Vol.10 No.2 J. of Comput. Sci. & Technol. March 1995

Translational Semantics for a Conceptual Level Query Language

Hock C. Chan

Department of Information Systems and Computer Science, National University of Singapore, Lower Kent Ridge Road, Singapore 0511
(*Email: chanhc@iscs.nus.sg*)

Received June 1, 1993; revised November 5, 1993.

Abstract

A conceptual level database language for the entity relationship (ER) model implicitly contains integrities basic to ER concepts and special retrieval semantics for inheritances of attributes and relationships. Prolog, which belongs to the logical and physical level, cannot be used as a foundation to directly define the database language. It is shown how Prolog can be enhanced to understand the concepts of entities, relationships, attributes and is-a relationships. The enhanced Prolog is then used as a foundation to define the semantics of a database query language for the ER model. The three basic functions of model specification, updates and retrievals are defined.

Keywords: Conceptual level, query language, semantics, logic, entity-relationship model, abstraction level, ER calculus.

1 Introduction

The entity-relationship (ER) model has become widely used for database design, accompanied with many proposed query languages. It is important to provide a precise definition of ER languages for the purposes of comparison and implementation. A formal logic-based definition of ER languages is, however, quite elusive because the semantics is situational. It depends on the actual data values and the constraints existing in the database. Consider the case where the ER statement is to delete a supplier. The meaning can be simply to delete that supplier if that supplier is not linked to other entities. If the supplier is linked to other entities, then these linkages (relationship instances) will also have to be deleted. The situation can be complicated by constraints on the cardinalities of relationships. For example, if every part must be supplied by at least one supplier, and at the time the supplier is to be deleted, then the parts that are supplied only by him must also be deleted. This can, in turn, lead to other deletions.

It is advantageous to use Prolog as the foundation to define ER languages. Since implementations of Prolog are widely available, Prolog-based definitions of ER languages can be easily checked and tested. However, Prolog belongs to the physical and logical level of abstraction, without any understanding of entities and relationships. In contrast, the ER model belongs to the conceptual level where there is no

need to specify logical level details such as foreign keys for relationships[1,2]. How the entity instances are traced to their relationship instances should be left to the particular implementation — whether by keys, pointers or surrogates. If the Prolog foundation uses keys for the relationship, this will bound the ER language to the use of keys as well, which is an unnecessary restriction. Hence, certain additions must be made to Prolog to allow the direct definition of conceptual level information.

The enhancement of Prolog is described in Section 2. Section 3 illustrates the definition of an ER language based on the enhanced Prolog. Section 4 summarizes the advantages of this approach of language definition.

2 Conceptual Level for Prolog

2.1 Abstraction Levels

Abstraction levels are used to focus on the desired level of information. For example, users of information systems will not want to know how the systems are implemented, what are the various modules and the data structures normally. On the other hand, maintenance programmers will need to know the modules and exact codes. Three levels of abstraction are commonly used in information systems and database research. These are the physical, logical and conceptual levels[1,3,4] as shown in Fig.1.

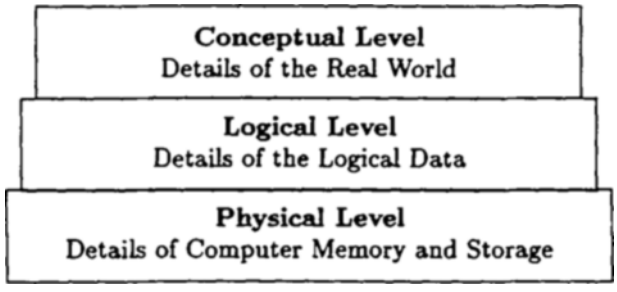

Fig.1. Abstraction levels.

The physical level is the closest to the computer, dealing with computer memory and storage. Its information is about data structures in the computer memory, the physical pointers, indexes and files. Prolog's need to distinguish the order of arguments is an example. If a predicate has ten arguments, then the user must remember the correct order. Some Prolog systems also allow access to other physical level details such as hash tables and B-trees. The logical level has information about logical data. A good example is the relational model[2,5] where the physical implementation of relational tables is of no significance (except for runtime efficiency). Linkages among data are made logically through keys and foreign keys instead of physical pointers. Prolog is also good at the logical level. In fact, some Prolog predicates are very similar to relations. For example, if supplier and part information is stored in the following predicate structures:

supplier(name, sno, address, industry)

part(name, pno, description)
supplier_part(sno, pno, price)

then retrieval of the suppliers of, say, tyres can be done with the following predicates:

part(tyre, P, _) , supplier_part(S, P, _), supplier(Name, S, _, _).

The above example illustrates the close similarity between Prolog and the relational model. Relationships are defined based on the collection of the keys of the entities. Retrieval is done using the same Prolog variable in two predicates to perform the relational join. This is similar to the normal relational joins that have to be specified in relational query languages such as SQL or QUEL.

The conceptual level's information is about objects (or entities) in the real world. Relationships are not specified based on keys and foreign keys, but are naturally specified between entities, such as *John marries Jane* or *Company employs employees*. More advanced concepts are possible at the conceptual level. These include is-a relationships, aggregation, and relationship among relationships. At the conceptual level, attribute and relationship inheritances are implicit. It is possible to say, e.g., *employee's address* instead of the *address of persons who are employees*, assuming that *person* is the super-entity of *employee*. At the conceptual level, integrity maintenance regarding entities and relationships should be automatic. For example, deleting an entity should lead to deletion of its relationships. This level of abstraction is not present in Prolog. Prolog does not understand entities, relationships, or inheritance. A programmer has to convert conceptual level information into logical and/or physical level information and code that in Prolog. Integrity rules that are implicit in the conceptual level information must be explicated and separately coded.

Fig.1 also indicates with the bigger rectangles that the lower levels contain more details than the higher levels. This is so if no information from the higher level is lost. Then more information about logical keys/foreign keys or physical pointers and indexes must be added.

2.2 Enhanced Prolog (ERLOG)

The additional predicates needed to make Prolog understand the meaning of entities and relationships are described below. The resultant product (Prolog plus these additional predicates), called ERLOG, contains ER knowledge so that it can maintain ER integrities and perform inheritances automatically. Conceptual information can be directly defined based on ERLOG. Furthermore, implicit integrities at the conceptual level need not be explicated. One characteristic of ERLOG is the emphasis on atomic facts of the ER model, i.e., the instances, attribute values and relationships. Unlike other systems, there are no complicated operations whereby, e.g., a group of instances becomes an attribute value of an instance.

The predicate notation follows that of Prolog. Briefly, a "+" in front of an argument means an instantiated argument, "−" means an uninstantiated argument, and "?" means either of them. The descriptions for the predicates show the primary intention. However, with automatic maintenance of integrities, secondary actions

may occur. For example, *d_i* is a predicate to delete an instance. This delete is the primary intention. Deletion of one (entity or relationship) instance can lead to deletion of other instances in order to maintain integrity. These secondary actions are described in the subsection on ER integrities.

Model Predicates

The following predicates are used to define and modify the concepts in an ER model. These include concepts of entity type, relationship type, entity attribute, relationship attribute, is-a relationship, cardinalities and roles. Predicates to query the ER model are included so that programmers can ascertain what has been specified. There are many versions of the ER model[6,7]. The specific concepts of the ER models that are supported are described in this section.

The following predicates beginning with *a_* are used to assert specific ER concepts, such as an entity *supplier*, a relationship *supply*, or an attribute *qty* of relationship *supply*.

a_e$(+T)$ — assert an entity type T, e.g., *a_e*(*supplier*).

a_a$(+T, +A)$ — assert attribute A for the type T, which can be an entity or a relationship, e.g., *a_a*(*supplier*, *name*).

a_key$(+T, +KeyList)$ — assert the key for the entity T. The key can be composite. This is used to check that no two instances have the same key values. The relationships are formed by the system using surrogates rather than these defined keys.

a_r$(+RT, +ETRoleCardList)$ — assert a relationship type T and a list of participating entity types with given roles and given lower and upper cardinalities. The cardinalities show the minimum and maximum relationship instances that the entity instance can have. An example is *a_r*(*supply*, [[*supplier*, *supplies*, $[0, n]$], [*part*, *supplied_by*, $[1, n]$]]), which states that a relationship supply exists between supplier and part. The roles of supplier and part are respectively *supplies* and *supplied_by*. A supplier can supply 0 or more parts, while a part must be supplied by at least 1 supplier.

a_isa$(+E1, +EList, +CharList)$ — assert an is-a relationship with a set of characteristics in *CharList*. The characteristics are t (total), p (partial), o (overlapping) and d (disjoint). An is-a relationship cannot be both total and partial, or both disjoint and overlapping. A total condition means that every instance of the supertype ($E1$) must appear in at least one of the subtypes (a member of *EList*). A partial condition does not impose this restriction. A disjoint condition means that any instance of the supertype cannot be a member of more than one subtype. On the other hand, an overlapping condition allows that. An example is *a_isa*(*employee*, $[full-time, part-time]$, $[d, p]$), which states that *employee* is the supertype of two subtypes *full-time* and *part-time*. This is-a relationship is disjoint and partial.

The following predicates beginning with *d_* are used to delete parts of the ER model, such as deleting an entity *supplier* or an attribute *name* of entity *part*.

d_t$(+T)$ — delete a type T.

$d\_a(+T, +A)$ — delete an attribute A that belongs to type T.

The following predicates beginning with $q\_$ are used to query the ER model, such as confirming the existence of entity *supplier* or an attribute *name* of entity *part*.

$q\_e(, ?T)$ — query the existence of an entity type.

$q\_r(?T)$ — query the existence of a relationship type.

$q\_a(?T, ?A)$ — ask for a type and the attributes of that type, including the inherited attributes.

$q\_ro(?RT, ?ET, ?Role)$ — ask about relationship participation.

$q\_isa(?Charlist, ?E, ?E2)$ — ask if $E2$ is a subtype of E, where $CharList$ is the characteristic list.

$q\_isalinked(?E1, ?E2)$ — ask if $E1$ and $E2$ are is-a related, i.e., if $E1$ and $E2$ are within the same is-a hierarchy.

Instance Predicates

Once an entity or a relationship type is specified, its instances can be added, modified or deleted. The following predicates beginning with $a\_$ are used to assert new instances and details of existing instances, either entity or relationship instances.

$a\_i(+T, -I)$ — assert an instance of T, which can only be an entity.

$a\_v(+I, +AV)$ — assert an attribute-value pair for the instance I which can be an entity or a relationship. An example is: $a\_v(I, [name, tom])$. I should be already instantiated. No attribute values can be given to uninstantiated instances.

$a\_re(+RI, +EIRoleList)$ — assert participation of entity instances with given roles in the relationship instance.

The following predicates beginning with $d\_$ are used to delete existing instances or their details.

$d\_i(+I)$ — delete the instance I, which can be an entity or a relationship.

$d\_v(+I, +A)$ — delete an attribute value of an instance.

The following three basic predicates beginning with $q\_$ are used to retrieve instances of entities or relationships.

$q\_i(?T, ?I)$ — with at least one argument instantiated.

$q\_v(+I, ?A, ?V)$ — confirm that attribute A of I has value V.

$q\_re(+RI, ?EIRoleList)$ — confirm that RI involves EIs with the stated roles, e.g., $q\_re(R, [[P1, father], [P2, mother]])$ queries if $P1$ is a father and $P2$ is a mother in the relationship instance R.

More complex queries are formed by combining these basic predicates with other Prolog predicates. These will generally involve comparison operators and list manipulations.

ER Integrities

The ER model has a set of inherent integrity constraints, which are automatically maintained by this logic system. The specific constraints are as follows.

- Entity and relationship names are unique. This is maintained by a_e and a_r predicates.

- The *a_r* predicate checks that multiple roles of an entity in a relationship are named differently.
- Attribute names are unique within an entity, a relationship, and an is-a hierarchy.
- The *a_v* predicate checks that key values are not duplicated.
- A relationship instance cannot exist without the existence of the involved entity instances. The predicates that maintain this integrity are *a_re* and *d_i*.
- A relationship type cannot exist if the involved entity types do not exist. The predicates to maintain this constraint are *a_r* (asserting a relationship type) and *d_t* (deleting a type).
- All instances can exist only if the type exists. This is maintained by the *a_i* (asserting an instance) and *d_t* (deleting a type) predicates.
- All subtype instances must exist in the supertype. The *a_i* and *d_i* predicates ensure this.
- Cardinality constrains an entity instance to participate a minimum and maximum number of times in a relationship. The predicates for this constraint are *a_i*, *a_re* and *d_i*.

These predicates for ERLOG have been implemented with Arity Prolog on the microcomputer. Implementation details are given in [8]. The source codes, together with the help facilities, take less than 60K of disk space.

3 An ER Database Language—KQL

One ER language, the knowledge level query language (KQL), is now defined based on ERLOG. The user-friendliness of KQL retrievals has been empirically tested against the standard relational language SQL[9]. The results showed that KQL users were 38% more accurate than SQL users. Learning time needed for KQL was also much shorter than that for SQL.

Semantic definition of KQL is syntax driven. Every syntactic construct has a semantic correspondence. The main features of KQL include: a high-level knowledge level user-database interaction with little requirement for data structures, is-a inheritance, automatic inheritance of attributes and relationships, full preservation of entity identity, and an English-like syntax.

The syntax is given in extended BNF. A string within quotes denotes the string itself. Others without quotes will be defined by further EBNF rules. Any construct within { } can be repeated 0 or more times. Any construct within [ ] is optional. In some cases, the subscribed notation $x_1, x_2, \ldots, x_n$ is used in place of the { and } notation. This is to enable references to particular x's. The semantics of a KQL command is defined based on ERLOG. Let the semantics of a KQL term K be denoted by $S(K)$.

The following basic terms are defined first.

- name: this is a particular string that starts with a letter, consists of letters and digits, and can be of any length, depending on the machine implementation.

- entity-type-name: this is the name of an entity type. For simplicity, we may use $e_1, e_2, \ldots$ in place of entity-type-name$_1$, entity-type-name$_2$, ... The lowercase letter e is used instead of the more common capital letter to avoid confusion with Prolog's naming of variables.
- relationship-type-name: this is the name of a relationship type. For simplicity, we may use r in place of relationship-type-name.
- type-name: this is either an entity-type-name or a relationship-type-name.
- attribute-name: this is the name of an attribute, which can belong to an entity or a relationship. For simplicity, we may use a in place of attribute-name.
- datatype-name: this is the name of a datatype. For simplicity, we may use d in place of datatype-name.
- role-name: this is a name to specify the role that an entity plays in a relationship. It is optional and used only in cases where the same entity type has multiple roles in a relationship.
- instance-identifier: this is a variable used to denote a particular instance of a particular entity or relationship type. A variable is a name. For simplicity, we may use I in place of instance-identifier.
- attribute-value: this refers to the value of an attribute of an instance. The BNF syntax is attribute-value ::= instance-identifier attribute-name. For simplicity, we may use $av_1, av_2, \ldots$ in place of attribute-value$_1$, attribute-value$_2$, ...
- value: this is a value, e.g., an integer, a string, an attribute-value, or an expression that returns a value. For simplicity, we may use $v_1, v_2, \ldots$ in place of value$_1$, value$_2$, ...

In general , a KQL command is defined as:

```
KQL ::= model-command "." | instance-command "."
```

Model Commands

A model-command is used to change the ER model.

```
model-command ::= new-entity-type | new-relationship-type | new-isa |
                  delete-type | delete-attribute | add-attribute
```

An entity type has a name and 0, 1 or more attributes. Each attribute is associated with a datatype. An entity may have a key that consists of one or more attributes. A key is not necessary as system surrogates will be generated to keep track of the instances.

```
new-entity-type::="NEW ENTITY" e "," ["ATTRIBUTE:DATATYPE"a1":"d1,
            a2":"d2,...,an":"dn] ["KEY" ak1, ak2, ..., am]
```

The meaning of this command, written as $S(new - entity - type)$, is:

```
a_e(e), a_a(e,[a1,d1]), a_a(e,[a2,d2]),..., a_a(e,[an,dn]),
   a_key(e,[ak1,ak2,...,akm]).
```

The key attributes $a_{k1}, a_{k2}, \ldots, a_{km}$ can be found in $a_1, a_2, \ldots, a_n$. When $m > 1$, the key is a composite key. Where there are no keys or no attributes, the appropriate predicates (a_key or a_a) will be omitted.

For example, the following KQL command to create a new entity type supplier

```
NEW ENTITY supplier, ATTRIBUTE:DATATYPE name:string, id:string, KEY id.
```

has the following semantics:

```
a_e(supplier), a_a(supplier, [name,string]), a_a(supplier,[id,string]),
   a_key(supplier,id).
```

A relationship has zero or more attributes. It must have at least two entity types, not necessarily distinct. If the entity types are not distinct, then their roles must be. A new relationship type is defined by the following command.

```
new-relationship-type ::= "NEW RELATIONSHIP" r
   ["ATTRIBUTE:DATATYPE" a1":"d1,a2":"d2,...,an":"dn]
   "ENTITY:ROLE:CARD" e1":"role-name1":"card1"," e2":"role-name2":"card2 ","
                    ...","  en":"role-namen":"cardn
   card ::= "[" lower-degree "," upper-degree "]"
   lower-degree ::= digit {digit}
   upper-degree ::= digit {digit} | "*"
   digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
S(new-relationship-type)= a_r(r,[[e1,role-name1,card1],[e2,role-name2,card2],
                  ...,[en,role-namen,cardn]]), a_a(r,[a1,d1]),a_a(r,[a2,d2]),
                  ...,a_a(r,[an,dn])
```

The definition of is-a relationship requires the superentity and the list of subentities. In addition, the properties (partial, total, disjoint, overlap) of the is-a relationship can be specified.

```
new-isa::= "NEW" "ISA" "SUPERENTITY:" e1,
              "SUBENTITY:" e2,e3,...,en,"PROPERTY:"isa-properties
isa-properties::= "total" | "partial" | "disjoint" | "overlap" | "total" ","
                  "disjoint" | "total" "," "overlap" | "partial" ","
                  "disjoint" | "partial" "," "overlap"
S(new-isa)=a_isa(e1,[e2,e3,...,en],[isa-properties])
```

The last three model commands are described below.

```
delete-type::= "DELETE TYPE" type-name
type-name::= entity-type-name | relationship-type-name
S(delete-type)= d_t(type-name).
delete-attribute::= "DELETE ATTRIBUTE" attribute-name "OF" type-name
S(delete-attribute)= d_a(type-name, attribute-name).
add-attribute::= "ADD ATTRIBUTE" attribute-name ":" datatype-name
                    "TO" type-name
S(add-attribute) = a_a(type-name, [attribute-name, datatype-name]).
```

Instance Commands

Instance commands are used to change or retrieve the instances of the entity and relationship types. The commands can add new instances, delete existing instances, modify the attribute values, or retrieve the instances and values. A single command is allowed to do all these.

```
instance-command::= instance-clause database-action-clause
                        [result-action-clause] [where-clause]
```

The result-action-clause performs further actions on the retrieved results, such as sorting, renaming, subtotals, or other report instructions. The result-action-clause is not further discussed since it is more of a report formatting than database operations.

Since ERLOG returns one set of answers at a time whereas a KQL instance command applies to all the possible sets of answers. Hence, the semantics of KQL must enclose the predicates of ERLOG with suitable Prolog structures to return all the possible results.

```
S(instance-command)= S(instance-clause), S(where-clause),
                     S(database-action-clause), fail.
S(instance-command).
```

The syntax and meaning of instance-clause are as follows:

```
instance-clause::=instance {, instance}
instance::=instance-identifier "IS" type-name
S(instance-clause)=S(instance1), S(instance2), ..., S(instancen)
S(instance)=q_i(type-name, instance-identifier)
```

The database-action-clause can contain any one, two, three or all of the four clauses: insert-clause, delete-clause, change-clause and select-clause. The insert-clause adds new instances, the delete-clause deletes chosen instances, the change-clause changes the attribute values of chosen instances, and the select-clause shows the attribute values (with possible computations) of chosen instances. The instances are those specified in the instance-clause which meet the conditions specified in the where-clause.

```
database-action-clause::= insert-clause delete-clause change-clause select-
                          clause|delete-clause change-clause select-clause|
                          change-clause select-clause|select-clause
S(database-action-clause)=S(insert-clause),S(delete-clause),
                          S(change-clause),S(select-clause).

insert-clause ::= "INSERT" new-instance {, new-instance}
new-instance ::= new-entity-instance | new-relationship-instance
new-entity-instance ::= e "(" I ")"
new-relationship-instance::= r"("I1,I2[role2], I3[role3],...,In[rolen]")"
```

where n is one more than the degree of the relationship r.

```
S(insert-clause)=S(new-instance1),S(new-instance2),..., S(new-instancen)
S(new-entity-instance) = a_i(entity-type-name, I),
```

where I will not have been instantiated before, but the predicate will instantiate I.

```
S(new-relationship-instance)=a_i(r,I1), a_re(I1,[[I2,role2],[I3,role3],...,
                                 [In,rolen]])
```

Again, I_1 is uninstantiated before. A new relationship instance involves existing entity instances. These instances are specified in the instance-clause and conditioned in the where-clause. The attributes of the new instances are added using the change-clause.

```
delete-clause::= "DELETE" I1 "," I2"," ..."," In
S(delete-clause)= d_i(I1), d_i(I2), ... , d_i(In).
change-clause::= "CHANGE" I1 a1 "TO" v1 "," I2 a2 "TO" v2 ","..."," In an
                 "TO" vn
S(change-clause)=V1 is v1, a_v(I1, a1, V1), V2 is v2, a_v(I2, a2, V2),...,
                 Vn is vn, a_v(In, an, Vn)
```

The Prolog *is* predicate is used here to evaluate the v's, which can be expressions, and instantiate the results of the V's.

```
select-clause::= "SELECT" select-item {"," select-item }
select-item::= I a | I * | expression
S(select-clause)= S(select-item1), S(select-item2),..., S(select-itemn)
```

There are three cases for the semantics of the select-items. The first case displays an attribute value of an instance: $S(I\ \ a) = q\_v(I, a, V)$, write(V). The second case displays all the immediate attribute values of an instance: $S(I*) = S(I\ A_1)$, $S(I\ A_2), ..., S(I\ A_n)$ where A's are from $q\_i(T, I), q\_a(T, A)$. The third case displays: S(expression) $= V$ *is* expression, write(V). This will be further described when expression is defined.

```
where-clause::= "WHERE" condition-list
condition-list::= condition-andlist ["OR" condition-list]
condition-andlist::= condition ["," condition-andlist]
condition::= NOT (condition-list) | relationship-existence -condition|
   expression-comparison-condition | membership-condition| exists-condition |
   isa-relationship-condition | entity-equality-condition | combination-
   relationship-condition-1 | combination-relationship-condition-2

S(where-clause)= S(condition-list)
S(condition-andlist OR condition-list)=S(condition-andlist)|S(condition-list)
S(condition, condition-andlist)= S(condition), S(condition-andlist)
S(NOT(condition-list))= not (S(condition-list))

relationship-existence-condition::=I1 ["(" role-name1")"] I2 [ "( "
                                        role-name2 ")"] I3
S(relationship-existence-condition)=q_re(I2,[[I1,role-name1],[I3,role-name2]])
```

As required in the *q_re* predicate, I_2 must be a relationship instance while I_1 and I_3 must be entity instances.

```
expression-comparison-condition::= expression relational-operator expression
expression::= [sign] term { additional-operator term}
term::= factor {multiplication-operator factor}
factor=i a|number|string|list|"("expression")"|statistical-expression
relational-operator::= "=" | "<>" | "<" | "<=" | ">" | ">="
addition-operator::= "+" | "-"
multiplication-operator::= "*" | "/"

S(expression1 relational-operator expression2)= V1 is expression1,
                        V2 is expression2, comp(relational-operator, V1, V2)
```

where *is* and *comp* are Prolog predicates.

```
statistical-operation::= instance-statistical-operation | attribute-
                              statistical-operation
instance-statistical-operation::= "COUNT" [ "UNIQUE" ] "(" i [ "FOR EACH"
                                    grouping-values ] ")"
attribute-statistical-operation::= statistical-operator [ "UNIQUE" ] "("
                                    I a ["FOR EACH" grouping-values])
statistical-operator::= COUNT | MAX | MIN | AVG | SUM
grouping-values::= grouping-value {, grouping-values}
```

```
grouping-value::= I | I a
```

Statistical operations add considerable complexity to the direct correspondence between KQL terms and ERLOG predicates. The most obvious effect is that a KQL command cannot be transformed into Prolog's fail loop to retrieve all answers. Now, all the sets of instances satisfying all conditions except the statistical conditions must be retrieved and stored. The statistical operations are then applied to the sets of instances. There are also cases where the statistical conditions cannot be specified with simple Prolog predicates. One such case is cyclic statistical condition, e.g., suppliers supply parts, the count of suppliers for each part > 3, and the count of parts for each supplier > 2. Here, the query wants the suppliers who supply at least 3 parts that are supplied by at least 2 of these suppliers, without counting other suppliers who are not supplying at least 3 of these parts. The answer requires a cyclical evaluation until no more changes occur to the selected sets of suppliers and parts. In general, KQL allows for any number of these cycles involving any number of entities and relationships. The full treatment of this is left for another occasion.

```
membership-condition::= expression "IN" list
list::= "[" expression {, expression} "]"
list::= "(" instance-clause "SELECT" expression [where-clause] ")"
S(expression IN [expression1,expression2,...,expressionn])=V is expression,
                V1 is expression1, V2 is expression2, ..., Vn is expressionn,
member(V, [V1, V2, ..., Vn])
```

where *member* is a Prolog predicate.

```
S(expression IN (instance-clause "SELECT" expression1 where-clause))=
   V is expression, [! S(instance-clause), S(where-clause), V1 is
   expression1, V == V1 !]
```

The construct [! ... !] is called the snip. When backtracking encounters a snip, the goals within the snip are skipped. This is to refine the results. Otherwise if the subquery has two values that equal V, then the goals will be true twice with resultant duplication of results.

```
exists-condition::= "EXISTS" "(" instance-clause [where-clause ] ")"
S(exists-condition)= [! S(instance-clause), S(where-clause) !]
isa-relationship-condition::= I "ISA" e
S(I ISA e)= q_i(T,I), q_isalinked(T,e)
entity-equality-condition::= I1 "=" I2
S(I1 = I2)= I1 == I2

combination-relationship-condition-1::=I1 ["("role-name")"] r"-RELATED" I2
S(combination-relationship-condition-1)=q_i(r,I), q_re(I,[[I1, role-name],
                                                [I2,_]])
```

This is a condition that I_1 and I_2 are related through the relationship r. The particular instance of r that relates these two entity instances is not important in the query.

```
combination-relationship-condition-2::= I1 ["(" role-name ")" ] r"-RELATED"
   number-spec e
number-spec::= ALL | NO | [relational-operator] number
```

Depending on the choice of number-spec, the semantics of the three choices are, respectively,

1. not $(q\_i(e, I)$, not$(q_i(r, I_2), q_re(I_2, [[I_1, rolename], [I, _]]))$
2. not $(q\_i(r, I), q\_re(I, [I_1, rolename]]))$
3. $setof(I, (q\_i(e, I), q\_i(r, I_2), q\_re(I_2, [[I_1, rolename], [I, \_]])), Iset)$, $count(Iset, N), comp(relational - operator, N, number)$

In English, this condition says that I_1 is related through the relationship r to all of the instances of entity type e, none of the instances of entity type e, a certain number of instances of entity type e, or greater than/less than/not equal to a certain number of instances of entity type e.

4 Conclusion

It is illustrated how the abstraction level of Prolog can be raised from its present physical/logical levels to the conceptual level. The result, ERLOG, is then suitable for being used as a foundation to directly define database languages that also belong to the conceptual level. The advantages are that conceptual level information need no longer be transformed into logical and physical level information, and the implicit integrities at the conceptual level need not be tediously and repeatedly explicated. This allows us to focus on conceptual details without digressing into lower levels. As illustration, a database language designed for the entity relationship model is defined based on ERLOG. This demonstrates a method that can be used to formally define and compare many ER languages, including graphical ER languages.

An important distinction for ERLOG, as compared to other ER calculuses[10-13], other ER languages[14,15], or other object-based logic systems[16,17], is that the three main tasks of database model specification, update and retrieval are all covered at the conceptual level. ERLOG and KQL cover definition, modification and query of the ER model, as well as insertion, modification and retrieval of entity instances and relationship instances. The inclusion of updates has meant the inclusion of automatic maintenance of ER integrities into the system. Other differentiating features include the implicit semantics of inheritance of attributes as well as relationships, the use of system generated surrogates by the system to identify entity and relationship instances, and the possibility of adding, deleting or modifying entity or relationship types even after the instances have been added. ERLOG also serves as an executable version of the ER model. Any particular ER model can be easily defined, and the data can be added and retrieved for verification of the model design.

References

[1] Batini C, Ceri S, Navathe S B. Conceptual Database Design, An Entity Relationship Approach. The Benjamin/Cummings Publishing Co. Inc., USA, 1992.

[2] Vossen G. Data Models, Database Languages and DBMSs. Addison-Wesley, UK, 1991.

[3] Elmasri R, Navathe S B. Fundamentals of Database Systems. Addison Wesley, 1989.

[4] Olive Antoni. Analysis of conceptual and logical models in information systems design methodologies. In *Information Systems Design Methodologies*, Olle T W, Sol H G, Tully C J (eds.), Elsevier Science Publishers, IFIP 1983.

[5] Gray P M D, Kulkarni K G, Paton N W. Object-Oriented Databases, A Semantic Data Model Approach. Prentice-Hall, USA, 1992.

[6] Chen P P. A preliminary framework for entity-relationship models. In *ER Approach to Information modeling and analysis*, Chen P P (ed.), North-Holland, 1981.

[7] Hainaut J L. Entity relationship models: formal specification and comparison. In *Entity Relationship Approach, The Core of Conceptual Modelling*, Kangassalo H (ed.), Elsevier Science Pub. (North-Holland), pp. 433-444, 1991.

[8] Chan H C. An entity-relationship enhanced logic system. In *The 2nd Int'l Symp. Database Systems for Advanced Applications*, Tokyo, April 1991, pp. 401-410.

[9] Chan H C, Wei K K, Siau K L. Conceptual level versus logical level user-database interaction. In *Proc. the 12th Int'l Conf. on Information Systems*, DeGross J I, Benbasat I, DeSantis G, Beath C M (eds.), New York, USA, Dec 16-18, 1991, pp. 29-40.

[10] Atzeni P, Chen P P. Completeness of query languages for the entity relationship model. In *Entity-Relationship Approach to Information Modeling and Analysis*, Chen P P(ed.), North-Holland, 1981. pp. 109-122.

[11] Chen P P. An algebra for a directional binary entity relationship model. In *The First Int'l Conf. on Data Engineering*, 1984, pp. 37-41.

[12] Parent C, Spaccapietra S. An entity-relationship algebra. In *The First Int'l Conf. on Data Engineering*, 1984, pp. 500-509.

[13] Parent C, Rolins H, Yetongnon K, Spaccapietra S. An ER calculus for the entity-relationship complex model. In *Proc. of the 8th Int'l Conf. on Entity-Relationship Approach*, Lochovsky F H (ed.), 1989, pp. 248-262.

[14] Hohenstein U. Automatic transformation of an entity-relationship query language into SQL. In *Proc. of the 18th Int'l Conf. on Entity-Relationship Approach*, 1989, pp. 309-327.

[15] Subieta K, Missala M. Semantics of query languages for the entity relationship model. In *Entity-Relationship Approach*, Spaccapietra S (ed.), Elsevier Sciences Publishers, 1987, pp. 199-216.

[16] Abiteboul S, Grumbach S. COL: A logic-based language for complex objects. In *Advances in Database Programming Languages*, Bancilhon F, Buneman P (eds.), ACM Press, New York, 1990, pp. 347-374.

[17] Bancilhon F, Khoshafian S. A calculus for complex objects. In *Proc. of ACM SIGACTS/SIGMOD Symp. on Principles of Database Systems*, 1985.

H. C. Chan is with the Department of Information Systems and Computer Science at the National University of Singapore. He obtained his B.A. and M.A. degrees from the University of Cambridge, UK, and his Ph.D. degree from the University of British Columbia, Canada. His research interests include database models, query languages, human-computer interaction and information systems. He has taught courses in programming, data structures, database and information systems.