CS 85-2
Tech Report

INTEGRATING A USER'S KNOWLEDGE INTO A KNOWLEDGE BASE USING A LOGIC BASED REPRESENTATION

by

Jose C. F. M. Neves*,
George F. Luger**
&
Luis F. Amaral*

*Department of Computer Science
Minho University
Largo do Paco, 4719 Braga Codex
PORTUGAL

**Department of Computer Science
University of New Mexico
Albuquerque, New Mexico
U.S.A. 87131

April 1985

ABSTRACT

The use of first order logic as a knowledge representation formalism offers a uniform medium for expressing and organizing knowledge as well as providing for efficient reasoning. The process of integrating the user's knowledge into a knowledge base also requires a broad and consistent representation for knowledge, its syntax and semantics. The aim of this paper is to present such a logic framework. The context is that of a logic data base system.

Keywords: First Order Predicate Logic, Horn Clause Logic, Logic Data Base, Query-By-Example, PROLOG.

1. INTRODUCTION

The acquisition and integration of expert knowledge into knowledge bases is fundamental to the creation and maintenance over time of knowledge based systems. This involves two (often overlapping) processes:

- Knowledge acquisition; i.e., obtaining the knowledge from the user.

- Knowledge structuring; i.e., integrating the user's knowledge into the knowledge base.

Logic Programming, evolved from automated theorem proving, has two aspects that make it ideal as a knowledge representation and manipulation formalism: a clear semantics (Hayes(1), Moore (2)), plus a practical language, the language PROLOG (PROgramming in LOGic, Clocksin and Mellish(3)), which the authors view as a first step towards the goal of "programming in logic".

Query-By-Example (QBE, Zloof(4)) is a data base query language developed for querying relational data bases (Codd(5)). The similarity between QBE syntax and PROLOG goals has been noted (Neves et al(6)).

The main goal of this work is to present a "layered" representation formalism built on an ordered set of first order logic languages for the process of translating from a sentence in a data base query language (such as QBE or natural language) to a Logic Data Base of ground clauses and general rules. None of the meta-theories for the ordered set of logic languages will be addressed here; i.e., although the ordered set of first order logic languages are in an object meta-object relationship, the partial axiomatization of the theory and proof procedure of one language in terms of another will not be considered (these inter-relationships will be investigated in subsequent papers). Finally, in an application, it is shown that in relating "aggregate" and "select" operators to logic, a clear semantics is established for both easy query handling and straightforward, efficient reasoning.

2. THE UNDERLYING LOGIC DATA BASE

The following conventions are used in this paper:

- Variables are denoted by strings of symbols beginning with an upper-case letter or the symbol underscore "_".

- Constants are denoted by strings of symbols starting with lower-case letters or digits.

- {X1,X2,...,Xn} denotes the set whose members are X1,X2,...,Xn.

- <X1,X2,...,Xn> denotes the ordered set or n-tuple whose members are X1,X2,...,Xn.

- An n-ary relation is a set of n-tuples.

- An n-ary relation is defined in a sentence of the form r(X1,X2,...,Xn) (e.g., r(X1,X2) for "X1 bears r to X2").

- With an n-ary relation r one associates the set of n-tuples that forms the extension of r; i.e., r = {<X1,X2,...,Xn> | r(X1,X2,...,Xn)}. The symbol "|" is "such that".

- <X1/T1,X2/T2,...,Xn/Tn> $\in$ r (r a relation), indicates that X1,X2,...,Xn are free variables of r and

T1,T2,...,Tn are their types. <X1/T1,X2/T2,...,Xn/Tn> is said to belong to the extension of relation r. "ϵ" is the element relation.

- An n-place predicate is interpreted as a set of ordered n-tuples; if considered semantically, an n-place predicate is called a relation.

- By a function is meant the assignment (function) which assigns an extension to each predicate constant in each interpretation.

- A term is defined inductively as:
  - (i) A variable is a term.
  - (ii) A constant is a term.
  - (iii) If f is an n-place function and L1,L2,...,Ln are terms, then f(L1,L2,...,Ln) is a term.

- An atom or atomic formula is defined inductively as: if p is an n-place predicate and L1,L2,...,Ln are terms, then p(L1,L2,...,Ln) is an atom or atomic formula.

- A literal is an atom or the negation of an atom.

- A clause is a formula of the form: for all X1,X2,...,Xn ($q \vee \sim p1 \vee \sim p2 \vee \ldots \vee \sim pn$), where q and each pi (i>=0) are literals and X1,X2,...,Xn are the variables ocurring in q and each pi. This clause is denoted by q <- p1,p2,...,pn. The symbol "$\sim$" is used for negating the truth value of a formula, the symbol "$\vee$" is inclusive "or", and the reverse arrow "<-" is "if". Clauses in this form are often referred to as "Horn clauses".

A logic data base is made up of a finite set of Horn clauses, which are universally quantified logic formulae of the general form: q <- p1,p2,...,pn. Such a logic formula can be interpreted procedurally as: to satisfy goal p, satisfy goals p1 and p2 and ... and pn; or declaratively as: goal p is true if goals p1 and p2 and ... and pn are also true. In a typical data base state one may have the ground clauses (Appendix 1):

```
parts(1,nut,red,12).                        parts(2,bolt,green,17).
suppliers(1,smith,20,vienna).               suppliers(2,jones,10,paris).
supplier-parts(1, 1, 300, 1).               supplier-parts(2, 1, 300, 1).
sales(1, 1, 1023, 100).                     sales(3, 2, 1028, 40).
employee(12, morley, 2, 6500, married, 10). employee(7, warren, 4, 7135, single, 7).
loantable(7, 570).                          loantable(17, 1500).
```

3. THE BASIC SYSTEM

It is clear that there is no "best" man-computer interface when different user groups, tasks and application areas are taken into account. Thus, it seems natural to expect that the interface be a multimodel, adaptive system integrating the basic communication modes:

characters * (voices) * figures * images

The communication mode used throughout this paper is an extension to the query language QBE which has been implemented in PROLOG and is described in Neves (7). This is done without any loss of generality and only for the sake of exposition. The distinct "layers" for the different types of knowledge that make up the representations of the logic system are depicted in Figure 1.

USER LEVEL INTERFACE

characters * (voices) * figures * images

PLANNING LEVEL INTERFACE

An high level representation formalism based on first order predicate logic.

METHODS LEVEL INTERFACE

A machine oriented representation formalism for efficient reasoning based on an extension to the language of Horn clause logic.

DOMAIN LEVEL INTERFACE

A machine dependent representation formalism based on Horn clause logic.

Figure 1 - Knowledge representation levels for a logic based data system.

The distinct levels exist for the different types of knowledge representation that make up the logic data base system. The system operates in the following manner. On striking a particular (function) key in the keyboard the user is presented with the skeleton of a table as follows:

```
          |
__________|____________________________
          |
```

The user then proceeds to enter a request by filling in some of the quadrants delimited by the skeleton with information about relations in the data base in the form of example elements or variables, constants and operators. As an example, suppose that a user who has access to the data base given in Appendix 1, wishes to find the name and status of each supplier who lives in Paris and supplies parts to department number 1. The user enters (the user's contribution is **bold**):

suppliers	sno	sname	status	city
	X	**p.N**	**p.A**	**paris**

supplier-parts	sno	pno	qty	deptno
	X			**1**

The operator "p." stands for "print". The full syntax of the query language is given in Appendix 2.

4. A FORMAL BODY OF DESIGN KNOWLEDGE

Multilevel problem solving methods are an efficient tool for knowledge acquisition by providing the framework for the articulation and creation of domain expertise; the problem is incorporating the different levels efficiently. Axiomatizing properly at each level is essentially building a representation language for each level.

4.1. PLANNING LEVEL INTERFACE

First order logic provides an ideal framework for formal, abstract knowledge specification due to its soundness and completeness properties. The assertion language given below is essentially a typed first order predicate logic language that allows both for abstraction and for subsets on a type. If T denotes a type, and X a proposition on T, $\{z \in T \mid X\}$ stands for the set of occurrences of z in X bound by $z \in T$. Sentences in this language are given by the context-free grammar:

planning-level-sentence ::= "{" basic-first-order-formula [("," basic-first-order-formula)*] "}"

basic-first-order-formula ::= [planning-level-primitive] "{" planning-level-knowledge-base-object "|" first-order-formula [("&" first-order-formula)*]"}"

first-order-formula ::= [planning-level-primitive "{"] preposition "(" pronoun tuple element-relation relation-name "holds" basic-planning-level-term [("&" basic-plannng-level-term)*]")" ["}"]

first-order-formula ::= proposition "(" planning-level-primitive tuple element-relation relation-name "holds" basic-planning-level-term [("&" basic-planning-level-term)*]

planning-level-knowledge-base-object ::= tuple element-relation relation-name | boolean | string-variable | string-constant

basic-planning-level-term ::= planning-level-term [("&" planning-level-term)*] | planning-level-term "<-" planning-level-term [("&" planning-level-term)*] | first-order-formula

planning-level-term ::= relation-name "(" term [("," term)*] ")"

planning-level-primitive ::= "cardinality-of" | "average-of" | "minimum-of" | "sum-of" | "maximum-of"

preposition ::= "for"

pronoun ::= "each" | "all" | "which" | "every" | "the" | "any"

term ::= string-variable | string-constant | underscore

tuple ::= "<" (string-variable "/" type | string-constant "/" type) [("," (string-variable "/" type | string-constant "/" type))*] ">"

relation-name ::= lower-case-letter*

type ::= integer | boolean | character

element-relation ::= "$\in$ "

The full set of productions are given in Appendix 2.

4.2. MEHODS LEVEL INTERFACE

Each data base object or operation at the methods level is given in an extension to the language of Horn clause logic defined by formulae of the form:

holds(<Q1/T1,Q2/T2,...,Qj/Tj> $\in$ U pr, (q <- p1,p2,...,pi), S) <- p1,p2,...,pi.

where Q1,Q2,...,Qj are objects (variable terms or constants) that may occur as arguments in q and each pi (i >= 1, j>= 1), and T1,T2,...,Tj are their types. This may be read as:

for all Q1,Q2,...,Qj in the domain of each pr such that if each pr (r<=i) holds then holds(<Q1/T1,Q2/T2,...,Qj/Tj> $\in$ U pr, (q <- p1,p2,...,pi), S) holds.

Every occurence of a predicate symbol pr in the q and each pi is to be regarded first as a propositional function symbol (in the clause head) and second as a predicate symbol (in the clause body). Such a propositional function of several variables may be interpreted as a condition on n-tuples of objects. For example, pr(Q1,Q2,...,Qm) (m <=j) determines a family of sets or domains S1,S2,...,Sm such that the names of the elements of Sr may be substituted for Qr in pr. S is the data base object that is returned as a consequence of the evaluation of the term conjunction p1,p2,...,pi. The symbol "U" models the process of set union.

Extended-Horn-formulae represent the user's knowledge at the methods level; this is defined according to the context-free grammar:

methods-level-sentence ::= "{" extended-Horn-formula [("," extended-Horn-formula)*] "}"

extended-Horn-formula ::= "{" methods-level-knowledge-base-object "|" extended-Horn-clause [("," extended-Horn-clause)*] "}"

extended-Horn-clause ::= methods-level-primitive "(" methods-level-knowledge-base-object "," basic-term-list ",{" methods-level-knowledge-base-object "}) <- " basic-atom-list

methods-level-knowledge-base-object ::= ["any" integer | "the"] tuple element-relation relation-name | term | boolean-constant

methods-level-primitive ::= "number-of" | "average-of" | "minimum-of" | "sum-of" | "maximum-of"

basic-term-list ::= methods-level-term [("," methods-level-term)*]

basic-atom-list ::= methods-level-atom [("," methods-level-atom)*]

methods-level-term ::= function-constant "(" term [("," term)*] ")" | extended-Horn-clause

methods-level-atom ::= predicate-constant "(" term [("," term)*] ")" | extended-Horn-clause

function-constant ::= methods-level-primitive | lower-case-letter*

predicate-constant ::= methods-level-primitive | lower-case-letter*

tuple ::= "<" (string-variable "/" type | string-constant "/" type) [("," (string-variable "/" type | string-constant "/" type)*] ">"

relation-name ::= lower-case-letter*

type ::= integer | boolean | character

term ::= string-variable | string-constant

4.3. DOMAIN LEVEL INTERFACE

At the domain level knowledge is represented by Horn formulae:

(i) p.
(ii) <- q1, q2, ..., qn.
(iii) p <- q1,q2,...,qn.

- (i) is a unit clause. If X1,X2,...,Xn are the variables of the unit clause p, then this causal notation is an abreviation of "for all X1,X2,...,Xn in the domain of p, p holds".

- (ii) is a goal clause; i.e., a clause with an empty consequent. If X1,X2,...,Xn are the variables of the goal clause, then this causal notation is an abreviation of "for all X1,X2,...,Xn in the domain of each qr (r<=n), ˜ q1 ∨ ˜ q2 ∨ ... ∨ ˜ qn"; or equivalently "there does not exists X1,X2,...,Xn in the domain of qr such that each qr holds".

- (iii) is a program clause with only one positive literal (viz. p). If X1,X2,...,Xn are the variables ocurring in the literals p and each qr (r<=n), then this causal notation is an abreviation of "for all X1,X2,...,Xn in the domain of each qr such that if each qr holds then p holds".

These sentences are defined by the context-free grammar:

domain-level-sentence ::= Horn-formulae

Horn-formulae ::= "{" Horn-clause [("," Horn-clause)*] "}"

Horn-clause ::= [[atom] ["<-" [[atom "."] | [atom ("," atom)* "."]]]]

atom ::= predicate-constant "("term [("," term)*] ")"

term ::= string-variable | string-constant

predicate-constant ::= lower-case-letter*

Horn-formulae represent the knowledge at the domain level; i.e., Horn formulae give a theory of the language, and being SLD resolution, the proof procedure. Standard PROLOG systems execute the proof procedure using depth first left-to-right search as the computation rule.

4.1. AGGREGATE OPERATORS

Aggregate operators act upon relations to produce results which are a summary of their arguments. Examples are the functions to sum attribute values, count the number of entries in a relation, or compute the average of values in a numeric field of a relation. QBE has the usual five aggregate operators, viz.

cnt. - count	sum. - sum	avg. - average	max. - maximum	min. - minimum

These are realized by the meta-logical primitives "number-of", "average-of", "minimum-of", "sum-of", and "maximum-of" respectively.

As an example, suppose that the user wishes to get the total number of suppliers from London who currently supply department number 1. Then enter (at the user level interface):

suppliers	sno	sname	status	city
	p.cnt.X			**london**

suppliers-parts	sno	pno	qty	deptno
	X			**1**

This is translated by the system to yield the logic sentence (at the planning level):

cardinality-of {<X/sno> ∈ suppliers | for(each <X/sno, _, _, _> ∈ suppliers holds
(suppliers(X,_,_,london) & supplier_parts(X,_,_,1)))}

The symbol underscore "_" stands for a unique variable existentially quantified immediately before the atom containing it. The ampersand "&" is used to denote conjunction.

The use of "number-of" can now be formalized (at the methods level) as:

{S | number-of((<Q1/T1,Q2/T2,...,Qj/Tj> ∈ U pr), (<- p1,p2,...,pi), S)}

This logic expression returns as its value, S, the cardinality of the set of data base objects that satisfy the term conjunction p1,p2,...,pi. (The use of the remaining operators may be formalized in a similar way). The expression above is then translated into the query language expression (at the methods level):

{N ∈ integer | number-of(<X/sno, _, _, _> ∈ suppliers,
(<- (all(<X/sno, _, _, _> ∈ suppliers, (<- (suppliers(X,_,_,london),
supplier_parts(X,_,_,1))), {<X>}))), N)}

The function "all" is used to ask for an exhaustive listing of data base objects. It returns as its value the list of data base objects (e.g., tuples) that satisfy the term conjunction p1,p2,...,pi. Formally:

{S | all((<Q1/T1,Q2/T2,...,Qj/Tj> ∈ U pr), (<- p1,p2,...,pi), S)}

Another quantifier is "any K" (K being an integer), which can be expressed as:

{S | any K((<Q1/T1,Q2/T2,...,Qj/Tj> ∈ U pr), (<- p1,p2,...,pi), S)}

This logic expression returns as its value a list of no more than K data base objects (e.g., tuples) that satisfy the term conjunction p1,p2, ..., pi. As an example, suppose that the user wants to print three entries in the suppliers relation for suppliers that supply parts. Then enter (at the user interface level):

suppliers	sno	sname	status	city
p.any 3.	**X**	**Y**	**Z**	**W**

supplier-parts	sno	pno	qty	deptno

| X

This query is then translated by the system to yield the logic sentence (at the planning level):

```
{any 3 <X/sno, Y/sname, Z/status, W/city> ∈ suppliers |
  for(any <X/sno, Y/sname, Z/status, W/city> ∈ suppliers holds
      (suppliers(X,Y,Z,W) & supplier-parts(X,_,_,_)))}
```

This expression may then be translated into the query language (at the methods level):

```
{ any 3 <X/sno, Y/sname, Z/status, W/city> ∈ suppliers
    | any(<X/sno, Y/sname, Z/status, W/city> ∈ suppliers,
        (<- suppliers(X,Y,Z,W), supplier-parts(X,_,_,_)), {<X, Y, Z, W>})}
```

Yet another useful quantifier is "the", which can be expressed as:

```
{S | the((<Q1/T1,Q2/T2,...,Qj/Tj> ∈ U pr), (<- p1,p2,...,pi), S)}
```

This expression succeeds only if there is a sole data base object, namely the data base object S, that satisfies the term conjunction p1,p2,...,pi and fails otherwise.

4.2. SELECT OPERATORS

Select operators provide criteria to follow in choosing a subset of the relation(s) arguments to which they apply. QBE provides a single select operator, the operator "for each". It is equivalent to the SQL "GROUP BY" operator (Gray (8)). The select operator conceptually splits the result relation into partitions such that within any one partition all rows have the same value for the select field. As an example, suppose that the user wants to get, for each department, the names of employees who have loans. Then enter (at the user interface level):

employee	empno	ename	deptno	salary	status	grade
	X	**p.N**	**foreach.D**			

loantable	empno	loan
	X	**Y --> (Y \=== 0)**

The symbol "-->" is "such that", and "\===" denotes strong inequality. This is translated by the system to yield the logic sentence (at the planning level):

```
{ <N/ename, D/depno> ∈ employee group-by D |
  for(each <_, N/ename, D/deptno, _, _, _> ∈ employee holds
    (employee(X,N,D,_,_,_) & loantable(X,Y) & Y\===0))}
```

This expression is then translated into the query language (at the methods level):

```
{ <N,D> group-by D |
  all(<_, N/ename, D/deptno, _, _, _> ∈ employee group-by D,
    all(<_, N/ename, D/deptno, _, _, _> ∈ employee,
        (employee(X,N,D,_,_,_), loantable(X,Y), Y\===0), {<N,D>}), {<N,D> group-by D}))}
```

The evaluation of the query will produce a list whose components are lists of elements N (employee names) grouped by D (their department number).

4.3. YES-NO QUERIES

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. As an example of the latter, suppose that the user wants to know if there are suppliers that supply parts numbered 1 and 2 who live in London. Then enter (at the user interface level):

suppliers	sno	sname	status	city
any.	X	R	S	**london**

supplier-parts	sno	pno	city	deptno
	X	1		
	X	2		

This is then translated by the system to yield the logic sentence (at the planning level):

{ true | for(any <X/sno, R/sname, s/status, london/city> ∈ suppliers holds
(supplier(X,R,S,london) & supplier-parts(X,1,_,_) & supplier-parts(X,2,_,_)))}

This expression is then translated into the query language (at the methods level):

{ true | any(<X/sno,R/sname,S/status,london/city> suppliers,
(suppliers(X,R,S,london), supplier-parts(X,1,_,_), {<X,R,S,london>))}

5. DATA BASE UPDATES

Insert, delete and modification operations have a uniform expression in the language that makes use of the same syntax and semantics used for queries. Tuples may be inserted, deleted or modified. New tuples can be generated from existing tuples or from constituents of tuples in one or more relations. Query dependent modications may also be specified. As an example, suppose that the user wants, for each department, to derive those entries in the "supplier-parts" relation for suppliers that supply only one part which has color red. The part-numbers for these parts are to be changed by a factor of 10. Then enter (at the user interface level):

supplier-parts	sno	pno	qty	deptno
	Y	X	A	B

parts	pno	pname	color	weight
the.	X	L	red	M

supply	sno	pno	qty	deptno
i.	Y	NX --> (NX = 10*X)	A	**foreach.B**

This representation is translated by the system to yield the logic sentence (at the planning level):

```
{ <Y/sno,NX/pno,A/gty,B/deptno> ∈ supply group-by B |
  for(each <Y/sno,X/pno,A/gty,B/deptno> ∈ supplier-parts holds
    supplier-parts(Y,X,A,B) & for(the <X/pno, L/pname, red/color, M/weight> ∈ parts holds
        parts(X,L,red,M)) & NX = 10*X))}
```

This expression is then translated into the query language (at the methods level):

```
{ <Y,NX,A,B> ∈ supply group-by B | all(<Y/sno,NX/pno,A/qty,B/> ∈ supply group-by B,
    (<- all(<Y/sno,X/pno,A/qty,B/deptno> ∈ supplier-parts, supplier-parts(Y,X,A,B),
          the(<X/pno,L/pname,red/color,M/weight> ∈ parts,
          (<- parts(X,L,red,M)), {<X,L,red,M>})), {<Y,X,A,B>}),
                NX=10*X, {<Y,NX,A,B> group-by B}))}
```

When entering new information, the incoming data must be checked for consistency. Representing the logic programming system as LP, and a set of integrity constrains as IC, the insertion operation may be expressed formally as:

```
LP U IC U {<Y,NX,A,B> ∈ supply} |– false
```

The symbol "U" models the process of set union, the symbol "|–" is to be read as "is deducible from", while the constant "false" denotes contradiction. The entry in {} stands for the set of data base objects (i.e., tuples) returned as a result of the evaluation of the query.

Logic programming systems use resolution as the inference rule (9). Resolution theorem provers are refutation systems; i.e., the negation of the formula to be proved is added to the axioms and a contradiction is derived. If it turns out to be inconsistent, the proof of inconsistency is returned as the answer to the query(Neves and Luger (10)), providing the information in the data base which contributes to the inconsistency of the statement.

As another example, suppose that the user wants to delete all the entries in the "supplier-parts" relation for suppliers that supply parts, at least one of which is red. Then enter the query:

supplier-parts	sno	pno	qty	deptno
d.	Y	X	M	N

parts	pno	pname	color	weight
any.	X	R	**red**	T

This is translated by the system to yield the logic sentence (at the planning level):

```
{ <Y/sno, X/pno, M/gty, N/deptno> ∈ supplier-parts |
  for(any <X/pno, R/pname, red/color, T/weight> ∈ supplier-parts holds
       (supplier-parts(Y,X,M,N) & parts(X,R,red,T)))}
```

This expression is then translated into the query language expression (at the methods level):

```
{ <Y,X,M,N> ∈ supplier-parts | all(<Y/sno,X/pno,M/qty,N/deptno> ∈ supplier-parts,
    (<- supplier-parts(Y,X,M,N), any(<X/pno,R/pname,red/color,T/weight> ∈ parts,
        (<- parts(X,R,red,T)), {<X,R,red,T>})), {<Y,X,M,N>}))}
```

As stated before if LP is a first order theory and IC is a set of integrity constraints, then the deletion operation referred to above may be expressed formally as:

$LP \cup IC - \{<Y,X,M,N> \in \text{supplier-parts}\} \vdash \text{false}$

where the symbol "-" models the process of set subtraction.

6. CONCLUSIONS

This paper presents a set of ordered languages for knowledge representation in the context of a logic data base. These languages guide the decomposition of data base problems into sub-problems in such a fashion that they provide a guide to their solution. The ordered set of representation languages provides intermediate abstractions, developed for the process of integrating a user's knowledge (as a sentence in the data base query language QBE) into a data base of knowledge (as a set of ground clauses and general logic rules).

The ordered languages turn out to be effective tools. Abstractions take the form of detail supression, and implementation relationships (i.e., relationships among constructs of different types). Abstract constructs are then implemented in terms of expanded constructs at lower levels. By recording all the levels in a formal notation, insight is gained in both comparing and understanding different sets of abstractions, as well as in grasping the principles of their design. In other words, breaking a problem into a hierarchy of sub-problems, leads ultimately to the understanding the causes, measures, and methods for further application of knowledge and the eventual production of solutions.

Finally, in relating both aggregate and select operators to logic, a clear semantics is provided for these operators that proves satisfactory for translating queries from QBE into sets of PROLOG goals.

ACKNOWLEDGEMENTS

We wish to thank our students at Minho University and University of New Mexico who triggered many useful discussions. This research is supported by a North Atlantic Treaty Organization grant for International Collaboration in Research awarded to George F. Luger and Jose C. F. M. Neves.

7. REFERENCES

(1) Hayes, P., *In Defence of Logic,* Proceedings IJCAI-5, 1977, 559-565.

(2) Moore, R., *The Role of Logic in Representation and Commonsense Reasoning,* Proceedgins AAAI-82, 1982, 428-433.

(3) Clocksin, W.F. and Mellish, C.S., *Programming in Prolog,* Springer- Verlag Publishing Co., New York, 1981.

(4) Zloof, M.M., *Query-By-Example - A Data Base Language,* IBM Systems Journal, 16, 4, 1977, 324-343.

(5) Codd, E. F., *A Relational Model for Large Shared Data Banks,* CACM 13, 6, 1970, 337-387.

(6) Neves, J.C.F.M., Anderson, S.O. and Williams, M.H., *A Prolog Implementation of Query-By-Example,* Proceedings of The 7th International Computing Symposium on Application Systems Development, (H.J. Schneider, Ed.), Nurnberg, Germany, 1983, 318-332.

(7) Neves, J.C.F.M., *The Application of Logic Programming to Data Bases,* Ph.D. Thesis, Heriot-Watt University, Edinburgh, U.K., 1983.

(8) Gray, P.M.D., *The 'Group-By' Operation in Relational Algebra,* Proceedings of the 1st British National Conference on Data Bases, Cambridge, U.K., 1981, 84-98.

(9) Neves, J.C.F.M., and Luger, G.F., *An Automated Reasoning System for Presupposition Analysis,* Research Report CS 85-3, The University of New Mexico, Albuquerque, New Mexico, 1985.

APPENDIX 1 - A DEPARTMENT STORE DATA BASE

Consider a simple department store data base which contains ("∈" is the element relation):

(i) A relation called "parts" with attributes: pno (part number), pname (part name), color and weight. Intended meaning: <A, B, C, D> ∈ parts if A is the number of a part called B which is of color C and has weight D.

(ii) A relation called "suppliers" with attributes: sno (supplier number), sname (supplier name), status and city. Intended meaning: <A, B, C, D> ∈ suppliers if A is the number of the supplier called B who has status C and lives in city D.

(iii) A relation called "supplier-parts" with attributes: sno (supplier number), pno (part number), qty (quantity supplied) and deptno (number of the department to which the part is being supplied). Intended meaning <A, B, C, D> ∈ supplier-parts if A is the number of the supplier that supply part number B in the amount C to department D.

(iv) A relation called "sales" with attributes: deptno (department number), salesno (sales number), prodno (product number) and qtysold (quantity sold). Intended meaning: <A, B, C, D> ∈ sales if in department number A sale number B sold product C in the quantity D.

(v) A relation called "employee" with attributes: empno (employee number), ename (employee name), deptno (department number), salary and grade. Intended meaning: <A, B, C, D, E, F> ∈ employee if employee number A called B works for department C earns salary D has status E and grade F.

(vi) A relation called "loantable" with attributes: empno (employee number) and loan. Intended meaning: <A, B> ∈ loantable if to employee number A is granted a loan B.

Typical values of these relations are:

parts	pno	pname	color	weight
	1	nut	red	12
	2	bolt	green	17

Table 1 - The "parts" relation.

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris

Table 2 - The "suppliers" relation.

supplier-parts	sno	pno	qty	deptno
	1	1	300	1
	2	1	300	1

Table 3 - The "supplier-parts" relation.

sales	deptno	salesno	prodno	qtysold
	1	1	1023	100
	3	2	1028	40

Table 4 - The "sales" relation.

employee	empno	ename	deptno	salary	status	grade
	12	morley	2	6500	married	10
	7	warren	4	7135	single	7

Table 5 - The "employee" relation.

loantable	empno	loan
	7	570
	17	1500

Table 6 - The "loantable" relation.

APENDIX 2 - CONCRETE SYNTAX FOR EXTENDED-QUERY-BY-EXAMPLE

The basic Extended-Query-By-Example (EQBE) format is as follow:

```
Table-name-field      | Column-name-field
______________________|____________________
Tuple-command-field   | Tuple-entry-field

|______________________________|
|      CONDITIONS              |
|______________________________|
|  Condition-entry-field       |
```

where the syntax of each of these components is defined as:

table-name-field ::= [string-constant "->" string-constant | string-constant | example-element | "p."]

column-name-field ::= ["p." | string-constant ["->" [string-constant]] | "->" string-constant]

tuple-entry-field ::= ["[" string-primary (" " string-primary)* "]" | ["p."] ["all." | "any." integer "."
| "the."] example-element | [example-element] p-relation | example-element "->". (numerica-expr
| string-expr) | string-constant | integer]

tuple-command-field ::= ["i." | "d." | "u." | ["p."]]"all." | "any" integer "." | "the."]
| integrity | scheme | type | security | links | triggers | views]

integrity ::= "dc." | "ic." | "pc."

scheme ::= "ds." | "is." | "us." | "ps."

type ::= "dty." | "ity." | "uty." | "pty."

links ::= ("i." | "d." | "p.") "links." users-list | ("il." | "dl." | "pl.") users-list

triggers ::= "tevent." user-operation "." users-list | ("i." | "d." | "p.") "taction." trigger-operation
| ("it." | "dt." | "pt.") trigger-operation

views ::= ("i." | "d." | "p." | "u.") "view." users-list | ("iv." | "dv." | "pv." | "uv.") users-list

user-operation ::= "[" ("i." | "d." | "p." | "u.")* "]" | example-element

trigger-operation ::= "[" ("i." | "d." | "p." | "u.")* "]" | example-element

security ::= (("i." | "d." | "p.") "autr." | "ia." | "da." | "pa.") authorization

authorization ::= access-rights-list "." users-list

access-rights-list ::= "(" access-right ("," access-right)* ")" | "." example-element

access-right ::= "p." | "i." | "d." | "u."

```
users-list ::= list | example-element

list ::= "[" string-constant ("," string-constant)* "]"

condition-entry-field ::= functional-dependency | multivalued-dependency
    | embedded-multivalued-dependency | boolean-expression

functional-dependency ::= set "->" example-element

multivalued-dependency ::= set "->->" set

embedded-multivalued-dependency ::= set "->->" set "/" set

set ::= "[" example-element ( "," example-element)* "]" | "[" example-element "]"

boolean-expression ::= boolean-secondary (" implies" boolean-secondary)*

boolean-secondary ::= boolean-term ( "or" boolean term)*

boolean-term ::= boolean-factor ( "and" boolean-factor)*

boolean-factor ::= ["not"] boolean-primary

boolean-primary ::= boolean-constant | relation | "(" boolean-expression ")"

boolean-constant ::= "true" | "false"

relation ::= numeric-exp relational-op numeric-exp | string-exp relational-op string-exp

p-relation ::= relational-op (numeric-exp | string-exp)

numeric-exp ::= [add-op] numeric-term (add-op numeric-term)*

numeric-term ::= factor (multiply-op factor)*

factor ::= [function-designator] [".all."] numeric-variables | integer | "(" numeric-exp ")"

relational-op ::= "<" | "=<" | "<=" | "=" | ">=" | "=>" | ">" | "<>"

multiply-op ::= "*" | "/"

add-op ::= "+" | "-"

function-designator ::= "max." | "min." | "avg." | "the." | "cnt." | "any" integer "." | "sum." | "foreach."

string-exp ::= string-primary ( "+" string-primary)*

string-primary ::= string-variable | string-constant

integer ::= digit+

string-constant ::= (""""non-quote-character*"""")+ | lower-case-letter [letter-or-digit*] | digit*
```

string-variable ::= example-element

numeric-variable ::= example-element

example-element ::= capital-letter [letter-or-digit*] | underscore [letter-or-digit*]

letter-or-digit ::= lower-case-letter | digit

capital-letter ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L"
| "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

lower-case-letter :: = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l"
| "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

underscore ::= "_"