combination([intra_entity,connector,[conditional!R]]).


/* type_of_condition([attribute,conditional,stored]).
/* type_of_condition([conditional,entity,stored]).
/* type_of_condition([conditional,entity,[conditional!R]]).
type_of_condition([entity,inter_entity]).
type_of_condition([stored,inter_entity]).
type_of_condition([intra_entity,stored]).
type_of_condition([conditional,stored]).
type_of_condition([connector,stored]).
type_of_condition([attribute,stored]).
type_of_condition([entity,stored]).
/*pe_of_condition([stored,entity]).
type_of_condition([inter_entity,stored]).
type_of_condition([attribute,[conditional!R]]).
type_of_condition([inter_entity,[conditional!R]]).
type_of_condition([entity,[conditional!R]]).
type_of_condition([intra_entity,[conditional!R]]).
type_of_condition([conditional,[conditional!R]]).


type_of_aggregate([aggregate,entity]).
type_of_aggregate([aggregate,attribute]).
type_of_aggregate([aggregate,[conditional!R]]).
type_of_aggregate([entity,aggregate,attribute]).
type_of_aggregate([entity,inter_entity,aggregate]).
```

```
intro <- system('clear') & system('type intro info',cms) & g &
fin.
```

```
/* is defines what a term "is" in the database world

  op(is,rl,50).

  X is stored       <-   quoted(X,*) & / .
  X is intra_entity <-  intra_entity(X,*,*).
/*  X is synonym      <-   synonym(X,Xb).
  X is stored       <-   stored_in_database(X) ; numb(X) .
  X is connector    <-   connectors(X).
  X is conditional  <-   conditionals(X).
/*    X is relation <-         relation(X,*,*).
  X is entity       <-   entity(X).
  X is dimension    <-   dimension(X).
  X is aggregate    <-   aggregates(X,*,*).
  X is attribute    <-   attributes(X,*,*).
  X is inter_entity <-   connect( X, *, *, *, *, *) ;
                         connect( *, *, *, *, *, X).
  X is stored       <-   attributes(Z,one_of,L) & st_to_at(Xs,X) &
                          member(Xs,L).
  X is filler      <-   filler(X).

/* The following items have been commented out to reduce duplication
/*   X is plural <- plural(X,Singular).
/*   X is synonym <- synonym(X,*) | synonym(*,X).
/*   X is entity <- entity( X).
/*   X is subset <- subset(*,X).
/*   X is superset <- subset(X,*).
```

```
output_relation(X,N,Out) <- relation(X,Key,Attributes) &
                            no_composite(Attributes) &
                            st_to_at(St,X) &
                            prst(St,Out) &
                            prst('(',Out) &
                            out_att_list(Attributes,N,Out) &
                            prst(' ) ',Out).

no_composite([H!T]) <- ¬ composite(H,Any) & / &
                       no_composite(T).
no_composite([]).


out_att_list([H![]],N,Out) <- make_variable(H,S,JJ,N,Out) & / &
                              prst(S,Out) .
out_att_list([H!T],N,Out) <- make_variable(H,S,JJ,N,Out) & / &
                             prst(S,Out) &
                             prst(',',Out) &
                             out_att_list(T,N,Out).
out_att_list([],N,Out).

output_subset([X,Y],N,Out) <- output_relation(X,N,Out) &
                              nl(Out) &
                              prst('         &',Out) &
                              nl(Out) &
                              output_relation(Y,N,Out).

output_inter_entity([X,I,Y],Ni,No,Out) <- relation(X,Keyx,Ax) &
                                          relation(Y,Keyy,Ay) &
/*                                          output_relation(X,Out) &
/*                                          prst(' &',Out) &
/*                                          nl(Out) &
                                          st_to_at(S,I) &
                                          prst(S,Out) &
                                          prst('(',Out) &
                                          same_key(Keyx,Keyy,Ni,No) &
                                          out_att_list(Keyx,Ni,Out) &
                                          prst(', ',Out) &
                                          out_att_list(Keyy,No,Out) &
                                          prst(' )',Out) .
/*                                          nl(Out) &
/*                                          prst('         &',Out) &
/*                                          nl(Out) &
/*                                          output_relation(Y,No,Out).


output_condition([[X,Y],Con,Object],Ni,No,Out) <-
                                          prst(' ( ',Out) &
                                          sub_cond(Object,Ni,No,Out) &
                                          output_relation(X,Ni,Out) &
                                          prst(' &',Out) &
                                          nl(Out) &
                                          crite(Con,Y,Out) &
                                          prst('(',Out) &
                                          out_att_list([Y],Ni,Out) &
                                          prst(' ,',Out) &
                                          out_cond(Object,No,Out) &
/*st('-------',Out) &
                                          prst(' ) ',Out) .

output_connector(or,Out) <- nl(Out) & prst('  |  ',Out) & nl(Out).
output_connector(and,Out) <- nl(Out) & prst(' &  ',Out) & nl(Out).
output_connector(not,Out) <- nl(Out) & prst(' & ¬ ',Out) & nl(Out).
```

```
out_cond(Object,N,Out) <- atom(Object) & / &
                          st_to_at(Sob,Object) &
                          writes(Sob,Out) &
                          prst(' ) ',Out) .
out_cond(Object,N,Out) <- numb(Object) & / &
                          writes(Object,Out) &
                          prst(' ) ',Out) .
out_cond([sub![[E,A]![T]]],N,Out) <-
                                    out_att_list([A],N,Out) &
                                    prst(' ) ',Out) .
sub_cond([sub![[E,A]![T]]],Ni,Noo,Out) <- increm(Ni,No) & / &
                                    nl(Out) &
                                    write_tail([],T,No,Noo,Out) &
                                    prst(' &',Out) &  nl(Out).
sub_cond(A,Ni,Ni,Out).

same_key(Keyx,Keyy,Ni,Ni) <- ¬ Keyx = Keyy & /.
same_key(Key,Key,Ni,No) <- increm(Ni,No).

crite("/=",Att,Out) <- prst('ne',Out).
crite("<",Att,Out)  <- prst('lt',Out).
crite("<=",Att,Out) <- prst('le',Out).
crite("=",Att,Out)  <- prst('eq',Out).
crite(">",Att,Out)  <- prst('gt',Out).
crite(">=",Att,Out) <- prst('ge',Out).
crite(rule,Att,Out) <- attributes(Att,composite,Rule) &
                       st_to_at(S,Rule) & prst(S,Out).
crite(after,Att,Out) <- prst('ge',Out).
```

```
/* succeeds if A is a member if the list B
member(A,[A!T]).
member(A,[H!T]) <- member(A,T).
```

```
/*****************************************************************/
/*                                                               */
/*  Erase user workspace to clear for a new user                 */
/*  Erase current user assumptions                               */
/*  Erase current workspace                                      */
/*                                                               */
/*****************************************************************/
new_user <- current_user(User) &
          ax(A,B,User,M) &
          delax(B) &
          prst('Removed ') &
          writes(A) & nl &
          fail.
new_user <- ax(A,B,[],M) &
          delax(B) &
          prst('Removed ') &
          writes(A) & nl &
          fail.
new_user().

/*****************************************************************/
/*                                                               */
/*  Erase user query assumptions                                 */
/*                                                               */
/*****************************************************************/
new_query <- axn(previously_asked,B,D,[],M) &
             delax(D) &
             fail.
new_query().
/*****************************************************************/
/*                                                               */
/*  Load in the profile for a new user                           */
/*                                                               */
/*****************************************************************/
loaduser <- nl & nl & prst('ENTER USER PROFILE ID') & nl &
            readat(User) &
            addax(current_user(User) ) &
            reconsult(User).

/*****************************************************************/
/*                                                               */
/*  Save the perceptions made by a user                          */
/*  Check if a user profile exists                               */
/*  If file does not exist write direct to file                  */
/*      "User prolog a"                                          */
/*  Otherwise append to existing file "User prolog a"            */
/*                                                               */
/*****************************************************************/
saveuser <- nl & nl & prst('ENTER ID FOR SAVED PROFILE') & nl & nl &
            readline(User) &
            system('set emsg off',cms) &
            stconc('listfile ',User,S1) &
            stconc(S1,' prolog a',S2) &
            system(S2,cms,Ret) &
            out(User,Ret) &
            system('set emsg on',cms).

/*****************************************************************/
/*                                                               */
/*  file does not exist write direct to file                     */
/*      "User prolog a"                                          */
/*                                                               */
/*****************************************************************/
```

```
out(User,28) <- dcio(out,output,file,User,prolog,a,f,80) &
                output_profile(out) &
                dcio(out,close).
/*************************************************************************/
/*                                                                       */
/*  file does exist write to file "junkzzq prolog a"                     */
/*  Then append to existing file "User prolog a"                         */
/*                                                                       */
/*************************************************************************/
out(User,0) <- dcio(out,output,file,junkzzq,prolog,a,f,80) & nl &
               prst('Appending to old user profile') & nl &
               output_profile(out) &
               dcio(out,close) &
               stconc('copyfile ',User,S1) &
               stconc(S1,' prolog a junkzzq prolog a ',S2) &
               stconc(S2,User,S3) &
               stconc(S3,' prolog a ( replace',S4) &
               system(S4,cms) &
               system('erase junkzzq prolog a',cms).

/*************************************************************************/
/*                                                                       */
/*  Output current user profile file specified Out                       */
/*                                                                       */
/*************************************************************************/
output_profile(Out) <- ax(A,B,[],M) &
                       write(B,Out) &
                       fail.
output_profile(Out).
```

```
ccondition_output([[connector,[O]]!T],[O!R]) <- condition_output(T,R).
ccondition_output([H!T],[and!R]) <- ¬ H = [connector![K]] &
                                    condition_output([H!T],R).
ccondition_output([H!T],[or!R]) <- ¬ H = [connector![K]] &
                                   condition_output([H!T],R).
ccondition_output([],[]).

condition_output([[subset![[A![B]]]]!T],D) <-
                                           / & ccondition_output(T,R) &
                                           append([B,is,a,A],R,D).
condition_output([[inter_entity![[A![B![C]]]]]!T],D) <- / &
                                           ccondition_output(T,R) &
                                           append([A,B,C],R,D).
condition_output([[condition![[A![B![C]]]]]!T],E) <-
                                            sub_condition(C,Co) & / &
                                            ccondition_output(T,R) &
                                            split_object(A,A1,A2) &
                                            ifrule(B,Bo) &
                                         append([the,A1,"'s",A2,is,Bo,Co],R,E).
condition_output([[connector,[not]]!T],[not!R]) <- condition_output(T,R).
condition_output([],[]).

split_object([An![A2]],An,A2).

sub_condition([sub![N![C]]],[N!["subject to"!Co]]) <- / &
                                          condition_output(C,Co).
sub_condition(C,C).

ifrule(rule,"according to a rule") <- /.
ifrule(X,X).
```

```
increm(Si,So) <- st_to_at(Si,Ai) &
                 Ao := Ai + 1 &
                 st_to_at(So,Ao).


out_query(F,S,Alist) <- dcio(out,output,file,qu_out,prolog,a,v,80) &
                        remove_previous_queries &
                        nl(out) & nl(out) &
                        write_head(F,Ret,Alist,out) &
                        nl(out) &
                        remove_default(S,Snd) &
/* prst('11111') & write([F,Snd]) &
                        prst(' <- ',out) &
                        nl(out) & / &
                        write_tail(Ret,Snd,'0',M,out) &
                        prst(' . ',out) &  nl(out) &
                        nl(out) & nl(out) &
                        dcio(out,close) &
                   system('copyfile qu_out prolog a col prolog a(append',
                                                                    cms).
out_query(F,S,Alist) <- error('wrtiting out query pred').

remove_default([[default,[X]]!T],Tr) <- / &
                                        remove_default(T,Tr).
remove_default([[object,X]!T],Tr) <- / &
                                     remove_default(T,Tr).
remove_default([H!T],[H!Tr]) <- remove_default(T,Tr).
remove_default([],[]).

remove_previous_queries <- ax(pquery(X),M,N) &
                           delax(pquery(X)) &
                           fail.
remove_previous_queries.

make_variable(Att,'',Att1,N,Out) <- atom(Att) &
                                    composite(Att,Att1) &
                                    out_att_list(Att1,N,Out) & /.

make_variable(Att,Str2,[Att],N,Out) <- atom(Att) &
                           st_to_at(Str,Att) &
                           st_to_li(Str,Lis) &
                           capital(Lis,Clis) &
                           st_to_li(Str1,Clis) &
                           stconc(Str1,N,Str2).
capital([H!T],[Ch!T]) <- upshift(H,Ch).


write_head(F,R,Alist,Out) <- prst('pred_query([',Out) &
                             out_ret_list(F,R,Alist,'0',Out) &
                             prst('])  ',Out).



out_ret_list([H],R1,A1,N,Out) <- / & out_ret_list(H,R1,A1,N,Out) .
out_ret_list([He![H![Ca![]]]],[He],R,N,Out) <- make_variable(H,S,R,N,Out)
                                               & / &
                                               prst(S,Out) .
out_ret_list([He![H![Ca!T]]],Ret,A1,N,Out) <- make_variable(H,S,A1,N,Out)
                                              & / &
                                              prst(S,Out) &
                                              prst(',',Out) &
                                          out_ret_list(T,Tret,Ta,N,Out) &
                                            append(A1,Ta,A1) &
```

```
                                                new_append([He],Tret,Ret).
out_ret_list([H!T],R1,A1,N,Out) <- / & out_ret_list(H,R2,A2,N,Out) &
                                    prst(',',Out) &
                                    out_ret_list(T,R3,A3,N,Out) &
                                    append(A2,A3,A1) &
                                    new_append(R2,R3,R1).
out_ret_list([],[],[],N,Out).

write_tail([X![]],[],N,M,Out) <- output_relation(X,N,Out) & / .
write_tail([X!T],[],N,M,Out) <- output_relation(X,N,Out) &
                                 prst(' &',Out) & nl(Out) & / &
                                 write_tail(T,[],N,M,Out).
write_tail([],[],N,N,Out) <- / .
write_tail([],S,N,M,Out) <-  ¬ S = [] &
                             nl(Out) & / &
                             writtail(S,N,M,Out).
write_tail(X,S,N,M,Out) <-  ¬ S = [] & write_tail(X,[],N,M,Out) &
                            prst(' &',Out) & nl(Out) & / &
                            writtail(S,N,M,Out).

writtail([X![]],N,M,Out) <-
                           wrt_tail(X,N,M,Out) & / & nl(Out).
writtail([X![[connector,[not]]![[inter_entity,Y]!T]]],N,Mo,Out) <-
                 wrt_tail(X,N,M,Out) & / &
                 prst(' &',Out) & nl(Out) &
          writtail([[connector,[not]]![[inter_entity,Y]!T]],M,Mo,Out).

writtail([X![[connector,[Y]]!T]],N,Mo,Out) <-
                                  wrt_tail(X,N,M,Out) & / &
                                  writtail([[connector,[Y]]!T],M,Mo,Out).
writtail([[connector,[not]]![[inter_entity,X]!T]],N,M,Out) <-
                                 negate_inter_ent(X,Z) & / &
                                 writtail([[inter_entity,Z]!T],N,M,Out).
writtail([[connector,[X]]!T],N,Mo,Out) <- / &
                                      wrt_tail([connector,[X]],N,M,Out) &
                                      writtail(T,M,Mo,Out).
writtail([X!T],N,Mo,Out) <- wrt_tail(X,N,M,Out) & / &
                            prst(' &',Out) & nl(Out) &
                            writtail(T,M,Mo,Out).
writtail([],N,M,Out).

wrt_tail([[X]],N,M,Out) <- wrt_tail([X],N,M,Out).
wrt_tail([relation,[X]],N,N,Out) <- output_relation(X,N,Out).
wrt_tail([subset,[X,Y]],N,N,Out) <- output_subset([X,Y],N,Out).
wrt_tail([inter_entity,[X,I,Y]],Ni,No,Out) <-
                                   output_inter_entity([X,I,Y],Ni,No,Out).
wrt_tail([condition,[[X,Y],Con,Obj]],N,M,Out) <-
                                 output_condition([[X,Y],Con,Obj],N,M,Out).
wrt_tail([connector,[X]],N,N,Out) <- output_connector(X,Out).
wrt_tail([default,X],N,N,Out).

putwrittail(F,Out) <- writes(F,Out) .
negate_inter_ent([E1,I1,E2],[E1,I2,E2]) <- st_to_at(S1,I1) &
                                           stconc('not_',S1,S2) &
                                           st_to_at(S2,I2) .
```

```
/* Swaps words for singular of plural
swap_plurals([HA!TA],[HB!TB]) <- plural(HB,HA) &
                                 swap_plurals(TA,TB).
swap_plurals([HA!TA],[HA!TB]) <- swap_plurals(TA,TB).
swap_plurals([],[]).
```

```
/*****************************************************************/
/*                                                               */
/*  prints out a list as a string of elements separated          */
/*  by a space. output directed to the console                   */
/*                                                               */
/*****************************************************************/
printlist([]) <- /.
printlist([H!T]) <- atom(H) & / &
                    st_to_at(Hs,H) &
                    prst(Hs) &
                    prst(' ') &
                    printlist(T) & /.
printlist([H!T]) <- numb(H) & / &
                    prst(H) &
                    prst(' ') &
                    printlist(T) & /.
printlist([H!T]) <- stringp(H) & / &
                    prst(H) &
                    prst(' ') &
                    printlist(T) & /.
printlist([H!T]) <- printlist(H) &
                    prst(' ') &
                    printlist(T) & /.

/*****************************************************************/
/*                                                               */
/*  prints out a list to the channel Out - Out should            */
/*  be specified in a DCIO statement                             */
/*                                                               */
/*****************************************************************/
printlistf([],Out) <- /.
printlistf([H!T],Out) <- atom(H) & / &
                         st_to_at(Hs,H) &
                         prst(Hs,Out) &
                         prst(' ',Out) &
                         printlist(T,Out).
printlistf([H!T],Out) <- prst(H,Out) &
                         prst(' ',Out) &
                         printlist(T,Out).
```

```
process_query(Query,R12,Ret2) <- function_check(Query) &
                                 analyse(Query,Parse,Listq) &
                                 consult_user_for_unknown(Listq,Parse) &
                                 identify_clauses(Listq,Parse,Ret) &
/* write([' * ',Ret,' * ',' *  proc***']) & readli(HH) &
                                 re_order_not(Ret,Ret2) &
                                 compose(Ret2,R12,Rep3) & / .
process_query(Query,R12,Ret2) <- ¬ Query = '' &
                                 ¬ a_function(Query) &
                                system('clear',cms) &
                                nl &
                                prst('I AM UNABLE TO PARSE THIS QUERY') &
                                nl & nl &
                                prst('PLEASE RESPECIFY YOUR REQUEST') &
                                nl & nl &
                                prst('HIT RETURN TO CONTINUE') &
                                nl & readli(Return) &
                                fail.

re_order_not(Ret,Ret2) <- locate([[connector,[not]],[subset,S],
                                  [inter_entity,I]],Ret,B,A) &
                          append(B,[[subset,S]],R1) &
                          append(R1,[[connector,[not]]],R2) &
                          append(R2,[[inter_entity,I]],R3) &
                          append(R3,A,Ret3) &
                          re_order_not(Ret3,Ret2).
re_order_not(Ret,Ret).

a_function('domain') .
a_function('save') .
a_function('new') .
a_function('load') .
a_function('DOMAIN').
a_function('SAVE') .
a_function('NEW') .
a_function('LOAD') .
```

```
<- pragma(list,1).

  op(gtn,rl,50).
  op(ltn,rl,50).
  op(gen,rl,50).
  op(len,rl,50).
  op(eqn,rl,50).

<- reconsult(append).
<- reconsult(csql).

/*****************************************************************/
/*                                                               */
/*  Equals constraint eqn                                        */
/*                                                               */
/*****************************************************************/

eqn(X,Y) <- var(X) &
            ¬ var(Y) &
            X = Y .
eqn(X,Y) <- ¬ var(X) &
            X = constraint(N) &
            ¬ var(Y) &
            change_eqn(N,Y).
eqn(X,Y) <- ¬ var(X) &
            ¬ X = constraint(N) &
            ¬ var(Y) &
            eq(X,Y).

change_eqn(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   test(Y,G1,N1) &
                   test(Y,L2,N2) &
                   label(constraint(N,eq,Y,null,null) ).

/*****************************************************************/
/*                                                               */
/*  Greater than constraint gtn                                  */
/*                                                               */
/*****************************************************************/

gtn(X,Y) <- ¬ var(X) &
            X = constraint(N) &
            ¬ var(Y) &
            change_gtn(N,Y).
gtn(X,Y) <- var(X) &
            ¬ var(Y) &
            new_con(N) &
            X = constraint(N) &
            label(constraint(N,gt,Y,null,null)).


change_gtn(N,Y) <- constraint(N,eq,N1,L2,N2) &
                   gt(N1,Y).
change_gtn(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   ¬ G1 = eq &
                   test(Y,lt,N2) &
                   ¬ test(Y,G1,N1).
change_gtn(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   test(Y,lt,N2) &
                   test(Y,G1,N1) &
                   label(constraint(N,gt,Y,L2,N2) ).

/*****************************************************************/
```

```
/*                                                                    */
/*  Greater than or equal to constraint gen                           */
/*                                                                    */
/********************************************************************/

gen(X,Y) <- ¬ var(X) &
            X = constraint(N) &
            ¬ var(Y) &
            change_gen(N,Y).
gen(X,Y) <- var(X) &
            ¬ var(Y) &
            new_con(N) &
            X = constraint(N) &
            label(constraint(N,ge,Y,null,null)).


change_gen(N,Y) <- constraint(N,eq,N1,L2,N2) &
                   gt(N1,Y).
change_gen(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   ¬ G1 = eq &
                   test(Y,L2,N2) &
                   ¬ test(Y,G1,N1).
change_gen(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   test(Y,L2,N2) &
                   test(Y,G1,N1) &
                   check_addax(N,ge,Y,L2,N2).


/********************************************************************/
/*                                                                  */
/*  Less than constraint ltn                                        */
/*                                                                  */
/********************************************************************/

ltn(X,Y) <- ¬ var(X) &
            X = constraint(N) &
            ¬ var(Y) &
            change_ltn(N,Y).
ltn(X,Y) <- var(X) &
            ¬ var(Y) &
            new_con(N) &
            X = constraint(N) &
            label(constraint(N,null,null,lt,Y)).

change_ltn(N,Y) <- constraint(N,eq,N1,L2,N2) &
                   lt(N1,Y).
change_ltn(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   ¬ G1 = eq &
                   test(Y,gt,N1) &
                   ¬ test(Y,L2,N2).
change_ltn(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   test(Y,gt,N1) &
                   test(Y,L2,N2) &
                   label(constraint(N,G1,N1,lt,Y) ).

/********************************************************************/
/*                                                                  */
/*  Less than or equal to constraint len                            */
/*                                                                  */
/********************************************************************/

len(X,Y) <- ¬ var(X) &
            X = constraint(N) &
```

```
            ¬ var(Y) &
            change_len(N,Y).
len(X,Y) <- var(X) &
            ¬ var(Y) &
            new_con(N) &
            X = constraint(N) &
            label(constraint(N,null,null,le,Y)).

change_len(N,Y) <- constraint(N,eq,N1,L2,N2) &
                   le(N1,Y).
change_len(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   ¬ G1 = eq &
                   test(Y,G1,N1) &
                   ¬ test(Y,L2,N2).
change_len(N,Y) <- constraint(N,G1,N1,L2,N2) &
                   test(Y,G1,N1) &
                   test(Y,L2,N2) &
                   check_addax(N,G1,N1,le,Y).

/*****************************************************************************

check_addax(N,ge,N1,le,N2) <- eq(N1,N2) & / &
                              label(constraint(N,eq,N1,null,null) ).
check_addax(N,G1,N1,L2,N2) <- label(constraint(N,G1,N1,L2,N2) ).

/*****************************************************************************

constraint(N,G1,N1,L2,N2) <- query_1(constraint(N,G1,N1,L2,N2)).

/*****************************************************************************

new_con(N1) <- query_1(num_con(N)) &
               N1 := N + 1 &
               label(num_con(N1)).
new_con(0) <-  label(num_con(0)).

/*****************************************************************************

test(Y,Sym,null).
test(Y,lt,Y1) <- ( ¬ Y1 == null ) & lt(Y,Y1).
test(Y,gt,Y1) <- ( ¬ Y1 == null ) & gt(Y,Y1).
test(Y,le,Y1) <- ( ¬ Y1 == null ) & le(Y,Y1).
test(Y,ge,Y1) <- ( ¬ Y1 == null ) & ge(Y,Y1).

/*****************************************************************************

sc <- query_1( constraint(A,B,C,D,E),N ) &
      write([A,B,C,D,E]) & fail.
sc.

/*****************************************************************************

sc(constraint(A)) <- query_1( constraint(A,B,C,D,E)) &
                     write([A,B,C,D,E]) & fail.
sc(X).

/*****************************************************************************

coutput(constraint(N),C) <-  query_1( constraint(N,eq,C,D,E)).
coutput(constraint(N),Ou) <-  query_1( constraint(N,B,C,D,E)) &
                              cwrt(B,C,Ou1) &
                              cwrt(D,E,Ou2) &
                              append(Ou1,Ou2,Ou).
```

```
/*****************************************************************************

cwrt(null,E,[]).
cwrt(lt,E,[' < ',E]).
cwrt(le,E,[' <= ',E]).
cwrt(gt,E,[' > ',E]) .
cwrt(ge,E,[' >= ',E]).

a(1).
a(2).
a(3).
a(4).
a(5).
a(6).
a(7).
a(8).
```

```
uppercase_data([H1!T1],[H2!T2],[H3!T3]) <- type_of_unknown(H2) &
                                           ¬ quoted(H1,Sh1) & / &
                                           st_to_at(Sh1,H1) &
                                           caseshift(Sh3,Sh1) &
                                           st_to_at(Sh3,H3) &
                                           uppercase_data(T1,T2,T3) .
uppercase_data([H1!T1],[H2!T2],[H1!T3]) <- uppercase_data(T1,T2,T3) .
uppercase_data([],[],[]) .


identify_quoted_terms([Hi!Ti],[Ho!To]) <- quoted(Hi,Unqhi) & / &
                                          st_to_at(Unqhi,Ho) &
                                          identify_quoted_terms(Ti,To).
identify_quoted_terms([Hi!Ti],[Hi!To]) <- identify_quoted_terms(Ti,To).
identify_quoted_terms([],[]).

quoted(X,R) <- st_to_at(S,X) &
               substring(S,'''',0,1) &
               stlen(S,L) &
               K := L - 1 &
               J := K - 1 &
               substring(S,'''',K,1) &
               substring(S,R,1,J) .
```

```
rationalise(Lin,Lout) <- find_subset([subset![[A![B]]]],Lin,B1,Af1) &
                         find_subset([subset![[A![B]]]],Af1,B2,Af2) &
                         ¬ member([condition!R],B2) &
                         append(B1,B2,B3) &
                         append(B3,[[subset![[A![B]]]]],B4) &
                         append(B4,Af2,Lout1) & / &
                         rationalise(Lout1,Lout).
rationalise(Lin,Lin).

find_subset(Ha,[Ha!Rlin],[],Rlin).
find_subset(Ha,[H!T],[H!Before],After) <- find_object(Ha,T,Before,After).
```

```
<- reconsult(str_list).
/*****************************************************************/
/*                                                               */
/* read a line test take off leading and trailing spaces         */
/* test if a continuation line or last line                      */
/*                                                               */
/*****************************************************************/
readlinef(A,In) <- readli(C,In) &
                   strip_spaces(C,C2) &
                   lastlinef(C2,A,In).
/*****************************************************************/
/*                                                               */
/* if a continuation line '-' read another line                  */
/*                                                               */
/*****************************************************************/
lastlinef(C,A,In) <- stlen(C,L) &
                     D := L - 1 &
                     substring(C,'-',D,1) &
                     / &
                     readlinef(Ab,In) &
                     substring(C,Cb,0,D) &
                     stconc(Cb,Ab,A).
lastlinef(C,C,In).

/*****************************************************************/
/*                                                               */
/* read a line test if lastline                                  */
/*                                                               */
/*****************************************************************/
readline(A) <- readli(C) &
               strip_spaces(C,C2) &
               lastline(C2,A).
/*****************************************************************/
/*                                                               */
/* if a continuation line '-' read another line                  */
/*                                                               */
/*****************************************************************/
lastline(C,A) <-  stlen(C,L) &
                  D := L - 1 &
                  substring(C,'-',D,1) &
                  / &
                  readline(Ab) &
                  substring(C,Cb,0,D) &
                  stconc(Cb,Ab,A).
lastline(C,C).
```

```
/* reverse the order of items in a list
reverse([],[]).
reverse([H!T],X) <- reverse(T,Z) & append(Z,[H],X).
```

```
/*****************************************************************
/*
/* obtain_retrieval_list(Rlout,Retqueryin)
/*
/*****************************************************************
obtain_retrieval_list(Rlout,Rlin,Rep) <- label(retri) &
                                         obtain_retrie_list(Rlout,Rlin,Rep).
obtain_retrie_list(Rl,Rl,Rl) <- member([mystic!Wl],Rl) & / .
obtain_retrie_list(Rlout,Rlin,Rep) <- member([object,Entity],Rlin) &
/*write(['rl pos 1',Entity]) &
                                         obtain_objects(Rlin,Rlout,Rep) &
/*write(['rl pos 2',Rlout]) &
                                            ¬ Rlout = [] & /.
obtain_retrie_list([relation![[Rlout]]],Rlin,Rlin2) <-
                                      obtain_context(Rlin,Rlout) &
                                      strip_object_default(Rlin,Rlin1) &
                                      add_subset(Rlout,Rlin1,Rlin2).

obtain_objects(Rlin,[Object!Out],Rep) <-
                                       find_object([object!R],Rlin,Before,After) &
                                       / &
                                      append(Before,After,Aff) &
                                       form_object([object!R],Object,Rlin,Rep1) &
                                       ¬ Rep1 = [error] & / &
                                       obtain_objects(Aff,Out,Rep2) &
                                       append(Rep1,Rep2,Rep).
obtain_objects(Rlin,[[relation!R]!Out],Rep) <-
                                     find_object([relation!R],Rlin,Before,After) &
                                      / &
                                     append(Before,After,Aff) &
                                     obtain_objects(Aff,Out,Rep).
obtain_objects([[default!R]],[],[]) <- /.
obtain_objects([H!Rest],Out,[H!Rep]) <- obtain_objects(Rest,Out,Rep).
obtain_objects([],[],[]).

form_object([object![[unknown![Attr]]]],[object![Entity_At]],R,Rep) <-
                                        / &
                                        get_entity(Entity_At,Attr,R,Rep).
form_object([object!A],[object!A],R,[]).

find_object(Ha,[Ha!Rlin],[],Rlin) <- /.
find_object(Ha,[H!T],[H!Before],After) <- find_object(Ha,T,Before,After).

get_entity(Object,Attr,R,Rep) <- obtain_context(R,Entity) &
                                 get_context(Attr,Entity,Object) &
                                 match_isa(Entity,Object,Rep) .
get_entity(Object,Attr,R,[error]) <- obtain_context(R,Entity) &
                                     get_context(Attr,Entity,Object) &
                                     ¬ match_isa(Entity,Object,Rep) &
                                     fail(retri).

obtain_context([[subset![[Entity![[Subent]]]]!R],Entity).
obtain_context([[subset![[Entity![[Subent]]]]!R],Subent).
obtain_context([[inter_entity![[Entity![[In![[Subent]]]]]]!R],Entity).
obtain_context([[default![[Entity]]]!R],Entity).
obtain_context([[relation![[Entity]]]!R],Entity).
obtain_context([H!T],Entity) <- obtain_context(T,Entity).


strip_object_default([[default!Entity]],[]).
strip_object_default([[object!Entity]!T1],T2) <- / &
                                      strip_object_default(T1,T2).
strip_object_default([[relation!Entity]!T1],T2) <- / &
```

```
                                    strip_object_default(T1,T2).
strip_object_default([H1!T1],[H1!T2]) <- ¬ H1 = [default!Entity]  &
                                    ¬ H1 = [object!Entity]  &
                                    strip_object_default(T1,T2).
strip_object_default([],[]).


add_subset(Rlout,Rlin1,Rlin2) <- subsets(X,Rlout) &
                                 add_subset(X,Rlin1,R13) &
                                 append([[subset,[Rlout,X]]],R13,Rlin2).
add_subset(Rlout,Rlin,Rlin) <- ¬ subsets(X,Rlout).
```

```
run_query(A1,s) <- reconsult(qu_out) &
                   count(Number) &
                   ne(Number,0) & / &
                   system('clear') &
                   prst('THERE ARE ') & num_writes(Number) &
                   prst(' TUPLES SATISFYING THE RETRIEVAL CONSTRAINTS') &
                   nl & nl &
                   prst('HIT RETURN TO VIEW OR Q TO QUIT') & nl &
                   readli(Junk) &
                   ¬ ( Junk = 'q' ; Junk = 'Q' )  & / &
                   output_retrieval(A1,Reply) &
                   Reply = '' &
                   system('clear') &
                   prst('NO MORE TUPLE TO BE RETRIEVED') & nl &
                   nl & nl &
                   prst('HIT RETURN TO CONTINUE ') & nl &
                   readli(Junk2).
run_query(A1,f) <- system('clear') &
                   prst('NO SATISFYING TUPLES  ') & nl &
                   nl & nl &
                   prst('HIT RETURN') & nl &
                   readli(Junk).

output_retrieval(A1,Reply) <- pred_query(O1) &
                              system('clear') &
                              out_ret(A1,O1) &
                              nl & nl &
                         prst('HIT RETURN TO CONTINUE - Q TO QUIT') & nl &
                         readli(Reply) &
                         ( Reply = 'q' ; Reply = 'Q' )  & / .
output_retrieval(A1,'').
out_ret([Ha!Ta],[Ho!To]) <- writes(Ha) & tab(30) & stprst(Ho) &
                            out_dim(Ha) & nl & out_ret(Ta,To).
out_ret([],[]) <- nl & tab(15) &
                  prst('_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _') &
                  nl.

out_dim(A) <- dimension( A, S ) &
              prst('  ') & prst(S) & / .
out_dim(A).


stprst(A) <- stringp(A) & prst(A) & /.
stprst(A) <- writes(A) & /.

/*************************************

count(X) <- ¬ ( delax(counter(N)) & fail ) &
                addax(counter(0)) & fail.
count(M) <- pred_query(P) &
            counter(N) &
            M := N + 1 &
            delax(counter(N)) &
            addax(counter(M)) & eq(M,10).
count(Y) <- counter(Y).

num_writes(Number) <- lt(Number,10) & writes(Number) & / .
num_writes(Number) <- ge(Number,10) & writes(Number) & prst(' or more ').
```

```
strip_leading_terms([[connector,[or]]!T1],T2) <-
                                        strip_leading_terms(T1,T2) &
                                        / .
strip_leading_terms([[connector,[and]]!T1],T2) <-
                                        strip_leading_terms(T1,T2) &
                                        / .
strip_leading_terms([H1!T1],[H1!T1]).
strip_leading_terms([],[]).
```

```
copy_till_dash([H!T],To,[H!Ty]) <- copy_till_dash(T,To,Ty).
copy_till_dash([],[],[]) <- fail.

/*******************************************************/
/*                                                     */
/* convert a list of characters from upper case        */
/* to lower case unless letters inside quotes          */
/*                                                     */
/*******************************************************/
shift(["'"!T],["'"!Ty]) <- copy_till_dash(T,To,Tz) & / &
                           shift(To,Tw) &
                           append(Tz,Tw,Ty) & / .
shift([H!T],[Hy!Ty]) <- upshift(Hy,H) & / &
                        shift(T,Ty) & / .
shift([H!T],[H!Ty]) <- shift(T,Ty) & /.
shift([],[]).

/*******************************************************/
/*                                                     */
/* convert a list of characters from lower case        */
/* to upper case unless letters inside quotes          */
/*                                                     */
/*******************************************************/
ushift(["'"!T],["'"!Ty]) <- copy_till_dash(T,To,Tz) & / &
                            ushift(To,Tw) &
                            append(Tz,Tw,Ty) & / .
ushift([H!T],[Hy!Ty]) <- upshift(H,Hy) & / &
                         ushift(T,Ty) & / .
ushift([H!T],[H!Ty]) <- ushift(T,Ty) & /.
ushift([],[]).


/*******************************************************/
/*                                                     */
/* convert a upper case string into lower case         */
/*                                                     */
/*******************************************************/
caseshift(U,L) <- var(L) & / &
                  st_to_li(U,Ul) &
                  shift(Ul,Ll) & / &
                  st_to_li(L,Ll).
/*******************************************************/
/*                                                     */
/* convert a lower case string into upper case         */
/*                                                     */
/*******************************************************/
caseshift(U,L) <- var(U) &
                  st_to_li(L,Ll) &
                  ushift(Ll,Ul) & / &
                  st_to_li(U,Ul).

/*******************************************************/
/*                                                     */
/* substitute defined word seperater characaters       */
/* for spaces                                          */
/*                                                     */
/*******************************************************/
space_separate(Xi,X) <- st_to_li(Xi,Xil) &
                        sub_sep(Xil,Xl) &
                        st_to_li(X,Xl).

sub_sep([Hxi!Txi],Newlist) <- separate(Hxi,Hnew) & / &
                              sub_sep(Txi,Tx) &
```

```
                                append(Hnew,Tx,Newlist).
sub_sep([Hxi!Txi],[Hxi!Tx]) <- sub_sep(Txi,Tx).
sub_sep([],[]).

/*separate(".",[" ","."," "]).
separate(",",[" ",","," "]).
separate("!",[" ","!"," "]).
separate("?",[" ","?"," "]).
separate(":",[" ",":"," "]).
separate(";",[" ",";"," "]).
```

```
/*****************************************************/
/*                                                   */
/*   Swaps words for alternative synonyms            */
/*                                                   */
/*****************************************************/
swap_synonyms([HA!TA],[HA!TB]) <- HA is XX & swap_synonyms(TA,TB).
swap_synonyms([HA!TA],[HB!TB]) <- synonyms(HA,HB) &
                                  swap_synonyms(TA,TB).
swap_synonyms([HA!TA],[HA!TB]) <-
                                   ¬ HA is XX &
                                   swap_synonyms(TA,TB).
swap_synonyms([],[]).

synonyms(A,B) <- synonym(A,B).
synonyms(A,B) <- ¬ var(A) & synonym(A,C) & synonyms(C,B) & ¬ var(C) &
                 ¬ var(B) & ¬ A = B.

  synonym(equal,"=").
  synonym(greater,">").
/*  synonym(is,"=").
  synonym("<>","/=").
  synonym(same,"=").
  synonym(less_than,"<").
  synonym(lessthan,"<").
  synonym(otherthan,not).
  synonym(under,"<").
  synonym(over,">").

/* synonym(outside term, dbase term).
/* synonym(their,person).
 synonym(department,department_id).
 synonym(attending,attend).
 synonym(from,resident_at).
 synonym(lives,resident).
 synonym('don''t',not).
 synonym(taught,lectured).
 synonym(teaches,lecture).
 synonym(dept,department).
 synonym(depts,department).
 synonym(tutees,student).
/* synonym(tutors,lecturer).
/* synonym(tutee,student).
 synonym(people,person).
 synonym(given,taught).
 synonym(personal_tutor,personal_tutor_to).
 synonym(teach,teacher_for).
 synonym(teach,lecturer_to).
 synonym(give,teacher_for).
 synonym(take,teach).
/* synonym(person,student).
 synonym(who,name).
 synonym(graduate,grad).
 synonym(graduates,grads).
 synonym(secretary,sec).
 synonym(earn,paid).
```

```
match(Value,Attribute) <- attributes(Attribute,one_of,List) &
                          member(Value,List).

match(Value,Attribute) <- int(Value) &
                          attributes(Attribute,integer,[L![U]]) &
                          ( le(Value,U) ; U = max ) &
                          ( ge(Value,L) ; L = min ).

match(Value,Attribute) <- int(Value) &
                          attributes(Attribute,real,[Db![Da]]) &
                          L := len(Value) &
                          ge(Db,L)  .

match(Value,Attribute) <- floatp(Value) &
                          attributes(Attribute,real,[Db![Da]]) &
                          A := abs(Value) &
                          F := A - 0.49999999 &
                          fl_to_int(F,I) &
                          Li := len(I) &
                          La := len(A) &
                          le(Li,Db) &
                          N := ( La - Li ) - 1 &
                          ge(Da,N).

match(Value,Attribute) <-
                          T := len(Value) &
                         ( attributes(Attribute,fchar,N) &
                          T = N ) ;
                          ( attributes(Attribute,vchar,N) &
                          ge(N,T) ).

match(Value,Attribute) <- attributes(Attribute,reference,Relation) &
                          relation(Relation,[Key],List) &
                          match(Value,Key).

match(Value,Attribute) <- attributes(Attribute,composite,Rule).
```

```
/* translate terms into database objects
translate([H!T],[Hx!Tx]) <- H is Hx & translate(T,Tx).
translate([H!T],[unknown!Tx]) <- ( ¬ H is Hx ) &
                                 ( ¬ plural(Hx,H) ) &
                                 translate(T,Tx).

translate([],[]).
```

```
/*****************************************************************/
/*                                                               */
/*  Unite conditionals with there objects and rules              */
/*                                                               */
/*****************************************************************/
unite_conditions(Ha,Hb,Ret) <- / &
                                    unit_co(Ha,Hb,unknown,Ret).


/*****************************************************************/
/*                                                               */
/*  Unite conditionals with there objects and rules              */
/*  Maintains the current entity context                         */
/*                                                               */
/*****************************************************************/
/*******************************************/
/*                                         */
/*  New entity is referred to so the       */
/*  context has been changed try to        */
/*  form condition using the new context   */
/*  but if not then use the old one if     */
/*  it was known                           */
/*                                         */
/*******************************************/
unit_co(A,B,Context,Ret) <- ¬ A = [] & ¬ B = [] &
                            ¬ member([inter_entity!E],A) &
                            ¬ member([condition!F],A) &
                            make_retrieval_list(A,B,Context,Ret).

unit_co([Ha!Ta],[entity!Tb],Context,Ret) <- / &
                                      new_context(Context,Ha,Newcon,Ta) &
                                      unit_co(Ta,Tb,Newcon,Ret1) &
                                    append([[relation![[Ha]]]],Ret1,Ret).

/*********************************************/
/*                                           */
/*  inter-entity is referred to so the       */
/*  context has been changed try to          */
/*  form condition using the new context     */
/*                                           */
/*********************************************/
unit_co([Ha!Ta],[inter_entity!Tb],Context,Ret) <- / &
                                    get_inter_entity(Ha,Con,Tcon) &
                                    unit_co([Ta],[Tb],Con,Ret).

/*********************************************/
/*                                           */
/*  a condition has been encountered so      */
/*  make a formal condition then parse       */
/*  the rest                                 */
/*                                           */
/*********************************************/
unit_co([Ha!Ta],[[conditional!R]!Tb],Context,Ret) <- / &
                    form_condition(Ha,[conditional!R],Context,Con,Ret1) &
                    new_context(Context,Con,Newcon,Ta) &
                    unit_co(Ta,Tb,Newcon,Ret2) &
                    append(Ret1,Ret2,Ret).

/*********************************************/
/*                                           */
/*  a connector has been encountered so      */
/*  record connection type then parse        */
/*  the rest                                 */
```

```
/*                                                  */
/****************************************************/
unit_co([Oper!Ta],[connector!Tb],Context,Ret) <- / &
                        unit_co(Ta,Tb,Context,Ret1) &
                        append([[connector![[Oper]]]],Ret1,Ret).

/****************************************************/
/*                                                  */
/*  a non condition object has been                 */
/*  encountered so record its presence              */
/*  then parse the rest                             */
/*                                                  */
/****************************************************/
unit_co([Ha!Ta],[Hb!Tb],Context,Ret) <-
                          get_context(Ha,Context,Newcon) &
                          match_isa(Context,Newcon,Rep) & / &
                          new_context(Context,Newcon,Ncon,Ta) &
                          unit_co(Ta,Tb,Ncon,Ret1) &
                          append([[object![[unknown![Ha]]]]],Ret1,Ret2) &
                          append(Rep,Ret2,Ret).
unit_co([Ha!Ta],[Hb!Tb],Context,Ret) <-
                          ¬ Ha = [Any!Other] & / &
                          unit_co(Ta,Tb,Context,Ret1) &
                          append([[object![[unknown![Ha]]]]],Ret1,Ret).
unit_co([Ha!Ta],[Hb!Tb],Context,Ret) <- / &
                                      unit_co(Ta,Tb,Context,Ret1) &
                                      append([[mystic![Ha]]],Ret1,Ret) .

/****************************************************/
/*                                                  */
/*  a null list indicates the end of the            */
/*  parse                                           */
/*                                                  */
/****************************************************/
unit_co([],[],[Context![Any]],[[default![[Context]]]]) <- /.
unit_co([],[],Context,[[default![[Context]]]]) <- /.

/*******************************************************************/
/*                                                                 */
/*  Form the actual conditions on which the retrieval              */
/*  is based                                                       */
/*                                                                 */
/*******************************************************************/
/****************************************************/
/*                                                  */
/*  doble bracketed condition remove one            */
/*  set of brackets and find formal                 */
/*  conditions                                      */
/*                                                  */
/****************************************************/
form_condition([[condition!A]],[[conditional!B]],Con,Non,Ret) <- / &
            form_condition([condition!A],[conditional!B],Con,Non,Ret).

/****************************************************/
/*                                                  */
/*  An entity has been encountered change           */
/*  context then continue find formal               */
/*  conditions                                      */
/*                                                  */
/****************************************************/
/* form_condition([condition![[E![A]]]],
  /*          [conditional![[entity![inter_entity]]]],Con,E,Ret) <- / &
    /*          reverse_get_inter_entity(A,Newcon,Tcon,Z) &
```

```
      /*         prst('**CONDITION - ')  &
        /*         writes([Con,E,Newcon]) & nl & nl &
          /*    append([[inter_entity![[Newcon![A![Con]]]]]],Ret1,Ret) .

form_condition([condition![[E!A]]],
               [conditional![[entity!B]]],Con,E,Ret) <- / &
                  form_condition(A,B,E,F,Ret).

/*********************************************/
/*                                           */
/*  An Attribute has been encountered        */
/*  change context then continue to find     */
/*  formal conditions                        */
/*                                           */
/*********************************************/
form_condition([condition![[Attr!A]]],
               [conditional![[attribute!B]]],Con,Newcon,Ret) <-
/*write('1') &
/*write(['Attr',Attr,'   **  ',A]) &
            get_context(Attr,Con,Newcon) &
/*write('2') &
/*write(['Con',Con,' Newcon ',Newcon,' Rep ',Rep]) &
            match_isa(Con,Newcon,Rep) &  / &
/*write('3') &
            form_condition(A,B,Newcon,F,Ret1) &
/*write(['Ret1',Ret1,'   **  ',F]) &
             ( ¬ F = Newcon &
               find_entity(Econ,Con) &
               append([[object![[Econ![Attr]]]]],Ret1,Ret2) &
               append(Rep,Ret2,Ret) ) |
             (  F = Newcon &
                append(Rep,Ret1,Ret) ).
form_condition([condition![[Attr!A]]],
               [conditional![[attribute!B]]],Con,F,Ret) <- / &
/*               get_context(Attr,Con,Newcon) &
/*write('2qq') &
/*write(['Attr',Attr,'   **  ',A]) &
/*write(['Con',Con,' Newcon ',Newcon,' Rep ',Rep]) &
               form_condition(A,B,Con,F,Ret1) &
               append([[object![[unknown![Attr]]]]],Ret1,Ret) .

/*********************************************/
/*                                           */
/*  A condition separated by a connector     */
/*  so find a formal condition for           */
/*  conditions before and after connector    */
/*                                           */
/*********************************************/
form_condition([[condition!E]![Oper!A]],
               [[conditional!Et]![connector!At]],
               Con,F,Ret) <- / &
        form_condition([[condition!E]],[[conditional!Et]],Con,F1,Ret1) &
        form_condition(A,At,Con,F2,Ret2) &
        append(Ret1,[[connector![[Oper]]]],Ret3) &
        append(Ret3,Ret2,Ret).

/*********************************************/
/*                                           */
/*  A condition made up of the comparator    */
/*  attribute then value output the          */
/*  resulting conditions                     */
/*                                           */
/*********************************************/
```

```
form_condition([condition![[Comp![Att![Value]]]]],
               [conditional![[conditional![attribute![stored]]]]],
               Con,Con,Ret) <- / &
/*              prst('*3CONDITION - ') &
                          out_cond(Att,Con,Ncon,Comp,Value,Ret).
form_condition([condition![[Att![Value]]]],
               [conditional![[attribute![stored]]]],
               Con,Con,Ret) <- / &
/*              prst('*2CONDITION - ') &
                          out_cond(Att,Con,Ncon,Comp,Value,Ret).
form_condition([condition![[Comp![Att!Condit]]]],
               [conditional![[conditional![attribute!Type]]]],
               Con,Ncon,Ret) <- / &
                   get_context(Att,Con,Ncon) &
                   match_isa(Con,Ncon,Rep) &
            form_condition(Condit,Type,Ncon,Non,Ret1) &
            Ret2 = [[condition![[Ncon![Comp![[sub![[Ncon![Ret1]]]]]]]
            ]]]] &
            append(Rep,Ret2,Ret).

/*********************************************/
/*                                           */
/*  A condition made up of the comparator    */
/*  and value output the resulting           */
/*  conditions                               */
/*                                           */
/*********************************************/
form_condition([condition![[Comp![Value]]]],
               [conditional![[conditional![stored]]]],
               Con,Ncon,Ret) <- / &
/*              prst('*1CONDITION - ') &
/*              write([Att,Con,Ncon,Comp,Value,Ret]) &
                          out_cond(Att,Con,Ncon,Comp,Value,Ret).

/*********************************************/
/*                                           */
/*  An inter_entity has been encountered     */
/*  change context then continue find        */
/*  formal conditions                        */
/*                                           */
/*********************************************/
form_condition([condition![[E!A]]],
               [conditional![[inter_entity!B]]],Con,Newcon,Rt) <- / &
               get_inter_entity(E,Newcon,Tcon) &
/*                prst('*1CONDITION - ') &
/*               writes([Con,E,Newcon]) & nl & nl &
                form_condition(A,B,Newcon,F,Ret1) &
                match_isa(Tcon,Con,Rep) &
                append([[inter_entity![[Tcon![E![Newcon]]]]]],Ret1,Ret) &
                append(Rep,Ret,Rt) .

form_condition([condition![[E![A]]]],
               [conditional![[stored![inter_entity]]]],Cn,Non,Rt) <- / &
                  reverse_get_inter_entity(A,Non,Tcon,B) &
/*                prst('*1CONDITION - ') &
/*                writes([Cn,E,Non,B,A]) & nl & nl &
                  form_condition([E],[stored],Non,F,Ret1) &
                  match_isa(Tcon,Cn,Rep) &
                  append([[inter_entity![[Tcon![B![Non]]]]]],Ret1,Ret) &
                  append(Rep,Ret,Rt) .

/*********************************************/
/*                                           */
```

```
/*  An intra_entity condition is followed  */
/*  by another condition, output intra     */
/*  condition then formalise following sub */
/*  conditions                             */
/*                                         */
/*****************************************/
form_condition([condition![[Int_en![[condition!R]]]]],
               [conditional![[intra_entity![[conditional!K]]]]],
               Con,Ncon,Ret) <- / &
                    intra_entity(Int_en,Att,Comp) &
                    get_context(Att,Con,Ncon) &
                    match_isa(Con,Ncon,Rep) &
/*                    prst('CONDITION - ') & writes(Ncon) &
/*                    prst(' ') & writes(Comp) & prst(' ') & nl &
/*                  prst('     sub ') &
            form_condition([condition!R],[conditional!K],Ncon,Non,Ret1) &
            Ret2 = [[condition![[Ncon![Comp![[sub![[Ncon![Ret1]]]
            ]]]]]] &
            append(Rep,Ret2,Ret).

/*********************************************/
/*                                           */
/*  An intra_entity condition is followed    */
/*  by a single value output the formal      */
/*  condition                                */
/*                                           */
/*********************************************/
form_condition([condition![[Int_en![Value]]]],
        [conditional![[intra_entity![stored]]]],Con,Con,Ret) <- / &
                        intra_entity(Int_en,Att,Comp) &
                        out_cond(Att,Con,Ncon,Comp,Value,Ret) .

/*********************************************/
/*                                           */
/*  A lone value using the none context      */
/*  output the formal condition              */
/*                                           */
/*********************************************/
form_condition([Value],[stored],Con,Con,Ret) <- / &
                        out_cond(Att,Con,Ncon,"=",Value,Ret) .
form_condition([condition,[not,Value]],
               [conditional,[connector,stored]],Con,Con,Ret) <- / &
                        out_cond(Att,Con,Ncon,"=",Value,Rt) &
                        append([[connector,[not]]],Rt,Ret) .

/*********************************************/
/*                                           */
/*  Oh dear! something must have gone wrong*/
/*  output the conditional information       */
/*                                           */
/*********************************************/
form_condition([A],[inter_entity],Con,Newcon,Rt) <- / &
              get_inter_entity(A,Newcon,Tcon) &
/*            form_condition(A,B,Newcon,F,Ret1) &
              match_isa(Tcon,Con,Rep) &
              append(Rep,[[inter_entity![[Tcon![A![Newcon]]]]]],Rt).
form_condition(A,B,C,C,[error]) <-
                        prst('CONDITION + ')  &  writes(A) & nl  &
                        prst('TRANSLATION - ') & writes(B) & nl &
                        prst('CONTEXT - ')  &  writes(C) & nl & nl .

/*************************************************************/
```

```
/*                                                                  */
/*  Get context given an attribute name and/or the context          */
/*  match or/find acceptable combinations returning a new           */
/*  context                                                         */
/*                                                                  */
/********************************************************************/
/*************************************************/
/*                                               */
/*  A perfect match the attribute is the         */
/*  attribute in context and the entity          */
/*  context                                      */
/*                                               */
/*************************************************/
get_context(Attr,[Con![Attr]],[Con![Attr]]) .

/*************************************************/
/*                                               */
/*  for all entities which the context is        */
/*  a subset see if attr is an attribute         */
/*  for one of them                              */
/*                                               */
/*************************************************/
get_context(Attr,Con,Newcon) <- subsets(Sup,Con) &
                                relation(Sup,Key,Att_list) &
                                member(Attr,Att_list) &
                                append([Sup],[Attr],Newcon).

/*************************************************/
/*                                               */
/*  The context is a relation which has          */
/*  the attribute attr as an attribute           */
/*                                               */
/*************************************************/
get_context(Attr,Con,Newcon) <- relation(Con,Key,Att_list) &
                                member(Attr,Att_list) &
                                append([Con],[Attr],Newcon).

/*************************************************/
/*                                               */
/*  A subset entity is a relation with           */
/*  the attribute attr as an attribute           */
/*                                               */
/*************************************************/
get_context(Attr,Con,Newcon) <- subsets(Con,Rel) &
                                relation(Rel,Key,Att_list) &
                                ¬ Rel = Con &
                                member(Attr,Att_list) &
                                append([Rel],[Attr],Newcon).

/*************************************************/
/*                                               */
/*  The attribute is an attribute for            */
/*  relation which has not been covered by       */
/*  any of the previous definitions              */
/*                                               */
/*************************************************/
get_context(Attr,Con,Newcon) <- ¬ var(Attr) &
                                relation(Rel,Key,Att_list) &
                                ¬ Rel = Con &
                                ¬ subsets(Rel,Con) &
                                ¬ subsets(Con,Rel) &
                                member(Attr,Att_list) &
                                append([Rel],[Attr],Newcon).
```

```
get_context(Att,[Con![Attr]],Newcon) <- get_context(Att,Con,Newcon) &
                                        ¬ Att = Attr .
out_cond(Att,Con,Ncon,Comp,Value,Ret) <-
                          get_context(Att,Con,Ncon) &
                          match(Value,Att) &
                          match_isa(Con,Ncon,Rep) &
                          get_comparator(Att,Comp,Compout) &
/*  prst('$$$$$CONDITION - ')  &  writes(Ncon) &
/*                            prst('CONDITION - ')  &  writes(Ncon) &
/*                            prst(' ') & writes(Comp) &
/*                            prst(' ') & writes(Value) & nl & nl &
                       Ret1 = [[condition![[Ncon![[Compout![Value]]]]]]] &
                          append(Rep,Ret1,Ret) .
out_cond(Att,Con,Ncon,Comp,Value,[]) <-
                          get_context(Att,Con,Ncon) &
                          match(Value,Att) &
                          match_isa(Con,Ncon,Rep) & / & fail.
out_cond(Att,Con,Ncon,Comp,Value,Ret) <-
  write([Att,Con,Ncon,Comp,Value,Ret]) &
                          get_context(Att,Con,Ncon) &
                          get_context(NAtt,Con,Nncon) &
                          match(Value,NAtt) &
                          same_entity(Ncon,Nncon) &
                          match_isa(Con,Nncon,Rep) &
/*                          prst('CONDITION - ')  &  writes(Ncon) &
/*                          prst(' ') & writes(Comp) & nl &
/*                          prst('     sub ') &
/*                          writes(Ncon) &
/*                          prst(' ') & writes(Nncon) &
/*                          prst(' ') & writes("=") &
/*                          prst(' ') & writes(Value) & nl & nl &
                    get_comparator(NAtt,"=",Compout1) &
/*write('66!!!!!!!!!') &
                    Ret1 = [[condition![[Nncon![Compout1![Value]]]]]] &
/*write('77!!!!!!!!!') &
                    get_comparator(Att,Comp,Compout2) &
/*write('88!!!!!!!!!') &
                Ret2 = [[condition![[Ncon![Compout2![[sub![Ncon![Ret1]]
                ]]]]]]] &
                    append(Rep,Ret2,Ret).

match_isa(Con,Con,[]).
match_isa(Con,Con1,[]) <- ¬ var(Con) & ¬ var(Con1) & Con = unknown.
match_isa(Con,[Con![A]],[]).
match_isa(Con,En,Rep) <-  subsets(En,Con) &
                                Rep = [[subset,[Con,En]]].
match_isa(Con,[En![A]],Rep) <-  subsets(En,Con) &
                                Rep = [[subset,[Con,En]]].
match_isa(En,Con,Rep) <-  subsets(En,Con) &
                                Rep = [[subset,[Con,En]]].
match_isa(En,[Con![A]],Rep) <-  subsets(En,Con) &
                                Rep = [[subset,[Con,En]]].
match_isa([Con![R]],[En![A]],Rep) <-
/* ¬ R = A &
                                    match_isa(Con,[En![A]],Rep) &

same_entity(Ncon,Ncon) <- atom(Ncon).
same_entity([E![R]],[E![P]]) <- ¬ R = P.
same_entity(E,[E![P]]).
same_entity([E![R]],E).

find_entity(E,E) <- atom(E).
```

```
find_entity(E,[E![P]]).

/* get new entity after inter entity reference
get_inter_entity(E,Newcon,Tcon) <- connect(E,Tcon,Newcon,*,*,*).
get_inter_entity(E,Newcon,Tcon) <- connect(*,Newcon,Tcon,*,*,E).
reverse_get_inter_entity(E,Ncon,Tcon,B) <- connect(B,Tcon,Ncon,*,*,E).
reverse_get_inter_entity(E,Ncon,Tcon,B) <- connect(E,Ncon,Tcon,*,*,B).


new_context(Con,Context,Context,[]) <- /.
new_context(Context,Context,Context,A) <- /.
new_context(unknown,Context,Context,A) <- /.
new_context(Context,unknown,Context,A) <- /.
new_context(Con,Context,Context,A).
new_context(Context,Con,Context,A).

get_comparator(Att,Comp,Comp) <- ¬ var(Comp) &
                                 ¬ Comp = "=" & /.
get_comparator(Att,Comp,rule) <- ¬ var(Att) &
                                 attributes(Att,composite,Rule) & / .
get_comparator(Att,Comp,"=").


make_retrieval_list(A,B,Context,Ret) <- get_objects(A,B,Ret) &
                                        ¬ Ret = [] .


get_objects([Ha!Ta],[entity!Tb],C) <- / & get_objects(Ta,Tb,Tc) &
                                      append([[relation![[Ha]]]],Tc,C).
get_objects([Ha!Ta],[attribute!Tb],C) <- / & get_objects(Ta,Tb,Tc) &
                                     append([[object![[unknown![Ha]]]]],Tc,C).
get_objects([Ha!Ta],[Hb!Tb],C) <- get_objects(Ta,Tb,C).
get_objects([],[],[]).
```

```
type_of_unknown(stored).
type_of_unknown(unknown).

unite_unknown([U1![U2!X]],[H1![H2!T]],Tt,Tw)
                        <-
type_of_unknown(U1) &
type_of_unknown(U2) &
                           st_to_at(S1,H1) &
                           st_to_at(S2,H2) &
                           stconc(S1,' ',St) &
                           stconc(St,S2,S3) &
                           st_to_at(S3,Hw) &
                           unite_unknown([unknown!X],[Hw!T],Tt,Tw).
unite_unknown([H1!T1],[H2!T2],[H1!T3],[H2!T4]) <-
                          unite_unknown(T1,T2,T3,T4) .
unite_unknown([],[],[],[]).
```

```
verify_retrieval(In,Out,Rep) <- member([object!A],In) &
                                member([relation!B],In) &
                                reduce(In,Out,Rep).
verify_retrieval(A,A,[]).

reduce(In,Out,Rep) <- locate([[object![O]]],In,B1,A1) &
                      locate([[relation![[R]]]],In,B2,A2) &
                      match_isa(R,O,Rep1) &
                      append(B2,A2,Next) &
                      reduce(Next,Out,Rep2) &
                      append(Rep1,Rep2,Rep).
reduce(In,In,[]).
```