341

TOWARDS A CO-OPERATIVE DATA BASE MANAGEMENT SYSTEM

J C NEVES and M H WILLIAMS

DEPARTMENT OF COMPUTER SCIENCE
HERIOT-WATT UNIVERSITY
EDINBURGH
SCOTLAND

KEYWORDS: Logic programming, Horn clauses, Prolog, Logic data base, Query-by-Example, Incomplete queries, Co-operativeness.

342

Abstract

A desirable feature of any high-level data base query system is that it should be user-friendly. This should extend beyond the provision of a query syntax which is easy to use, to some attempt at intelligent helpfulness or co-operativeness. In particular additional knowledge about the structure of the data in a data base or the incomplete data contained in a query may be used to benefit the user. In this respect, despite its simplicity and ease of use, the data base management language Query-by-Example is relatively inflexible.

This paper looks at several ways in which the co-operativeness of Query-by-Example can be improved. These are concerned with incomplete queries (i.e. queries in which certain information has been omitted), incomplete updates and queries which fail as a result possibly of misconceptions on the part of the user. Consideration is also given to how these are implemented in Prolog.

343

1. Introduction

The application of first order logic and resolution based theorem proving to machine intelligence problems started during the early 1970's. Recently, logic programming has received a considerable boost due to its choice as the basis of the core programming languages for the Japanese Fifth Generation Computer Systems [1].

Prolog [2] is a qualified implementation of Horn clauses which has become important as a vehicle for Artificial Intelligence applications. In particular there is growing interest in its use for data base applications [3]. Since Prolog itself is not very convenient as a query language, various researchers have sought to develop other user interfaces to Prolog data bases. These include natural language interfaces [4] and Query-by-Example [5].

Query-by-Example (QBE) is a non-procedural data base query language developed by Zloof [6] in which queries are expressed by filling in skeleton tables with examples of the result required. In a human factors experiment conducted by Thomas and Gould [7] to determine the ease of use of data base query languages, the advantages of QBE over SQUARE and SEQUEL were clearly demonstrated. In particular they found that subjects using QBE required about one-third the training time, were somewhat faster in expressing queries and were about twice as accurate [7].

In view of this and the similarity between the syntax of Prolog goals and QBE [8], an implementation of QBE interfacing with a logic data base has been realized in Prolog. Details of the implementation are given in [5].

Despite its simplicity and ease of use, QBE is relatively inflexible and makes no attempt at intelligent helpfulness or co-operativeness. This paper consider some ways in which the co-operativeness of QBE can be improved.

2. Incomplete queries

In QBE all queries must be expressed in full in a manner which reflects the way in which the data has been stored in the data base. However, the inexperienced or casual user may have difficulty in remembering the internal structure of the data and the way in which any particular query must be framed in order to reflect this. On the other hand the experienced user may find the process a little clumsy and look for short cuts. The idea of an incomplete query may appeal to either type of user.

In QBE any simple query which involves the join of two relations makes use of a common variable which occurs in one field of each of the two relations.

For example, given the data base in Appendix 1, suppose that the user wants to find the names of all suppliers who supply part number 2. The parts which he/she might enter are underlined "___". The entry might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	

The common variable here which serves to join the two relations is S. Such a variable will be referred to as a link variable, and the fields of the two relations which are linked together (sno of suppliers and sno of supplier_parts) will be referred to as link fields.

In general there is no choice in the pair of link fields which can be used to join two relations together. For example, in the case of the relations suppliers and supplier_parts, the field sno of relation suppliers and field sno of relation supplier_parts are the only possible pair of fields which can be used to join these two relations.

In some cases it may not be possible to join relations directly and a join may only be effected via one or more intermediate relations.

For example, suppose that the user wishes to retrieve the names of all suppliers who supply at least one red part. The essential information in this query is:

suppliers	sno	sname	status	city
		p.N		

parts	pno	pname	colour	weight
			red	

although the complete query is:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	X	

parts	pno	pname	colour	weight
	X		red	

where S and X are both link variables and supplier_parts is an intermediate relation.

Since in general link variables are not an essential part of a query but rather a result of the way in which data are stored in the system, it should be possible for the user to omit link variables from any query (together with any empty intermediate tables which may result). Any query in which one or more of the link variables have been omitted will be referred to as an incomplete query.

However, there is one snag with the omission of the link variables. Consider the query:

suppliers	sno	sname	status	city
		p.N		

supplier_parts	sno	pno	qty
		2	p.X

If this is treated as an incomplete query the system would attempt to link together these two requests. The result might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	p.X

which would be interpreted as "print the name of each supplier who supplies part number 2 and the quantity supplied". On the other hand, the original query is sufficient in its own right being interpreted as "print the names of all suppliers and the quantities of part number 2 as supplied by different suppliers". The latter is a form of OR-query.

In general an incomplete query will have the same form as an OR-query and the system will be unable to distinguish between the two. Thus it must be assumed that an incomplete query will not involve an OR-condition and that the user will indicate when an incomplete query has been issued.

In the next section the underlying data structures and the general approach to implementation of incomplete queries are discussed.

3. Implementation of incomplete queries

If a user wishes to issue an incomplete query, the query is entered in exactly the same way as any other query except that a different key (for example, a special function key in the keyboard) is used to signal the end of the query.

When the system is presented with an incomplete query, it attempts to link together the separate parts of the request. If it succeeds in finding appropriate links, the resulting query will be displayed in full to the user. If this resulting query satisfies the user, he/she indicates acceptance of the query by pressing the key normally used at the end of a complete query; if it is not what the user wants, a different key is used to indicate to the system to continue its search. If no suitable links can be found, the system reports this to the user.

To illustrate this, consider a request for the names of any suppliers who supply widgets and to whom one does not owe money at the present moment. IQ and CQ are used to denote the keys corresponding to Incomplete Query and

Complete Query respectively. The dialogue might be as follows (commentary is in /* ... */ brackets):

suppliers	sno	sname	status	city
		p.N		

parts	pno	pname	colour	weight
		widget		

supplier_balance	sno	amountowed
		X::X=<0

IQ /* signals the end of an incomplete query */

The infix operator "::" is used for syntactic convenience only and is to be read as "such that".

In response to this the system might display:

suppliers	sno	sname	status	city
	A	p.N		

supplier_parts	sno	pno	qty
	A	B	

parts	pno	pname	colour	weight
	B	widget		

supplier_balance	sno	amountowed
	A	X::X=<0

3.1. Formal specification of links

The data structure used to represent the data base relations and the connections between the relations is an undirected graph.

Fig 1 is a diagrammatic representation of the graph representing the links of the data base in Appendix 1. Every data base relation is represented by a vertex or node, called a relation node, and for every two nodes, if the same attribute occurs in both relations, an edge will connect the pair. This edge is labelled with the pair of attribute names from the two relations.

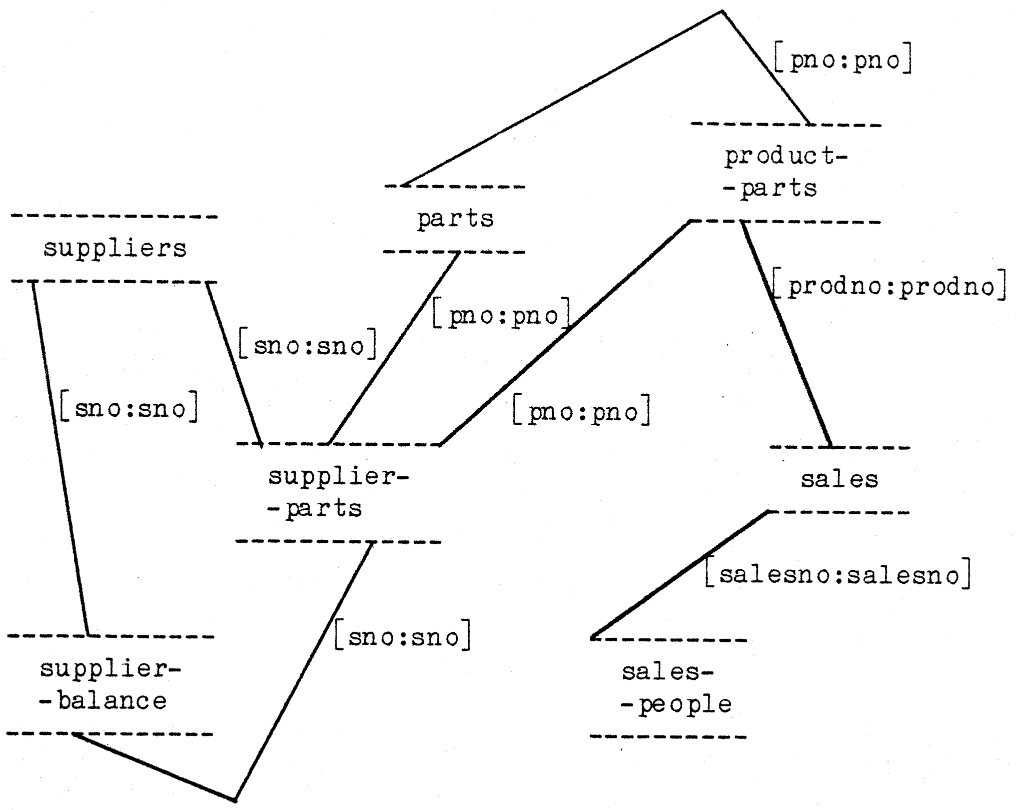

Fig 1 - The graph structure representing the link dictionary for the data base given in Appendix 1.

This information is represented within the query language system by a set of clauses of the form:

```
   link(RELATION NAME 1, RELATION NAME 2,
        [
           ORDERED SEQUENCE OF LINK FIELDS OF RELATION 1:
           ORDERED SEQUENCE OF LINK FIELDS OF RELATION 2
        ]).

   setofnodes(GRAPH NAME,
              [
                 SET OF GRAPH NODES
              ]).
```

In Appendix 2 the link dictionary for the data base in Appendix 1 is given. Also presented is the Prolog program for searching for a path linking any pair of relations in the data base.

The link dictionary described can be accessed by the user through the normal query mechanism, thus enabling the user to examine or update the structure of the data in the data base.

For example, suppose that the user wants to find which relations are linked with which . The entry might be:

```
p.links    |
_______    |
-----------|------------
           |

CQ  /* signals the end of a complete query */
__
```

The system will respond by displaying for each relation R a list of relations linked to R, e.g.

```
links    | supplier_parts
---------|-----------------
         | supplier_balance
         | suppliers
         | parts
         | product_parts

        ...
        ...
        ...
```

3.2. Handling join conditions

In order to handle join conditions the system determines the number N of unlinked components of the request and then seeks to establish the paths linking them together.

For example, suppose that the user wants to find the names of any suppliers to whom no money is owed at the present moment and who supply part number 1023. The entry might be:

suppliers	sno	sname	status	city
	X	p.N		

supplier_balance	sno	amountowed
	X	A::A=<0

product_parts	prodno	pno	noreqd
	1023		

IQ /* signals the end of an incomplete query */

where the user has partially specified the links by using the variable X to link relation suppliers with relation supplier_balance.

Given a query which contains join conditions, the paths linking the different components of the whole request may be established using:

> (i) - relation merging; that is, if two relations x and y which form part of the query are related to each other through the join variables A1, A2, ..., An (n>=1), merge relations x and y by performing joins between relations x and

y. Repeat this operation until no further merges are possible.

(ii) - graph generation; that is, look for paths which connect the remaining unlinked components of the graph (these must involve intermediate relations).

Let join-relation be the relation obtained by the join of relation suppliers with relation supplier_balance. Then the graph for the unlinked components of the initial request is as shown in Fig 2.

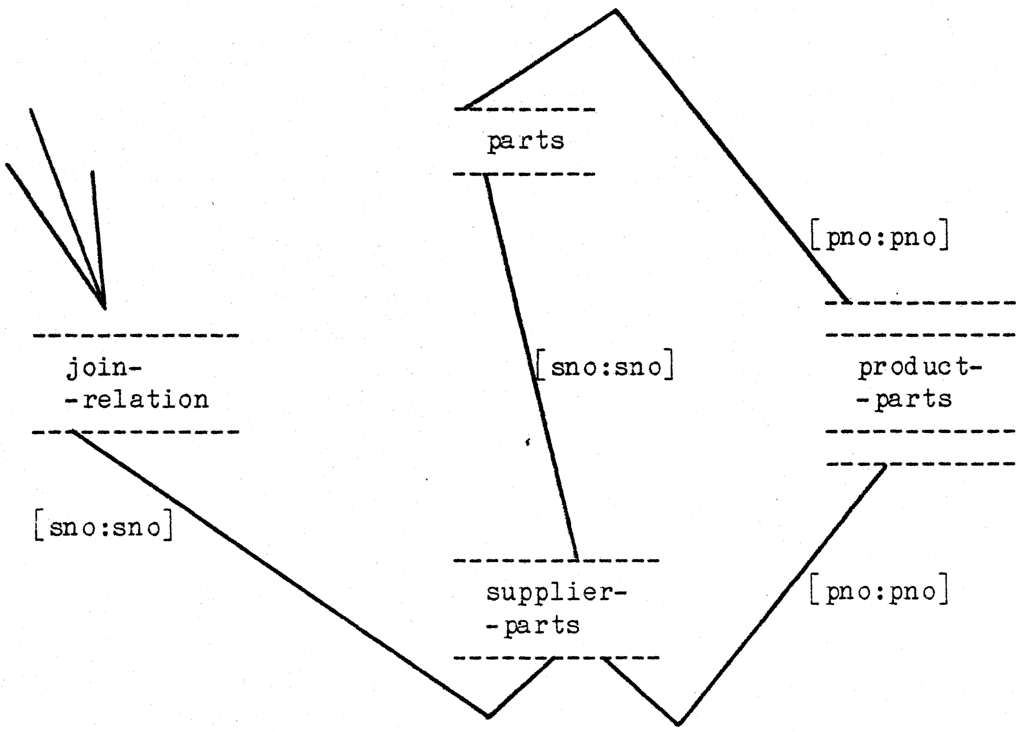

Fig 2 - Graph structure after merging.

The start node is indicated on the graph by an arrow, and double bars have been used to distinguish the final node. As a response, the system might display:

```
suppliers         | sno    sname    status    city
------------------|-----------------------------------
                  | X      p.N

supplier_balance  | sno    amountowed
------------------|---------------------
                  | X      A::A=<0

supplier_parts    | sno    pno    qty
------------------|-------------------
                  | X      Y

product_parts     | prodno    pno    noreqd
------------------|--------------------------
                  | 1023      Y
```

4. Incomplete updates

The ideas outlined in the previous section apply also to update operations.

For example if the user wishes to set the quantity to zero for all suppliers living in London, he/she might enter:

```
supplier_parts     | sno      pno      qty
_______________    |
-------------------|------------------------
u                  |                   0
_                                      _
suppliers          | sno      sname      status      city
__________         |
-------------------|-----------------------------------------
                   |                                 london
                                                     ______
IQ  /* signals the end of an incomplete query */
__
```

to which the system will respond with:

```
supplier_parts   | sno      pno      qty
-----------------|----------------------
u                | Q                 0
```

```
suppliers        | sno      sname      status      city
-----------------|--------------------------------------
                 | Q                               london
```

In addition this link information may also be used in the case of update operations to ensure that when the user attempts to update a value in a link field of some relation, he/she is reminded of the possibility that the corresponding link field in some other relation may need to be updated too. In such a case the system might ask the user whether he/she wishes the same operation to be performed in the corresponding link field in the appropriate relation.

For example, if the user wishes to change the supplier number 13 to 3, the user might enter:

```
suppliers   | sno      sname      status      city
_________   |
------------|-------------------------------------
u           | 3
__          | __
            |
            | 13
              __

CQ  /* signals the end of a complete query */
__
```

The system should then ask the user whether in addition he/she wishes to perform the following updates:

supplier_parts	sno	pno	qty
u	3		
	13		

supplier_balance	sno	amountowed
u	3	
	13	

In such case the user must indicate whether he wishes the additional update operations to be performed or rejected.

5. Queries which fail

In formulating a query a user inevitably makes certain presuppositions about the data present in the data base. These presuppositions are inherent in the information contained in the query and are an indication of what the user believes about the state of the information in the data base.

A data base query can be viewed either as requesting the selection of a subset (termed the response set) from a set of qualified instances in the data base, or as expressing some general belief about the data in the data base. In either case queries presented in QBE are translated into an intermediate meta language before being presented as a conjecture that a resolution-based theorem prover (e.g. Prolog) attempts to prove. This meta language is a graph structure, the nodes of which represent both data base relations and conditions imposed on the relation's attribute(s).

The query graph is divided into connected subgraphs, each of which in itself constitutes a well-formed query in the meta language and is translated into a conjecture that can be presented to the theorem prover to be proved (i.e. each connected subgraph corresponds to a presupposition the user has made about the domain of discourse).

The next section discusses how the presuppositions inherent in these subgraphs can be used to provide a more co-operative response to users for both queries that request the selection of a subset of qualified instances in the data base and YES-NO queries.

5.1. Constructing corrective indirect answers

When dealing with queries requesting the selection of qualified instances in the data base (i.e. with queries defining a property of data base objects) consider the situation where the system fails to prove the conjecture (the initial query returns the empty set as an answer). In this case, on request from the user, the system will try to establish the user's presuppositions by translating each connected subgraph into a conjecture to be proved. This approach ensures that should a presupposition fail, an appropriate corrective indirect answer [9] will be returned to the user.

For example, suppose that the user wishes to retrieve the numbers of all suppliers living in London who supply part number 2. The entry might be:

```
suppliers         | sno    sname    status    city
_________         |
----------------- |------------------------------------------
                  | p.X                       Y
                    ___                       _

supplier_parts    | sno       pno       qty
______________    |
----------------- |-------------------------
                  | X         Z
                    _         _


 |-----------------------|
 |      CONDITIONS       |
 |-----------------------|
 |   Z=2 and Y=london
     ________________


CQ  /* signals the end of a complete query */
__
```

This query is based on the following presuppositions (i.e. the preconditions for the correctness of any direct answer):

(i) There are suppliers.
(ii) There are suppliers who supply parts.
(iii) There are suppliers supplying part number 2.
(iv) There are suppliers living in London.
(v) There are suppliers living in London who supply part number 2.

Should any of these presuppositions fail to be true, the system would, in general, respond with an empty list or "NULL". If, however, this query were addressed to a human being one might expect a more co-operative response which identifies the failing presupposition(s).

A complex query asking for the display of certain data items subject to a variety of retrieval conditions will be decomposed into a number of basic components in the meta language (i.e. connected subgraphs), each of which are acceptable queries in their own right. With each sub-query is associated a subset of the original set of presupposition(s). In the case of the above example, this can be represented diagramatically as:

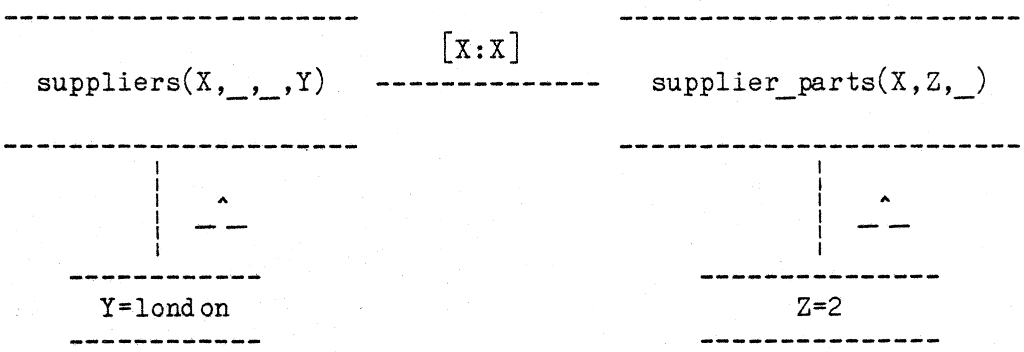

Fig 3 - The complete query.

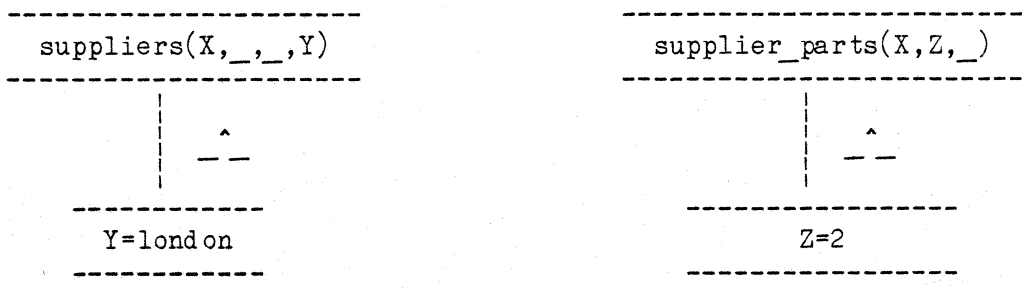

Fig 4 - Two first-level components

```
-----------------------          --------------------------
  suppliers(X,_,_,Y)               supplier_parts(X,Z,_)
-----------------------          --------------------------
```

Fig 5 - Two second-level components

which will be translated by the system to yield:

```
<- suppliers(X, _, _, Y),
   Y=london,
   supplier_parts(X, Z, _),
   Z=2.
```

This clearly consists of two components:

```
<- suppliers(X, _, _, Y),
   Y=london.

<- supplier_parts(X, Z, _),
   Z=2.
```

each of these in turn depend on components:

```
<- suppliers(X, _, _, Y).

<- supplier_parts(X, Z, _).
```

In this case the system's response "NULL" will be produced only in the case where the top level query has failed but all sub-queries have succeeded. Otherwise the message "NULL-LOWER LEVEL QUERY FAILED" will be displayed.

On request the system will attempt to determine the cause of failure. If any sub-query fails and its failure contributes to the failure of the top level query, then:

- the failure of the sub-query will be reported back together with any other sub-query on the same level which contributes to the failure of the top level query,

- any higher level failing sub-queries, not failing due to failure of component sub-queries, will be also reported back.

In the current implementation this is achieved by typing the keyword "WHY".

For example, if the query described above is presented to the system but the system fails to find any supplier living in London, it will respond with:

```
suppliers   | sno      sname      status      city
------------|--------------------------------------
            | p.X                             london

results: NULL  /* the empty set */
```

that is, the system recognizes the failure of the component:

```
<- suppliers(X, _, _, Y),
   Y=london.
```

and responds appropriately.

On the other hand, an OR-query fails if and only if all of its sub-queries fail. If one succeeds, the query as a whole succeeds, even if all the others fail. Such a situation might contribute to the misinterpretation by the user of the system's response due to the false assumptions made about the way the answer was inferred.

For example, suppose that the user wishes to retrieve the names of all suppliers who live in London or Paris. The entry might be:

```
suppliers    | sno      sname       status      city
_________    |
-------------|--------------------------------------
             |          p.X                     london
             |          ___                     ______
             |
             |          p.Y                     paris
                        ___                     _____

CQ  /* signals the end of a complete query */
__
```

to which the system's answer is:

```
results    | sname
-----------|------
           | jones
           | blake
```

But it is known (see Appendix 1) that Jones and Blake both live in Paris, and that no one is in London. That is, the user can think of suppliers living in London and living in Paris to be correct, and carry on with a frustrating series of questions, or worse, misinterpret the system's response. To avoid such a situation, the user can, as soon as the answer has been displayed, request further information about the process used in the evaluating of the query. The result might be:

```
suppliers    | sno      sname       status      city
-------------|--------------------------------------
             |          p.X                     london

results: NULL  /* the empty set */
```

which indicates to the user that the presupposition that some suppliers were living in London is incorrect.

To the extent that update operations involve an initial request (query) aimed at locating certain qualified instances or tuples in the data base, a similar type of analysis as the one described above would apply in the case of failure.

In the case of a query which expresses some property of the data base as a whole, should the system fail to prove the conjecture, an attempt is made to prove the negation of the conjecture in order to answer "NO". Should the system fail to prove or disprove a given conjecture an answer of "DON'T KNOW" is returned to the user. This is the case when neither a "YES" nor "NO" answer is possible from the axioms in the data base.

If the answer is "NO" or "DON'T KNOW" an analysis of the presuppositions made might follow if requested.

6. Conclusions

Most query systems currently available respond to queries in a very literal manner, giving an answer to what the user actually asked for - no more and no less. Though the responses are literally correct, such rigidity can be very unhelpful at times, and a more flexible system is desirable. This flexibility in the interpretation of queries in a manner which is both natural and of benefit to the user is termed co-operativeness.

This paper outlines several ways in which the query language QBE can be made more co-operative. These features have been added to a version of QBE implemented in Prolog, which is running under UNIX on both a PDP 11/34 and a DEC 10 system.

The main features of such a system are:

(i) A link dictionary has been implemented which contains information about the data base relations and the linkages between them. This facility was interfaced with the query facility to provide the user with the means to examine how the data in the data base are organized and how they should be accessed and used.

(ii) The system attempts to handle incomplete queries and updates by filling in link variables. This can be of use to casual users of the data base who do not have the details of the structure of the data base at their fingertips, as well as to experienced users who seek short cuts.

(iii) The system reminds users of possible side effects when updates are performed on link variables.

(iv) The system attempts to provide a helpful response when a complex query fails to give the user an indication of why it failed. The same tabular form is used to explain the reasoning it followed to arrive at the answer as that used to enter the initial request.

Acknowledgements

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700 BRAGA, PORTUGAL.

7. References

[1] T. Moto-Oka et al, Challenge for knowledge information processing systems, in: Fifth Generation Computer Systems (North-Holland, Amsterdam, 1982) 3-89.

[2] D. H. D. Warren, Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[3] H. Gallaire and J. Minker (eds), Logic and data bases, Plenum Press, New York, 1978.

[4] F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks, Artificial Intelligence 13 (1980) 231-278.

[5] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[6] M. M. Zloof, Query-by-Example: A data base language, IBM Systems Journal 16(4) (1977) 324-343.

[7] J. C. Thomas and J. D. Gould, A psychological study of Query-by-Example, Proc. National Computer Conference (1975) 439-445.

[8] M. H. van Emden, Computation and deductive information retrieval, in: Formal Description of Programming Concepts (North-Holland, Amsterdam, 1978).

[9] S. J. Kaplan, Co-operative responses from a portable natural language query system, Artificial Intelligence 19 (1982) 165-187.

Appendix 1: A simple business data base

The examples in this paper make use of the following relations:

(i) A relation called parts with attributes (columns): pno (part number), pname (part name), colour and weight.

(ii) A relation called suppliers with attributes: sno (supplier number), sname (supplier name), status and city.

(iii) A relation called supplier_parts with attributes: sno (supplier number), pno (part number) and qty (quantity supplied).

(iv) A relation called supplier_balance with attributes: sno (supplier number) and amountowed.

(v) A relation called sales_people with attributes: salesno (sales number) and salesname.

(vi) A relation called product_parts with attributes: prodno (product number), pno (part number) and noreqd (number of parts required).

(vii) A relation called sales with attributes: salesno (sales number), prodno (product number) and qtysold (quantity sold).

Typical values of these relations are as follows:

369-B

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	15	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

supplier_balance	sno	amountowed
	1	100
	2	90
	3	0
	4	0
	5	145

Table 1.4 - The supplier_balance relation

sales_people	salesno	salesname
	1	flanagan
	2	ellis
	3	smith
	4	schafer

Table 1.5 - The sales_people relation

product_parts	prodno	pno	noreqd
	1027	1	350
	1023	1	200
	1028	1	100
	1033	3	275
	1040	4	435
	1072	5	555
	1045	2	315
	2001	6	125
	1067	5	111

Table 1.6 - The product_parts relation

sales	salesno	prodno	qtysold
	1	1023	100
	1	1027	45
	2	1028	40
	3	1033	150
	3	1040	75
	4	1072	20

Table 1.7 - The sales relation

Appendix 2: The link dictionary for the data base in Appendix 1.

The link dictionary for the data base in Appendix 1.

```
link(supplier_parts, supplier_balance, [sno]:[sno]).
link(suppliers, supplier_balance, [sno]:[sno]).
link(supplier_parts, product_parts, [pno]:[pno]).
link(parts, product_parts, [pno]:[pno]).
link(sales, product_parts, [prodno]:[prodno]).
link(sales_people, sales, [salesno]:[salesno]).
link(supplier_parts, suppliers, [sno]:[sno]).
link(supplier_parts, parts, [pno]:[pno]).

setofnodes(graph, [supplier_balance, product_parts, sales,
                   sales_people, suppliers, parts,
                   supplier_parts]).
```

The Prolog program for searching for a path linking any pair of relations in the data base:

```
                        ?- op(40, xfx, :).

                        /* declare ":" infix operator   */


   clause 1             path(GRAPH, X, Y, PATH) <-
                        setofnodes(GRAPH, SET),
                        member(X, SET),
                        member(Y, SET),
                        walk(GRAPH, [X], Y, PATH).

   clause 2    walk(GRAPH, [Y|L], Y, [Y|L]).
   clause 3    walk(GRAPH, [X|L], Y, PATH) <-
                        (
                           link(X, Z, _);
                              link(Z, X, _)
                        ),
                        not(member(Z,L)),
                        walk(GRAPH, [Z,X|L], Y, PATH).

   clause 4    member(X, [X|_]).
   clause 5    member(X, [_|Y]) <-
                        member(X, Y).
```