A Universal Relation Approach For Natural Query In Logic Database System

HARIHODIN SELAMAT
Fakulti Sains Komputer dan Sistem Maklumat,
Universiti Teknologi Malaysia,
54100 Jalan Semarak,
Kuala Lumpur.

ABSTRACT

Recent advances in Artificial Intelligence and Relational Database systems have contributed to the development of Logic Database systems. A lot of research in logic database have been encompassed on the query evaluation and optimization techniques. Very litde effort has been put to the development of the user's query system to facilitate the naive users interacting with the database. Currently, the form of query is based on the primitive logic form. Therefore, this project aims at developing a prototype natural query system to the logic database as a compromise between the primitive logic query and the natural language query systems. This paper describes a conceptual framework of the natural query system. The concept of predicate universal relation is introduced as an interface to bridge the natural query and the corresponding primitive logic query. The concept of data types is employed as a tool towards the construction of the predicate universal relation. We believe this is the first attempt towards a natural query system for logic database via a universal relation approach.

Keywords: query interface, logic database, deductive database, universal relation, query processing, query language, natural query, first order logic, data types

INTRODUCTION

The integration of concepts from artificial intelligence and database management creates a promise of more intelligent use and manipulation of data. Recent advances in Artificial Intelligence and Relational Database systems have contributed to the development of Logic Database (sometimes called Deductive Database) systems. Logic database is based on first-order predicate logic [Lloyd 1983]. Every logic database user would expect that in the future, databases will be easier to access and use. At present users have to learn complex, formal logic query which require considerable knowledge about the database structure. The immediate consequence is that the ability to utilise data is limited. A lot of research in logic database have been focussing on the query evaluation and optimization techniques. Currently, the form of query to the logic database is based on the primitive logic form. Very little effort has been put to the development of the user query language to facilitate the naive users interacting with the database.

Accesses to information systems and its databases by she end-users have traditionally been via a computer specialist who transforms the user's request into some formal database query languages. Computers understand a formal language which is usually difficult to comprehend by ordinary people. On the other hand, the easiest way for humans to communicate with computers is to talk in terms of natural language (NL). The link connecting the end-users and database system is therefore an interface which hide the users from the technical details of the system. Several studies have focus on the natural language support to facilitate end-user accesses. Many researchers have experimented such systems and high-lighted some of the issues [Barr 1982]. However, H.W. Beck noted that developing programs that can process natural language has been and is still a difficult task [Beck 1990].

An altemative approach to ease user query formulation is via a universal relation assumptions [Ullman 1989, Brady 1985, Maier 1984, Kent 1981]. This approach have been mainly applied and experimented on relational database system. The main aim is to achieve logical data independence which is the main objective of all database management systems. In relational databases, navigation is based on data values, using the relational operators. Knowledge of the attributes, relation names and logical structure of

PERPUSTAKAAN SULTANAH ZANARIAH
Universiti Teknologi Malaysia

the database is necessary. This is the drawback that is intended to be overcome by the universal relation as an interface between the user's view and the actual database structure. In this approach, only the attribute names are required in the query formulation.

The motivation of this project is therefore the simplification of user's view of the logic database whereby the knowledge of the predicate/relation names and of which attributes/arguments belong to which predicate is no longer required. This project aim at developing a prototype natural user query language for the logic database as a compromise between the primitive logic query and the natural language query systems. This paper describes a conceptual framework of the natural query processing system. It begins with a brief introduction of the logic database system considered in this project. Section 3 briefly introduces the concept of data types followed by the notion of universal relation assumptions in Section 4. Section 5 introduces the notion of predicate universal relation and discuss how data types is used as a mechanism in the definition and construction of the predicate universal relation. Section 6 introduces our version of natural query language with respect to predicate universal relation and followed by the framework of the query interpretation.

LOGIC DATABASE

An Introduction

In this section, we will briefly introduce the syntax and semantics of logic database (LDB). LDB (sometimes known as deductive database) evolved from the combined applications of logic programming and relational database systems [Lloyd 1983, Gallaire 1977]. An LDB expressed in First Order Predicate Logic (FOPL) has the capability of reasoning. It consists of a collection of facts (extension) and rules (intension). There is no direct specification of algorithm to be executed but requires specification of objects, properties of objects and relationships between objects. In most LDB and AI systems, rules and facts are used to represent assertional knowledge about the domain of discourse in the form of Horn clauses. J. W. Lloyd and D. Jacobs conclude that relational database is a special kind of logic database [Jacobs 1985].

From a conceptual point of view, an LDB may be viewed as a Prolog program [Clocksin 1984]. J. W. Lloyd has demonsrated this concept by means of using Prolog clauses to represent facts, and how simple deductions can be carried out Lloyd 1983]. It is shown that LDB offers the expressibilty of logic in the form of FOPL for data modelling, at the same time allowing the possibility of space saving by having large number of tupples collapse into a general rule. S. A. Naqvi noted that when FOPL is coupled with databases, it has the ability to capture the semantics of database operations and the integrity constraints on the data in one consistent and uniform language, at the same time increased functionality [Naqvi 1986] This section defines the form of the logic database considered in our project.

Definition 1.1: An *alphabet* of a first order language is composed of a set of variables, a set of constants, a set of predicates,a set of connectives, a set of quantifiers, a set of punctuation symbols.

Variables correspond to attributes which have an association to their respective set of domain values. This concept is normally denoted by $A_i/Dom(A_i)$ where A_i represent an attribute or variable and $Dom(A_i)$ represent its permissable set of domain values with respect to that attribute.

Definition 1.2: A *term* is defined inductively as follows:

* A variable is a term.
* A constant is a term.

Definition 1.3: A *well-formed formula* (wff) is defined inductively as follows:

* If p is an n-ary predicate and $t_1,..., t_n$ are terms, then $p(t_1, ..., t_n)$ is a formula (called an atomic formula or an atom).
* If F and 6 are formulas, then so are (¬,F), (F L G), (F V G), (F− > 6), and (F ′ G).
* If F is a formula and X is a variable then (VX) F and ($X) F are formulas.

Definition 1.4: The first *order language* given by an alphabet consists of all formulas constructed from the symbols of the alphabet.

Definition 1.5: A *closed formula* is a formula with no free occurrences of any variable (a free variable is a variable which is not quantified).

Definition 1.6: A *literal* is an atom or the negation of an atom. A positive literal is just an atom. A negative literal is the negation of an atom.

Definition 1.7: A clause is a formula of the form

$(L_1 V ... VL_m)$,

where each L_i (0 £ i £ m) is a literal.

In the following, a clause $(A_1 V ... V A_k V \neg B_1 V ... V \neg, B_n)$, where $A_1, ..., A_k, \neg, B_1, ..., \neg, B_n$ are atoms, will be denoted by

$A_1, ..., A_k$ ¨ $\neg B_1, ..., \neg B_n$.

Thus, in this notation, all the variables are assumed to be unversally quantified, the commas in the antecedent $B_1, ..., B_n$ denote conjunction and the commas in the consequent $A_1, ..., A_k$ denote disjunction.

Definition 1.8: A *database clause* is aclause of the form

A ¨ B1, ..., B_n

which contains precisely one positive literal (i.e. A). A is called the head and $B_1, ..., B_n$ is called the body of the database clause.

Definition 1.9: A *logic database* is a finite set of database clauses.

Definition 1.10: A *logic database L*, is a theory composed of an *extensional database* (denoted by EDB) and an *intensional database* (denoted by IDB). The EDB is a set ground instances of atoms defining the extensions of the base predicates. The IDB is a set of deduction rules defining the virtual predicates.

Definition 1.11: A *goal* clause is a clause of the form

< – $B_1, ..., B_n$

that is, a clause with an empty consequent. Each B_i (i = 1..n) is called a subgoal of the goal clause.

Definition 1.12: A *Horn* clause is a clause which is either a database clause or a goal clause.

Models of a Logic Database

The declarative semantics of a logic database is given by the model-theoretic semantics of first order logic. This section introduces the notion of interpretations and models which play a particular role in the theory.

Model-Theoratic Semantics

M. H. Van Emden and R. A. Kowalski noted two kinds of semantics; operational and fixpoint. Which have been defined for programming languages Emden 1976]. Operational semantics defines the input-output relation computed by a program in terms of the individual operation evoked by a program inside a machine. The meaning of the program is the input-output relation obtained by executing the program on the machine. As the machine independent alternative to operational semantics, fixpoint semantics defines the meaning of a program to be the input-output relation which is the minimal fixpoint of a transformation associated with the program. Fixpoint semantics is a special case of model-theoretic semantics.

Model-theoretic semantics provides a simple method for determining the denotation (meaning) of a predicate symbol p in a database L;

$$d(p) = \{(t_1, ..., t_n)|L| = p(t_1, ..., t_n)\},$$

where the notation X |= Y means that X logically implies Y. d(p) is the denotation of p as determined by the model-theoretic semantics. The completeness of first-order logic means that there exists an inference system X |- Y iff X |= Y.

Definition 2.1: An *interpretation* o f first order language consists of the following:

* A non-empty set D, called the domain of the interpretation.
* For each constant a, the assignment of an element a' in D.
* For each n-ary predicate p, the assignment of a relation p' on Dn.

It follows from thin definition that the domains are interpreted as unary predicates. Given a domain d, we will note a (E d instead of Dom(a) when convenient.

Definition 2.2: Let I be an interpretation and let F be a closed formula. Then I in a *model* for F if the truth value of F wrt I in true.

Definition 2.3: Let S be a set of closed formulas and I be an interpretation. We say I is a *model* for S if I is a model for each formula of S.

Definition 2.4: Let S be a set of closed formula of a first order language L. We say that S is *satisfiable* if L has an interpretation which is a model for S. S is valid if every interpretation of L is a model for S. S is *unsatisfiable* if it has no models.

Definition 2.5: Let S be a set of closed formulas and F be a closed formula of a first order language L. We say that F is a *logical consequence* of S denoted by S |= F if, for every interpretation I of L, I is a model for S implies that I is a model for F.

Proposition 2.1: Let S be a set of closed formula of a first order language. Then F is a logical consequence of S iff S U (¬F) is unsatisfiable.

Proof: The proof can be found in [Lloyd 1984].

The basic problem in logic programming is, given a set of database clauses L and a goal G, to show that LU (¬G) is unsatisfiable. However, only a specific class of interpretations, namely the Herbrand interpretations, needs to be investigated to prove the unsatisfiablity of L U (¨¬6). We therefore focus our attention on Herbrand interpretation.

Definition 2.6: A *ground* term is a term not containing variables. Similarly, a pound atom is an atom not containing variables.

Definition 2.7: Let L be a first order language. The *Herbrand universe* H_u is the set of all ground terms, which can be formed out of the constants appearing in L.

Definition 2.8: Let L be a first order language. The *Herbrand base* H_b is the set of all ground atoms which can be formed by using predicates from L with ground terms from the Herbrand universe as arguments.

Definition 2.9: Let L be a first order language. An interpretation I for L is a *Herbrand interpretation* if the following conditions are satisfied:

* The domain of the interpretation is the Herbrand universe H_u.
* Constants in L are assigned to themselves in H_u.

Definition 2.10: Let L be a first order language and S be a set of closed formulas of L. A *Herbrand model* for S is a Herbrand interpretation for L which is a model of S.

Proposition 2.2: Let S be a set of clauses and suppose S has a model. Then S has a Herbrand model.

Proof: Let I be an interpretation of S. We define a Herbrand interpretation I' of S as follows:
I' = (p(t_1, ..., t_n)(E H_b|p(t_1, ..., t_n) is true wrt I).
It is straightforward to see that if I is a model, then I' is a model.

A Herbrand interpretation simultaneously associates, with every n-ary predicate symbol in L, a unique n-ary relation over Hu. The relation {(t_1, ..., tn)|p(t_1, ..., t_n) Œ I) is associated by I with the predicate symbol p in L.

* A ground atomic formula A is true in a Herbrand interpretation I iff A Œ I.
* A ground clause L_I V... V L_m is true in I iff at least one literal L_i is true in I.
* In general, a clause C is true in I iff every ground instances C s of C is true in I. (C s is obtained by replacing every occurrences of a variable in C by a term in the Herbrand universe Hu. Different occurrences of the same variable are replaced by the same term).
* A set of clauses A is true in I iff each clause in A is true in I.

Proposition 2.3: Let S be a set of clauses. Then S is unsatisfiable iff S has no Herbrand models.

Proof: If S is satisfiable, then proposition 2.2 shows that it has a Herbrand model.

The Herbrand models of a logic database are subsets of its Herbrand base. Therefore, there exists a unique minimal model called the **least Herbrand model**, which is the intersection of all Herbrand models of the database.

Least Herbrand Model

The least Herbrand model is of central importance in the theory as it contains precisely the set of ground atoms which are the logical consequences of the database. Every logic database has a least Herbrand model. Intuitively, this model reflects all information expressed by the database and nothing more [Nilsson 1990]. Logic databases as Horn theories have the 'model-intersection property'.

Proposition 2.4: Let L be a logic database and Let M(L) be the set of all Herbrand models of L, then LM(L), the intersection of all Herbrand models of L, is itself a model of L (that is, a Herbrand interpretation of L).

Proof: Clearly LM(L) is a Herbrand interpretation for L. Now, suppose LM(L) is not a model. Then, LM(L) falsifies a ground instance C q of a clause C Œ A. Let C q as A ˇB_1,.... B_n (n > = 0). It follows that,
* A œ LM(L)
* B_1, ..., B_n (ELM(L)
Thus for some i Œ I, A œ M(L) and B_1, ..., B_n (E M(L). It follows that C is false in M(L) contrary to the assumption that M(L) is a model.

Every database L has M(L) as a Herbrand model. The set of all Herbrand models for L is non-empty. Thus the intersection of all Herbrand models (M_1LM_2L...LM_n) is a Herbrand model called the ***least Herbrand model*** of L, denoted by M_L. M_L can be characterised as the set of ground instances of atoms that are logical consequences of L. The following theorem shows the importance of the least Herbrand model as all the atoms in M_L are precisely those that are logical consequences of the database.

Theorem 2.1: Let L be a logic database. Then $M_L = \{p\ (E\ H_b \mid L \mid= p)$

Proof: We have that
$p(t_1, ..., t_n)$ is a logical consequence of L
iff $L| = p(t_1, ..., t_n)$,
iff $L\ U\ \{\neg p(t_1, ..., t_n)\}$ has no model,
iff $L\ U\ \{\neg p(t_1, ..., t_n)$ has no Herbrand model,
iff $\neg p(t_1, ..., t_n)$ is false wrt all Herbrand models of L,
iff $p(t_1, ..., t_n)$ is true in all Herbrand models of L,
iff $p(t_1, ..., t_n)$ Œ M_L (LM(L))

If L contains the predicate symbol p, then the meaning or denotation d(p) is the relation associated with p by the Herbrand interpretation LM(L). In symbols,

$$d(P) = \{(t_1, ..., t_n) | p(t_1, ..., t_n) \text{ Œ } LM(L))$$

for any set of clauses in a database L.

Answer Substitutions

This section introduces the concept of answer substitution based on unification theory, which provides a declarative understanding of the desired output from a logic database and a goal.

Definition 2.11: A *substitution* q is a finite set of the form $(v_1/t_1, ..., v_n/t_n\}$, where
* each v_i is a variable
* each t_i is a term distinct from v_i
* all the v_i are distinct
* if v_i is a variable, t_i is a constant

Each element v_i/t_i is called a binding for v_i. q is called a pound substitution if the v_i are all ground terms.

Definition 1.12: An *expression* is either a term, a literal or a conjuction or a disjunction of literals. A simple expression is either a term or an atom.

Definition 2.13: Let $q = (v_1/t_1, ..., v_n/t_n)$ be a *substitution* and E be an expression. Then Eq, the instance of E by q, is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i $(i = 1..n)$

Definition 2.l4: Let L be a logic database and G be a goal. An *answer substitution* for L U {G} is a substitution for variables of G, that is a substitution whose domain is included in the set of variables of G.

Definition 2.15: Let L be a logic database, 6 be a goal ¨- $A_1, ..., A_k$ and q be an answer substitution for LU {G}. We say that q is a correct answer substitution for L U {G} if $((A_1\ L...\ L\ A_k)q)$ is a logical consequence of L.

THE TYPE SYSTEM IN LOGIC PROGRAMS

In this section, we will introduce the notion of types in logic programming. The notion of types has a long tradition in programming languages. R.Dietrich noted that most logic programming system has a good rapid prototyping quality, however the systems lack certain properties which are essential when developing large programs by several programmers [Dietrich 1988]. Alan Mycroft and Richard O'Keefe have made the first attempt and implemented on the application of a polymorphic type scheme to Prolog which makes static type checking possible [Mycroft 1983]. The type system is based on Milner's work on typing a simple

applicative language which is used in the ML type checker [Milner 1978]. Each call conforms to the type declaration of that object. Each construct in the language is associated with a type and using that information, well-typing is defined. They noted simply that the only additions to the language are type declarations, which an interpreter can ignore if it so desires, with the guarentee that a well-typed program will be have identically with or without type checking. Roland Dietrich and Frank Hagl enhanced Mycroft and O'Keefe's type checker to deal with polymorphic subtypes by adding modes to the program [Dietrich 1988].

Prateek Mishra on the other hand, has a different view of typing and type checking in Prolog where a type-inference system would detect goals which can never succeed [Mishra 1984]. The type of a predicate describes all terms for which the predicate may succeed, and for any term not described by the type of the predicate, the predicate cannot succeed.

Eyal Yardeni and Ehud Shapiro in their recent research in typ. system for logic programs [Yardeni 1991], have developed a theory of type system for pure logic programs which addresses the question of type declaration, type inference and well-typing. Their work was based on a restricted class of types, called regular types and sub-class of logic programs, called regular unary logic (RUL) programs, for which type checking is decidable. They noted that their theoretical model is a suitable basis for type systems for concrete logic programming languages. The notion of the following type system is primarily due to Eyal Yardeni and Ehud Shapiro [Yardeni 1991]. The formalisation of how to declare the type of a program by regular unary logic (RUL) program is shown.

Definition 3.1: Let S be a set of first-order formulas, and L a first-order language. The signature of S, sig(S), is the minimal set containing all predicates, function symbols and constant that appear in S. Similarly, we define the signature of L, sig(L), to be the signature of all formulas that can be constructed in L. A logic program defines a signature.

Defination 3.2: Let Hb be the Herbrand base of a program P. Define the mapping Tp : 2^{Hb} -> 2^{Hb} as follows : let I be a Herbrand interpretation. Then,

TP (I) = { A Œ H_b| A ¨B_1,..., B_n << C Œ P and B_1,.../B_n (E I}

Notation A $<<_L$ B means that A is a ground instances of B over L. Van Emden and Kowalski [Emden 1976] proved that the intersection of all Herbrand models for p is equal to Tp∅ w, where w is the first infinite ordinal, which is the meaning of the program denoted by d(P).

Definition 3.3: With each term t we associate a labelled tree, which we refer to as the ***associated tree*** of t. The edges and the leaves of the tree are labelled according to the construction rules as follows:

(1) if t is a constant or a variable symbol, then make a leaf and label it with t.
(2) ift = f(t_1, ..., T_n); f is of arity n. then:
   (a) make a new node associated with t
   (b) for all i Œ [1.. n] construct recursively the subtree associated with t_i, and draw an edge. labelled f(n.i), from the new node to the root of the subtree.

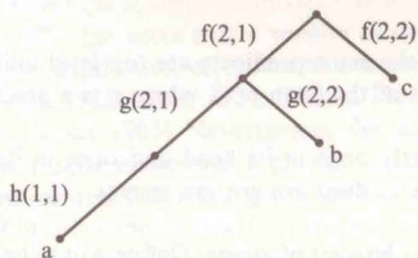

Figure 1 : The Associated Tree of f(g(h(a),b),c)

Definition 3 4: Let S be a set of ground terms. Define paths(S) = $U_{t\ Œ\ S}$ paths(t). The *tupple-distributive closure* of S is

a(S) = {t | t is a term and path s(t) C paths(S)}.

S is tupple-distributive if a(S) = S.

Intuitively, the tupple-distributive closure of a set of terms is the set of all terms constructed recursively by permuting each argument position among all terms that have the same functor-arity combination.

Example 3.1: If S = {f(a,b), f(c.d)). then paths(S) = (f(2.l)a, f(2,2)b. f(2,l)c. f(2,2)d) and a(S) = (f(a.b), f(a.d). f(c.b). f(c.d)).

Definition 3.5: Let p/n be a predicate of n-arity in a program P. Then the meaning of a programme P is

$d(P)_{p/n} = \{(p(t_1, ..., t_n)|p(t_1, ..., t_n)$ Œ d(P)).

We extend the notions of paths and tupple-distributivity to include atoms.

Claim: Let P be a program with different predicates $p_1, ..., P_n$. Then a(d(P)) = a(d(P)p_1) U... U a(d(P)pn)

Definition 3.6: A *type* is a recursively enumerable (r.e.) tupple-distributive set of ground atoms with a finite signature. The association of a type to a program is intended to mean that only ground atoms that are elements of the type may be derived from the program. It determines the set of permissable values an argument variable may take.

Definition 3.7: Let S be a programme. Then,

Ss, = (f(n,i) | f/n Œ sig (S),i (E (1,...,n}} U {c | c Œ sig(S)is a constant)

Definition 3.8: A set of ground terms S with a finite signature is regular iff there exists a regular language L C S_s* such that for every term t, t Œ S iff paths(t) C L. A logic program P is regular if d(P) is regular.

Lemma 3.1: For every regular set of terms S, paths(S) is regular.

The proof can be found in [Yardeni 1991].

Definition 3.9: Two terms are top-level *unifiable* if at least one of them is a variable or they have the same principle function symbol.

Definition 3.10: A *regular unary logic* (RUL) *programme* P is a logic program satisfying the following syntactic rules.

(1) Every predicate in P is unary.
(2) No two head arguments of clauses of the same predicate are top-level unifiable.
(3) Every body goal of every clause in P is of the form p(x), where p is a predicate name and x is a variable.
(4) Every variable in a clause occurs exactly once in its head and once in its body (note that the arguments off acts (clauses with empty bodies) are ground terms).

Definition 3.11: Let p be a unary predicate and A be a set of atoms. Define A/p to be the set {t | p(t) (E A}.

Theorem 3.1: (1) RUL programmes are regular.
(2) For every regular set of terms S there exists a RUL programme P with a predicate P such that S = d(P)/p.

Proof: The proof can be found in [Yar91].

Example 3.2: a simple logic programme with type declaration is as shown below,

Natural ::= O; s(O).
Procedure plus (Natural, Natural, Natural).

plus (O, X, X).
plus(s(X), Y, s(Z)) <- plus (X, Y, Z).

The semantic interpretation (or role) of each argument in a relationship defined by a predicate is given special attention. is We associate a type in a logic programme to data type with respect to the role of an argument. This in view that the naive users only communicate in terms of the real world definition. Every defined predicates in a database reflects the real world to be model and every argument in a relationship has a semantic role. We therefore view an argument at the schema level to posses a semantic property identified by an argument's (attribute's) identifier which reflect the meaning of the real world.

Definiton 3.13: Let $A_i :: a_{i,1}; a_{i,2}; ...; a_{i,n}$ and $A_j :: a_{j,1}; a_{j,2}; ...; a_{j,n} (i \pi j)$ be data types and $p(t_1, T_2$ be a predicate with a semantic interpretation as customer T_1 lives at address T_2. Let the arguments T_1 is of type Ai and t_2 is of type A_j. If t_1 and t_2 are ground terms, then t_1 Œ A_i and t_2 Œ A_j such that A_i and A_j are said to be defined as *data types* 'customer' and 'address' respectively.

The notion of data type is thus associated to an attribute's identifier in a relationship of a predicate. It may be viewed as an abstraction of meaning. Usually, a designer has in mind the type of arguments associated to a predicate which may be interpreted as the set of values defined by the database. That is, we can think of a predicate as true for its arguments if and only if those arguments form a tupple of the corresponding relation which is an element of a least Herbrand model.

Definition 3.14: Let $p(t_1, ..., t_n)$ be apredicate in a database L. Let M_L be the least Herbrand base of L. Then the meaning of a database w.r.t. the predicate p is,

$$d(L)_{p(t1,..,tn)} = \{< t_1, ..., t_n > |p(t_1, ..., t_n) << L, L| =_{ML} p(t_1, ..., t_n)\}$$

THE PREDICATE UNIVERSAL RELATIONS IN NATURAL QUERY PROCESSING

Introduction

At this point, we have introduce the concept of a logic database and the type system in logic programmes. In this section, we will discussed the concept of universal relation and demonstrate how it may be used as an interface in an attempt to offer the users a new version of query called a **natural query**. A lot of research have been conducted on the application of universal relation assumptions to relational database system [Fagin 1982, Kent 1981, Kuck 1982, Maier 1984, Ullman 1989]. There are positive and negative issues such as described in [Kent 1981, Brady 1985]. Nevertheless, the motivation of universal relation assumptions is very favourable to the user's point of view, though, the implementations imposed some restrictions [Codd 1990]. The researchers however feel that it is a good research to continue in order to explore new ideas and possibilities in the area of information retrieval. This view is supported by W. Kent, a researcher who has investigated the consequences of assuming universal relation [Kent 1981].

The universal relation model is first introduced as a means to free the users fnom the need to know the logical navigation of the database. The major objective is to achieve complete access-path independence, whereby data retrieval only requires the names of the attributes [Maier 1984, Ullman 1989, Korh 1984, Brady 1985, Kent 1981, Maier 1983b, Fagin 1982, Kuck 1982, Beeri 1982, Ullman 1982]. There are positive and negative views on the assumptions [Codd 1990a, Kent 1981]. Several versions of universal relation assumptions with respect to relational systems have been introduced to satisfy different objectives. A simple illustration of the notion of universal relation assumptions is as follows,

UR Assumptions :

For the set of relations $S = \{R_1 < X_1; D_1 >, R_2 < X_2; D_2 >, ..., R_n < X_n; D_n >\}$, there exist a Universal Relations $U < T; G >$ such that

(1) The columns of U consist of all the columns of the relations in S :

$$T = X_1 \cup X_2 \cup ... \cup X_n,$$

(2) Each relation in S is a projection of U :

$$R_i = U[X_i]$$

Currently, the form of query to the logic databases is very primitive. The naive users often find it difficult to use the database. We feel that the technical details should be hidden from the users and the system should be human-oriented, user-friendly and understand a close approximation to natural language. Based on the literature, there has been no attempt to investigate the universal relation approach in the query processing of logic databases. Therefore, it is the objective of this project to investigate and explore the concept of universal relation for the natural query interface to the logic database. In the subsequent sections we will introduce our version of universal relation assumptions called the predicate universal relation (PUR). PUR is based on the notion of object-structure universal relation [Maier 1983b] intoduced by D. Maier and J. D. Ullman. The concepts of object-structure universal relation and the FOPL have common sructural properties.

We believe that this is the first attempt in adopting universal relation approach to query processing of logic database. This is in view that both relational databases and logic databases have a common underlying mathematical model of first-order logic (FOL). However, it is worth distinguishing the differences between the two types of databases. Firstly, the existence of the rules in the logic databases which may be recursive, thus the relation described may be infinite in contrast to the traditional relational databases. Secondly, the use of variables and compound terms which appear in the rules of logic databases. Generally, given a query, there is also a difference in the process of answer substitutions with respect to the query [Nilsson 1990, Maier 1983a, Lloyd 1984, Kroenke 1983, Hogger 1990, Gallaire 1977, Deville 1990, Date 1986, Codd 1990a, Thayse 1988]. These differences influence the design and implementation of the intended universal relation assumptions which should be designed to be compatible with the characteristics and properties of the logic database system.

The Role of Predicate Universal Relation

At this point, it is important to distinguish the distinctions between the role of universal relation concept applied in relational and logic database systems. Informally, the differences are, firstly, the application of universal relation in relational databases involves actual relations (base relatrons) during the answer substitutions. On the other hand, the application of PUR does not concern with the base relation. It is only used as an abstraction between the database predicate schemes and the natural query (see Figure 2). Therefore, its application is only at the schema level. Secondly, since the universal relation deals with the base relation, it thus involves with the relational operations such as joins, union, difference, projection, selection and product. On the other hand, the PUR is implemented at the schema level, it does not involve with the relational operations. The role of the PUR act as an interface, that is, to serve as an intermediate expression of the natural query. Figure 2 illustrates the role of the universal relation.

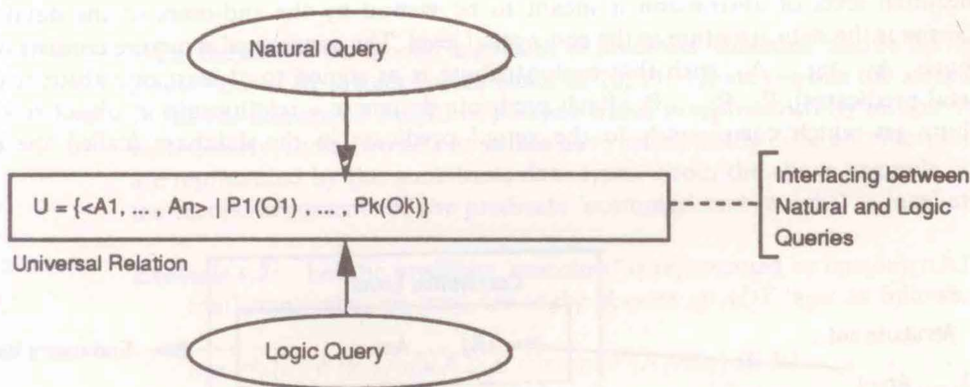

Figure 2: Role of Universal Relation in Query Processing

Defining Universal Relation by Predicates

This section demonstrates the definition of universal relation by predicate based on object-structure representation [Maier 1983b, Ullman 1990, Fagin 1982]. The universal relation consists of distinct attributes such that an object is defined in terms of a specific set of interrelated attributes designated by abstract predicates. The object-structure universal relation is due to D. Maier and J. D. Ullman.

Assuming a set of attributes $A_1, A_2, ..., A_n$, then a universal relation can be defined by predicate definitions as follows,

$$U = \{ < A_1, A_2, ..., A_n > \mid P_1 \text{ L } P_2 \text{ L} ... \text{ L } P_k\}$$

where each P_i is a predicate taking some set of the A_j, s as arguments.

If P_i involves $A_{j1}, A_{j2},..., A_{ji}$, then the set of attributes $R_i = \{A_{j1}, A_{j2},..., A_{ji}\}$ is said to be an object. Objects are meant to be the sets of attributes among which there is a significant connection.

Definition 4.1: If X C $A_1, A_2,..., A_n$, the connection among the attributes in X, denoted by [X], is defined by

$$[X] = P_X (U)$$

That is, [X] is the projection of the universal relation onto the attributes in X which corresponds to a predicate in the logic database.

Levels of Data Representation

At least three levels of data abstraction are recognised to exist in the specification of the database structure. The architecture is devided into three levels, known as the conceptual or the user's view (the **PUR**), the implementation or the designer's view and the physical level. Diagramatically illusrates the three level architecture of abstraction or representation of data in the system. The main purpose of the **PUR** is to provide users with an abstract view of the actual database structure. The separation of levels of abstraction largely depends on the distinction of views of data by the database users.

This paper will focus on the conceptual level of data abstraction, and how the conceptual view is mapped to the logical or implementation view of the data model. Data model is a representation of data and its interrelationships which describes ideas of the real world [Brodie 1984, Codd 1990a]. Physical level is below the implementation level which describes how data are actually stored in the physical storage devices and access techniques.

The conceptual level of abstraction is meant to be viewed by the end-users of the database. The conceptual schema is the data structure at the conceptual level. The conceptual structure consists of a set of distinct attributes, A_1, A_2,..., A_n, such that each attribute is as signed to at least one abstract predicate (called universal predicates), P_1, P_2,..., P_k. Each predicate designate a relationship of object relating to a specific attribute set which corresponds to the actual predicate in the database (called the database predicates).

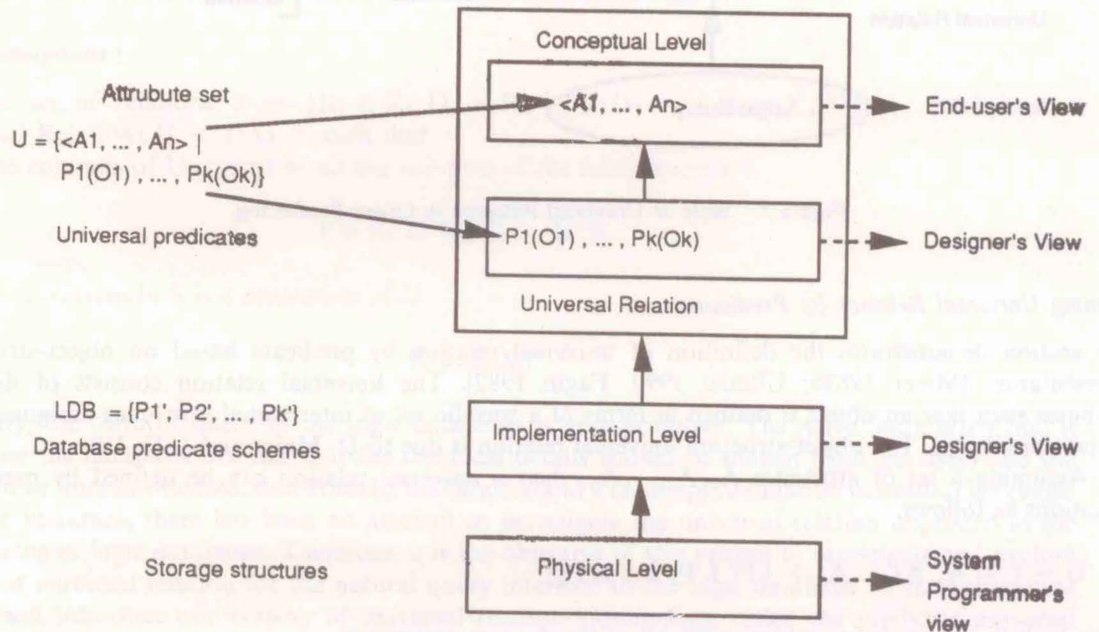

Figure 3: Levels of Abstraction Viewed by Database Users

The implementation scheme of the logic database is the actual database structures. At this level, there exists rules and facts which form the base relations. For each distinct predicate symbol that exist at this level, there is a corresponding predicate scheme at the conceptual level. Thus it form a unique pair of database-universal predicates. Both types of predicate schemes posses common properties which enable a mapping from the universal predicates to database predicates.

Domains Viewed as Declared Data Types

The concept of domain has played a very important role in the relational model since the model was introduced [Codd 1990a]. It is well accepted that the domain concept is fundamental. Similarly, the concept of domain is important as an interpretation mechanism of formulas in logic database. This section introduces the relationship between the concept of domains and data types. The main idea is to enable the reader to realise the connection between domains and abstract data types (ADT) and how they may be used to define the predicate universal relation.

An ADT is intended to capture some of the meaning of the data. If, however, two semantically distinguishable types of real-world objects or events happen to be represented by values of the same basic data type, it is thus requires the assignment of distinct attribute names to these types.

Example 4.1:

let us consider the following predicates,
customer(adam,10) to mean 'adam is a customer with age of 10', and
weight(adam,10) to mean 'adam has weight 10 stones'.

From the above, the second argument of predicate 'customer' shows the domain of ages of a persons which is represented by integer values (such as 10, 11, 13, etc.), while the second argument of predicate 'weight' shows the domain of weight of persons which is represented by integer values (such as 10, 13, 14, etc.) as well. Clearly, these two sets of values have semantically different interpretations or meanings, though they are represented by the same basic data types. From the above example, we could thus design an ADT for the second arguments of the predicate 'customer' and 'weight' as 'age' and 'weight' respectively.

Example 4.2: Let the predicate 'customer' is represented by customer(A1, A2) in a logic database L, then, mathematically, we may define the *domain of ADT* 'age' as follows,

age = (c | c (E Dom(A2) L customer (A_1, A_2) Œ L)

Similarly, every argument of the predicates defined in the database are treated in the same manner. We have demonstrated the relationship between domains and ADTs in a database.

Relationships Between Models, Domains and Data Types

At this point, we have discussed the concepts of nodels, data types and domains. In this section, we will demonstrate the relationships between the concepts of model (least Herbrand model), domains and ADT. The main reason for demonstrating the connections between these three concepts is to show how the application of ADT to PUR as a natural query interface could provide a means for checking database semantic integrity in order to achieve a certain level of data independence. Figure 4 illustrates diagramatically the connections between the concepts of database models and domains.

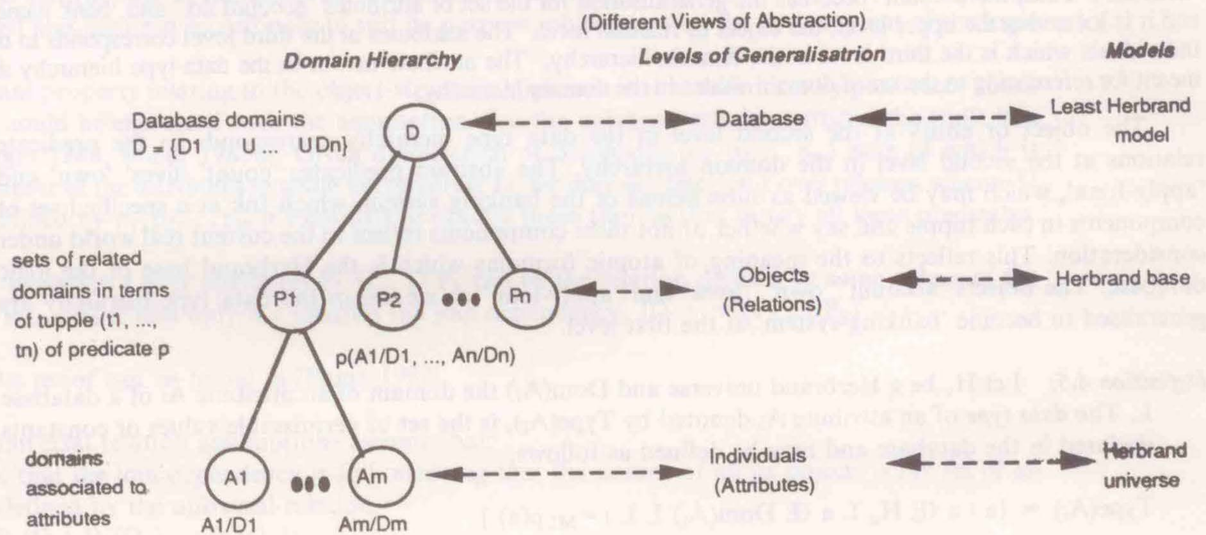

Figure 4: The Connections Between Domains and Models

The relationship between domains and models may be viewed at different levels of abstraction : database level, object level and individual level. As a result of generalisation and specialisation, a three level domain hierarchy may be consructed. The uppermost level shows a global view of database domain. The second level shows n different sets of domain values which are classified according to objects (predicates). The third level shows domain values classified according to arguments or attributes of various objects. From the illustration we could see that the Herbrand universe H_u, is a set of individuals which corresponds to the third level in the domain hierarchy. The Herbrand base corresponds to the second level which are the predicate relations.

Definition 4.2: Let H_u be a Herbrand universe, p(a) be a predicate in a database L and M_L be the least Herbrand model of the database. The ***domain*** of an attribute Ai (which correspond to a term. T_i) is a non-empty finite set of constants and may be defined as follows,

$Dom(A_i)$ = {a | a Œ H_u L L | = $_{ML_p}$(a)}

Definition 4.3: Let L be a database and Dom(L) be a set of database domains relative to the attributtes A1,.... An. The *Herbrand Universe* H_u of L is the set of all pound terms which can be formed out of the constants and functions appearing in L and may be defined as follows,

H_u = {a | a Œ Dom(L) L Dom(L) = {Dom(A1) U... U Dom(A_n)}}

Definition 4.4: Let L be a database. The *Herbrand base* of a database H_b, is the set of all ground atoms which can be formed by using predicates from L with ground terms from the Herbrand universe as arguments and may be defined as follows,

H_b = {p(t_1,....t_m) | (t_1,... t_m) Œ H_u L p((t_1,... t_m)) Œ L}

Let us now see how domains is associated to the concept of ADT. For the purpose of explaination, an
Let us now see how domains is associated to the concept of ADT. For the purpose of explaination, an example of the banking system is used to demonstrate the connection between the two concepts. We can observe that by using generalisation method [Brodie 1984], the abstraction of the banking system may be viewed as a hierarchic data model. The attribute level which is the lowest level in the data type hierarchy refers to the individuals. For example, 'account no.' and 'bank name' are attributes specialisation which are generalised to an object called 'account'. Thus, the'account' becomes the generalisation for the set of attributes 'account no.' and 'bank name', and it is located at the upper level, the object or relation level. The attributes at the third level corresponds to the individuals which is the third level in the domain hierarchy. The attribute names in the data type hierarchy are meant for referencing to the set of domain values in the domain hierarchy.

The object or entity at the second level in the data type hierarchy corresponds to the predicate relations at the second level in the domain hierarchy. The abstract predicates 'count' 'lives' 'own' and 'apply-loan', which may be viewed as subschemas of the banking system, which Ink at a specified set of components in each tupple and say whether or not these components reflect to the current real world under consideration. This reflects to the meaning of atomic formulas which is the Herbrand base of the logic database. The objects 'account' 'own', 'lives' and 'apply-loan' as we see in the data type hierarchy are generalised to become 'banking-system' at the first level.

Definition 4.5: Let H_u be a Herbrand universe and Dom(A_i) the domain of an attribute Ai of a database L. The *data type* of an attribute A_i denoted by Type(A_i), is the set of permissable values or constants declared in the database and may be defined as follows,

Type(A_i) = {a | a Œ H_u L a Œ Dom(A_i) L L | = $_{ML}$p(a) }

We have now demonstrated how the relationships between the three concepts; models, domains and ADTs. In the subsequent sections, the application of ADTs to the universal relation will be discussed by demonstrating how it may be used as a tool for construction of the PUR.

The Predicate Universal Relation Assumptions

At this point, we have understand the role of data types. In this section, we will formalise on the application of types on the new version of universal relation assumptions, called predicate universal relation (PUR) assumptions, which is aim at achieving its objective as an interface in natural query processing of a logic database.

There are two known approaches which have been proposed for defining universal relation [Fagin

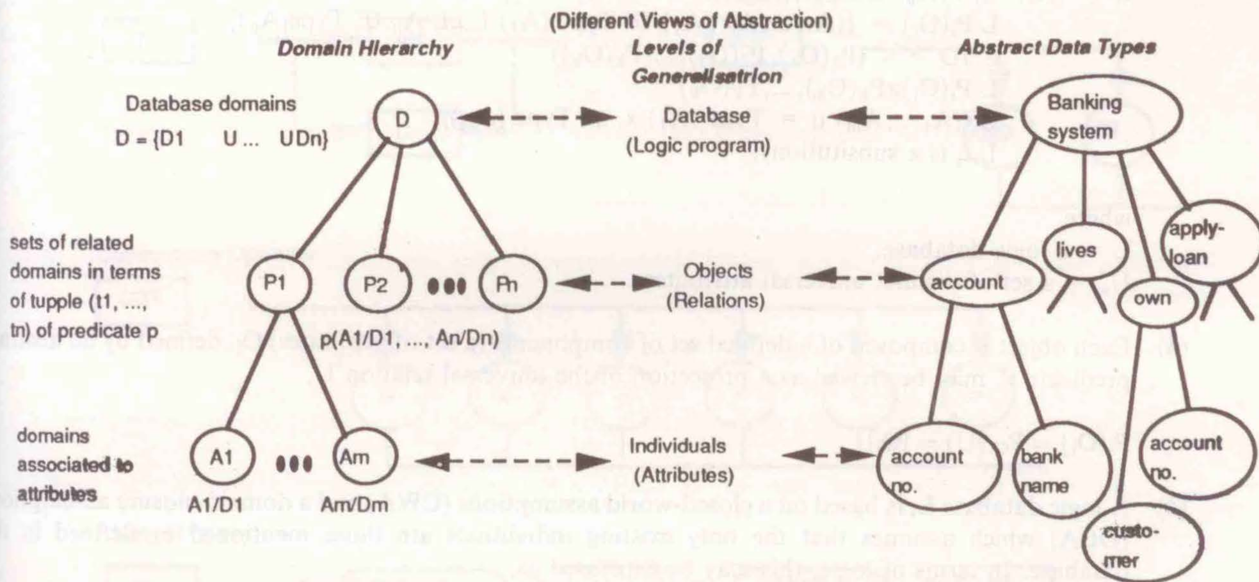

Figure 5: The Connection Between Domains and Abstract Data Types

1982, Maier 1983b, Ullman 1989, Maier 1984]. For our purpose, the object-structure approach seems to be the most suitable representation since this particular approach posseses common charateristics of predicate logic. In order to suit the role of the universal relationas an interface to logic query, some of the assumptions are reviewed and taylored accordingly to suit its purpose, objectives and the environment of the logic database system.

An important property relating to the object-structure approach is that definition by predicates about the 'real world' could be expressed with the assumption that the universal relation satisfies the (full) join dependency [Fagin 1982, Maier 1983b]. Given a collection of predicates ($p_1, p_2, ..., p_m$) each of which is defined over a subset of the attributes in some universal set U, we may say relation r over relation scheme U is the relation defined by $p_1, p_2, ..., p_m$ if r consists of exactly those tupples that satisfy all these predicates.

Theorem 4.1: A relation r over attributes P_1 U... U P_k can be the relation defined by some values of the predicates $P_1, ..., P_k$ if and only if r satisfies the join dependency, JD > < ($P_1, ..., P_k$)

Proof: The proof can be found in [Fagin 1982].

The predicate universal relation assumptions assume that:

(1) We assume that the join dependency is full, meaning that the unions of all its objects is the set of all attributes defined by the universal relation,
JD > < ($P_1(O_1), P_2(O_2), ..., P_k(O_k)$)

(2) We assume that every attributes set Oi = ($A_1, ..., A_m$) belonging to a predicate symbol Pi, there exist a data type corresponding to each Ai, which may be defined as follows,
$P_i(0_i) = (A_1, ..., A_m)|A_1$ Œ Type (A_1)L....LA_m Œ Type (Am)}

(3) For the set of object relationships (defined by predicates) 5 = {$P_1(O_1), P_2(O_2), ..., P_k(O_k)$}, where each P_i(0 < i £ k) is a predicate relation scheme, each O_i(O < i £ k)} is an object which is characterised by the properties of a set of typed attributes (A_1 (E Type(A_1) L...L A_m Œ Type(A_m)), and there exists a full join dependency such that defined in (1), then there exists a predicate universal relation scheme which may be defined as follows,

$U = \{A_1, A_2, ..., A_n$ Œ $U_a | P_1(O_1)$ L... L $P_k(O_k)$ Œ sig(L) L O_i C U_a
L $P_i(O_i) = \{(A_1, ..., A_m) | A_1)$ Œ Type (A_1) L...L A_n Œ Type$(A_n)\}$
L JD > < $(P_1(O_1), P_2(O_2), ..., P_k(O_k))$
L $P_i(O_i) \pi P_k(O_k), ..., P_k(O_k)$
L $((A_1, .., A_m)$ q = Type (A_1) x...,x Type $(A_m))$
L q is a subsitution)}

where,
L = a logic database.
U_a = a set of distinct universal attributes.

(4) Each object is composed of a defined set of components (a set of attributes) O_i, defined by an abstract predicate P, may be viewed as a projection of the universal relation U,

$P_i(O_i) = P_{Oi}(U) = [O_i]$

(5) A logic database L, is based on a closed-world assumptions (CWA) and a domain-closure assumptions (DCA) which assumes that the only existing individuals are those mentioned or defined in the database. In terms of logic, this may be expressed as,

$(Vx)(x = c_1 Vx = c_2 V...Vx = c_n)$

where $c_1, c_2, ..., c_n$ are all constants occurring in the database.

(6) A logic database *, is based on a normal logic program P, which consists of a finite set of normal clauses, $C_1, C_2, ..., C_k$, and each clause has the form H ¨$B_1, B_2, ..., B_n$ (1 £ n)

where H and B_i are of predicate structures (consisting no functors, i.e. datalog). Each predicate structure is of the form $p(t_1, t_2, ..., t_n)$ where p is a predicate symbol and $t_1, t_2, ..., t_n$ are terms

(7) The (valid) natural queries of the logic database are those that corresponds to the existing attributes and predicates defined in the database, no new attributes with the exception of synonyms. No predicates and rule heads are generated except those defined in the database.

QUERY LANGUAGE

Logic offers a uniform paradigm for software technology, i.e. one formalism which serves for constructing and manipulating programmes, databases and software tools. The only drawback is that it requires the user to know the structure of the database. This definitely cause a burden to the naive users. We therefore attempt to introduce a query language which hide the user from the database structure based on PUR called natural query language.

Natural Query

Natural query is a query to the logic database based on PUR scheme as an interface. The form of queries in the PUR environment is based on QUEL.like notation [Maier 1983, Ullman 1989], with the exception that it contains no range-statements. This is because all the tupple variables would range over the universal relation. The format of queries that is used in this project is as follows:

RETRIEVE (attribute set)
WHERE (condition set)

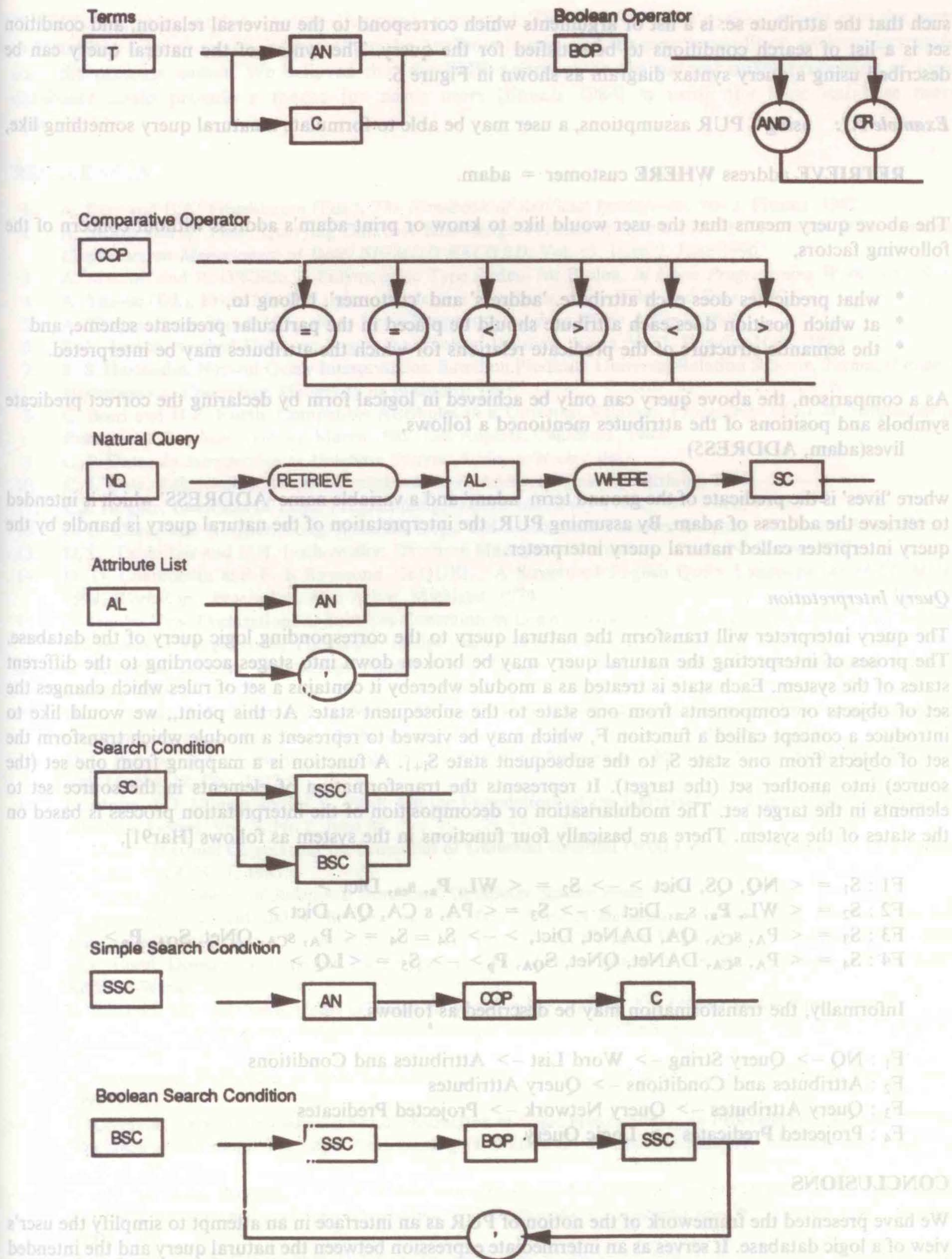

Figure 6: Query Syntax Diagram

such that the attribute se: is a list or arguments which correspond to the universal relation, and condition set is a list of search conditions to be satisfied for the query. The syntax of the natural query can be described using a query syntax diagram as shown in Figure 6.

Example 5.1: using a PUR assumptions, a user may be able to formulate a natural query something like,

RETRIEVE address **WHERE** customer = adam.

The above query means that the user would like to know or print adam's address without concern of the following factors,

* what predicates does each attribute. 'address' and 'customer', belong to,
* at which position does each attribute should be placed in the particular predicate scheme, and
* the semantic structure of the predicate relations for which the attributes may be interpreted.

As a comparison, the above query can only be achieved in logical form by declaring the correct predicate symbols and positions of the attributes mentioned a follows,

lives(adam, ADDRES5)

where 'lives' is the predicate of the ground term 'adam' and a variable name 'ADDRESS' which is intended to retrieve the address of adam. By assuming PUR, the interpretation of the natural query is handle by the query interpreter called natural query interpreter.

Query Interpretation

The query interpreter will transform the natural query to the corresponding logic query of the database. The proses of interpreting the natural query may be broken down into stages according to the different states of the system. Each state is treated as a module whereby it contains a set of rules which changes the set of objects or components from one state to the subsequent state. At this point,, we would like to introduce a concept called a function F, which may be viewed to represent a module which transform the set of objects from one state S_i to the subsequent state S_{i+1}. A function is a mapping from one set (the source) into another set (the target). It represents the transformation of elements in the source set to elements in the target set. The modularisation or decomposition of the interpretation process is based on the states of the system. There are basically four functions in the system as follows [Har91],

F1 : S_1 = < **NQ**, QS, Dict > –> S_2 = < WL, $\mathbf{P_a}$, $\mathbf{s_{ca}}$, Dict >
F2 : S_2 = < WL, $\mathbf{P_a}$, s_{ca}, Dict > –> S_3 = < PA, s CA, **QA**, Dict >
F3 : S_3 = < P_A, s_{CA}, **QA**, DANet, Dict, > –> S_4 = S_4 = < P_A, s_{CA}, QNet, $\mathbf{S_{QA}}$, $\mathbf{P_p}$ >
F4 : S_4 = < P_A, $\mathbf{s_{CA}}$, DANet, QNet, $\mathbf{S_{QA}}$, $\mathbf{P_p}$ > –> S_5 = < **LQ** >

Informally, the transformation may be described as follows,

F_1 : NQ –> Query String –> Word List –> Attributes and Conditions
F_2 : Attributes and Conditions –> Query Attributes
F_3 : Query Attributes –> Query Network –> Projected Predicates
F_4 : Projected Predicates –> Logic Query

CONCLUSIONS

We have presented the framework of the notion of PUR as an interface in an attempt to simplify the user's view of a logic database. It serves as an intermediate expression between the natural query and the intended logic query. We have also demonstrated how the ADTs may be used as a tool in the construction and definition of PUR. Global attributes may be identified based on the semantics and the role played by the arguments of the predicates. We have introduced our version of a query language using the RETRIEVE...

WHEPE...statement and the syntax diagram is presented. This form of query syntax, would ease the users to formulate a query simply by providing the attribute names and the search condition without having to state the prdicate names. We believed that the PUR approach to the natural query processing of logic databases could provide a means for naive users [Ennals 1984] in using the logic database more comfortably.

REFERENCES

1. A. Barr and E.A. Fiegenbaum (Eds.), *The Handbook of Artificial Intelligence*, Vol.2, Pitman, 1982.
2. A. Motro and Q. Yuan, Querying Database Knowledge, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD RECORD*, Vol. 19, Issue 2, June 1990.
3. A. Mycroft and R. O'Keefe, A Polymorphic Type System for Prolog, *In Logic Programming Workshop*, 1983.
4. A. Thayse (Ed.), From Modal Logic to Deductive Databases, *John Wiley and Sons*, 1989.
5. A. Thayse, From Standard Logic to Logic Programming, *John Wiley & Sons*, 1988.
6. B. E. Jacobs, Applied Database Logic: *Fundamental Database Issues*, Vol.1, Prentice-Halls, 1985.
7. B. S. Harihodin, Natural Query Interpretation Based on Predicate Universal Relation Scheme, *Technical Report, Department of Computing*, University of Bradford, 1991.
8. C. Beeri and H.F. Korth, Compatible Attributes in a Universal Relation, *Proceedings of ACM Symposium on Principle of Database Systems*, March 1982, Los Angeles, California, 1982.
9. C. J. Date, *An Introduction to Database System*, Addison-Wesley 1981.,
10. C. J. Date, *Relational Database* (selected writings), Addison-Wesley Publishing, 1986.
11. C.J. Hogger, Essentilas of Logic Programming, *Oxford University Press*, New York, 1990.
12. C. L. Chang and R. Char..Tong, Symbolic Logic and Mechanical Theorem Proving, *Academic: Press*, 1973.
13. D. C. Tsichritzis and F.H. Lochovosky, Database Management, *Academic Press*, N. Jersey, 1977.
14. D. D. Chamberlin and F. B Raymond, SEQUEL : A Structured English Query Language, *ACM-SIGMOD, 1974, Workshop - Proceeding*, Ann Arbor, Michigan, 1974.
15. D. Jacobs, Type Declarations as Subtypes Consraints in Logic Programming, *Proceedings of ACM SIGPLAN'90 Conference on Programming Languages, Design and Implementations*, Jun 1990.
16. D. Kroenke, *Database Processing*: Fundamentals, Design and Implementation, Science Research Institute, Sydney, 1883.
17. D. L. Parnas, J. E. Shore and D. Weiss, Abstract Types defined as Classes of Variables, *Proceedings of the Conference on Data Abstraction, Definition and Structure, SIGMOD F.D.T.*, Vol.8, No.2, 1976.
18. D. L. Parnas, On the Criteria to be Used in Decomposition Systems into Modules, *Communication of the ACM*, Vol. 15, No. 12, Dec. 1972.
19. D. Maier, J.D. Ullman and M.Y. Vardi, On the Foundations of the Universal Relations, *ACM Transaction on Database System*, Vol. 9, No.2, June 84.
20. D. Maier, Maximal Objects and the Semantics of Universal Relation Databases, *ACM Transaction on Database Systems*, Vol.8, No.1, 1983.
21. D. Maier, *The Theory of Relational Databases*, Computer Science Press,1983.
22. D. Stemple, T. Sheard and R. Bunker, *Abstract Data Types in Databases* : Specification, Manipulation and Access, In Object-Oriented Database Systems (S.B. Zdonik and D. Maier, Eds.), Morgan Kaufmann Publ., 1990.
23. E. F. Codd, Domains as Extended Data Types, *In The Relational Model for Database Management* : Version 2, Addison-Wesley,1990.
24. E. F. Codd, *The Relational Model of Database Management, Version 2*, Addison-Wesley Publ., 1990.
25. E. Yardeni and E. Shapiro, A Type System for Logic Programs, *In Concurrent Prolog*, Vol. 2, *M.I.T. Press* (E. Shapiro, Ed.), 1987.
26. E. Yardeni and E. Shapiro, A Type Systems for Logic Programs, *Journal of Logic Programming*, Vol. 10, No.2, 1991, 1991.
27. G. Dayantis, Types, Modularisation and Abstraction in Logic Programming, *In Proceedings of 1st. International Workshop on Algebraic and Logic Programming* (J.Grabowski, P. Lescanne and W. Wechler, Eds.), Gaussig, GDR, 1988.
28. G. Smolka, Logic Programming with Polymorphically Oeder-Sorted Types, *In Proceedings of 1st. International Workshop on Algebraic and Logic Programming* (J.Grabowski, P. Lescanne and W. Wechler, Eds.), Gaussig, GDR, 1988.
29. H. F. Korth and A. Silberschatz, Database System Concepts, MacGraw Hill, 199l. [Kow79] R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.
30. H. F. Korth, G.M. Kuper, J. Feigengaum, A. Gelder, J.D. Ullman, System/U: A Database System Based on the

Universal Relation Assumptions, *ACM Transactions on Database Systems*, Vol.9, No.3, Sept 1984.

31. H. Gallaire, J. Minker and J.M. Nicolas, An Overview and an Introduction to Logic and Databases, *In Logic and Databases* (H. Gallaire and J. Minker, Eds.), Plenum Press, 1977.
32. H. W. Beck and 5. Navathe, Integrating Natural Language, Query Processing and Semantic Data Models, *COMCON 5PRING 90-PROC IEEE CONFERENCE*, 1990.
33. J. Bishop, *Data Abstraction in Programming Languages*, Addison-Wesley Publ., 1986.
34. J. D. Ullman and C Zaniolo, Deductive Databases : Achievements and Future Directions, *SIGMOD RECORD*, Vol.19, No.4, December 1990.
35. J. D. Ullman, Principles of Database and Knowledge-Based Systems, Vol. 1 and Vol.2, *Computer Science Press*, 1989.
36. J. D. Ullman, Principles of Database Systems, *Computer Science Press*, 1983.
37. J. D. Ullman, The U. R. Strikes Back, *Proceedings of ACM Symposium on Principle of Database Systems*, Los Angeles, California, March 1982.
38. R. Kowalski, *Logic for Problem Solving*, North-Holland, 1979.
39. J. J. MArtin, *Data Types and Data Structures*, Prentice-Halls, 1986.
40. J. Martin, *Introduction to Database Organisation*, Prentice-Halls, 1978.
41. J. V. Guttag, E. Horowitz and D. R. Musser, The Design of Data Type Specifications, *Proceedings Conference on Software Engineering*, 1976.
42. J. V. Guttag, The Specification and Application to Programming of Abstract Data Types, *Computer Systems Research Report*, CSRG-59, Univ. of Toronto, Dept. of Comp. Science, 1975.
43. J. W. Lloyd, An Introduction to Deductive Database Systems, *Australian Computer Journal*, Vol. 15, No.2, May 1983.
44. J. W. Lloyd, *Foundation of Logic Programming*, Springer-Veriag, 1984.
45. J. W. Schmidt, *Type Concepts for Database Definition, In Databases* : Improving Usability and Responsiveness (Shneiderman, Ed.), Academic Press, 1978.
46. L. I. Brady, A Universal Relation Assumption Based on Entities and Relationships, *In Entity-Relationship Approach* (P.P. Chen, Ed.), IEEE CS Press/North Holland, 1985.
47. M. Cogolla, A Note on The Translation of SQL to Tupple Calculus, *SIGMOD RECORD, Vol 19*, No.1, ACM Press, March 1990.
48. M. H. Emden and R. Kowaiski, The Semantics of Predicate Logic as Programming Languages, *Journal of ACM*, Vol.23, No.4, 1976.
49. M. L. Brodie and 5. N. Zilles (Eds), *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling*, ACM, Jun 23-26, Pingree Park, Colorado, 1980.
50. M. L. Brodie, On The Development of Data Models, In On Conceptual Modelling: *Perspective from Artificial Intelligence, Databases and Programming Languages*, Springer-Verlag, 1984.
51. M. L. Brodie, The Application of Data Types to Database Semantic Integrity, Information Systems, Vol.5, Pergamon Press, 1980.
52. P. Mishra, Towards a Theory of Types in Prolog, *IEEE International Symposium on Logic Programming*, 1984.
53. R. Dietrich and F. Hagl, A Polymorphic Type System with Subtypes for Prolog, *Proceedings of the 2nd. European Symposium on Programming Languages*, Nancy, France, 1988.
54. R. Ennals, J. Briggs and D. Brough, What the Naive User Wants From Prolog, *In Implementations of Prolog* (J. Cambell, Ed.), Ellis Horwood Ltd, 1984.
55. R. Fagin, A.O. Mendelzon and J.D. Ullman, A Simplified Universal Relation Assumptions and Its Properties, *ACM Transactions on Database Systems*, Vol. 7, No.3, Sep. 1982.
56. R. Milner, A Theory of Type Poiymorphism in Programming, *Journal of Computer Science*, Vol. 17, No. 3, 1978.
57. S. A. Naqvi,1986. A Brief Survey of Logic and Database System, In On Knowledge Base Management Systems: *Integrating Artificial Intelligence and Database Technologies* (M.L. Brodie and J. Mylopoulos, Eds.), Springer-Verlag, 1986.
58. S. M. Kuck and Y. Sagiv, A Universal Relation Database System Implemented via The Network Model, *Proceedings ACM Symposium on Principle of Database Systems*, Los Angeles, March 1982.
59. U. Nilsson and J. Matuszynski, *Logic, Programming and Prolog*, John Wiley and Sons, 1990.
60. W. F. Clocksin and C.S Mellish, *Programming in Prolog*, Springer-Verlag, 1984.
61. W. Kent, Consequences of Assuming Universal Relation, *ACM Transaction on Database System*, Vol.6, No.4, Dec. 1981.
62. Y. Deville, 1990. *Logic Programming* : Systematic Programme Development, Addison-Wesley, 1990.