This paper describes a system (ERLOG) that allows the entity relationship (ER) model to be represented and manipulated declaratively in the Prolog programming language. The predicate calculus inherent in Prolog is enhanced with a set of ER predicates that allows for the definition as well as the manipulation of ER types and instances. The result is a seamless integration of entity-relationship knowledge and Prolog. The enhanced logic also provides a formal definition for an ER calculus. There is also a short discussion on how the system compare with various ER calculi and algebra that have been proposed. A simple implementation is given for ERLOG.

Secondly, it aims to provide a basic set of operations that are described and understood purely in terms of entities and relationships. This set can then be used to define the semantics of ER query languages, covering updates and retrievals of the ER model as well as the ER instances. This will eliminate the need to define the semantics of ER query languages based on implementation issues [10]. Thirdly, it is hoped that the ER enhanced Prolog system can be more easily implemented and may eventually be used for real database applications. The system will have the added benefits of coupling logic with database, benefits such as easy recursive queries.

Part 2 shows the predicates needed to define and manipulate the ER model. Part 3 is a short discussion on ERLOG, with some comparison to other approaches. Part 4 gives a simple implementation of these predicates.

2. The ER Predicate Calculus

In order to enhance Prolog with ER knowledge, we need to introduce some ER predicates for declaring and manipulating knowledge of entity and relationship types and instances. These will be the only predicates that a user will need to deal with. The user is completely free from worries of how the ER knowledge is to be stored physically. Moreover, the system should maintain all the integrities inherent in the ER model. These include the constraint that a relationship instance can exist only when the involved entity instances also exist. Another constraint is that a subset entity instance can exist only if it also exist at the superset type. These predicates are fully integrated into the Prolog language so that they can be used just like any other Prolog predicates. This provides a seamless integration, unlike, say, the embedding of SQL in C which requires a lot of extra effort from the user.
An essential criterion is that the additional predicates must be comprehensive enough to cover all the basic functions needed to define, update and retrieve from the ER database. The required predicates can be classified into the following three categories:

1. Model Predicates. These are used to define, modify and query the ER model.
2. Update Predicates. These are used to assert new ER instances, to change the attribute values of existing instances and to delete existing instances.
3. Retrieval Predicates. These are used to retrieve the ER instances.

The notation used to describe the predicates is one that is commonly used for describing Prolog predicates. Briefly, words starting with a small letter cannot be replaced by other words, words starting with a capital letter denote variables and can be replaced by other words. A "+", "-" or "?" may appear in front of an argument. A "+" means that the argument must be instantiated (i.e. given a value) when the predicate is called; a "-" means that the argument will be returned a value and therefore should not be instantiated; a "?" means that the argument can be used in both ways. More details of the syntax of Prolog can be found in, for example, [6].

Throughout this paper, we will use the ER model in Figure 1 for illustrations.

2.1 Model Predicates

There are many versions of the ER model [4]. We will use a version very close to that proposed in [3] but with some additions. This version of the ER model consists of entities, and relationships can exist among entities. Both entities and relationships can have attributes, and a relationship can involve any number of entities; If an entity is involved multiple times in a relationship, i.e. it plays many roles in the relationship, then role names must be assigned. In other cases, the role names are optional. In addition, this version includes the concept of is-a relationships among entities.

![Figure 1. An Entity Relationship Model](image)
relationships involving multiple entity types can be defined. The system allows the role names to be optional unless the same entity appears more than once in the relationship, in which case the role names are needed for distinguishing purpose.

Special is-a relationships can be specified with the "a_isa" predicate. This predicate defines a superclass entity type and a list of subclass entity types. The treatment of is-a relationships is described more fully in a separate section.

In addition, the predicates allow for dynamic modification of the ER model. New entity and relationship types and new attributes for old types can be added as and when required. Also, existing attributes and types can be deleted when necessary. The system takes care of the impact on the instances of the affected types.

There are some points about this ER model that depart from the usual. First, there is no facility at this stage for the definition of a datatype to be associated with an attribute. It is therefore up to the user to enforce any such restrictions. There are however advantages, as the attributes are not restricted to some predefined datatypes such as integers and characters. In particular, an attribute can contain complex values or references to other instances.

Second, there is no definition of key attributes. The system uses surrogate keys to keep track of the individual instances. It allows the creation of two instances with the same set of attributes, but with different surrogate keys. This allows for the situation such as where a user wants to keep track of two chairs which are identical in attributes. The chairs may, at times, be occupied. When occupied, the chairs can be differentiated. When both are unoccupied, the user treats both chairs as identical. Correspondingly, the system treats both chairs as identical, except for its own system surrogates.

In addition to defining and manipulating the ER model, the user will need to know what is the ER model already defined. This need is met by a set of predicates to query the ER model.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(e,?T)</td>
<td>ask for an entity type</td>
</tr>
<tr>
<td>t(r,?T)</td>
<td>ask for a relationship type</td>
</tr>
<tr>
<td>a(?T,?A)</td>
<td>ask for type and attribute of that type</td>
</tr>
<tr>
<td>ro(?RT,?ET,?Role)</td>
<td>ask about relationship participation</td>
</tr>
<tr>
<td>is_a(?IS,?E,?E2)</td>
<td>ask if E2 is subset of E in the is-a IS</td>
</tr>
<tr>
<td>isalinked(?E1,?E2)</td>
<td>ask if E1 and E2 are is-a linked.</td>
</tr>
</tbody>
</table>

Notice that the "t" predicate accepts "?T" as the second argument. It can be used to find an existing type name, or to check the existence of a certain type. For example, entering "t(e, X)" may get X instantiated to "staff", whereas entering "t(e, staff)" will get a true answer.

More predicates may be added to allow trivial change of names. For example, the user may want to change a role name without changing anything else. If no predicate caters for this, then the user will have to delete the previous relationship and define a new relationship. Instances of the relationship will have to be inserted again. This is obviously a very tedious process just to change a role name.

2.2 Update Predicates

After having defined an ER model, the user would want to assert and manipulate instances for the various entity and relationship types. This capability is provided by the following predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_i(+ T, I)</td>
<td>assert instance of T</td>
</tr>
<tr>
<td>a_v(+I,+A,+V)</td>
<td>assert value V for attribute A of instance I</td>
</tr>
<tr>
<td>a_re(+RI,+EI,+Role)</td>
<td>assert participation of instances</td>
</tr>
<tr>
<td>d_i(+I)</td>
<td>delete instance</td>
</tr>
<tr>
<td>d_v(+I,+A)</td>
<td>delete an attribute value of an instance</td>
</tr>
</tbody>
</table>
Notice that the "a_i" predicate's second argument should not be instantiated. The actual value is given by the system. It is just a way to refer to an instance without having to mention any of its attribute values. We can think of this as a system surrogate. The system provides default attribute values to this instance. The attribute values can be changed by using the "a_v" predicate.

In addition, the user can delete instances, or the attribute values of instances with the "d_i" and "d_v" predicates. These predicates require the 'I' argument to be instantiated. Hence, they have to be used in conjunction with the predicates for retrieval since an instance must be retrieved (identified) before it can be deleted. The retrieval predicate is described later.

The updates may have propagation effects. For example, deleting an entity instance will result in the automatic deletion of relationship instances that involve this entity instance. The propagation effect is carried out automatically in order to preserve the integrity of the database. Full details are given in Section 2.4 and 2.5.

2.3 Retrieval Predicates

The functions to retrieve instances of entities or relationships are provided by the following three predicates:

\[
g_i(+T,?I) - \text{get instance I of type } T
\]

\[
v(+I,?A,?V) - \text{confirm that attribute A of I has value V}
\]

\[
re(+RI,?EI,?Role) - \text{confirm that RI and EI are related by Role}
\]

These three predicates constitute the only basic conditions that can be said about the ER instances. This is because they cover all the basic facts in the ER model. These conditions are that an instance belongs to a certain type, that an instance’s attribute is some value, that two instances are related. All other conditions can be formulated in terms of these.

The "g_i" predicate retrieves an instance of the specified type. Retrieval of subsequent instances will have to be done through backtracking. The "v" and "re" predicates can be used to specify conditions on the instances or to retrieve values about the instances, depending on whether the relevant variables have been instantiated. For example, to retrieve the salary of a staff member with the name "John", the following predicates will be used:

\[
g_i(staff, X),
\]

\[
v(X, name, "John"),
\]

\[
v(X, salary, S),
\]

\[
write(S).
\]

Another example, to list the name of all professors, is:

\[
getallnames :-
g_i(professor, X),
\]

\[
v(X, name, N),
\]

\[
write(X), nl,
\]

\[
fail.
\]

getallnames.

These retrieval predicates can be combined to form more complex queries. One example is the following query to find the staff members who have higher salaries than their department heads.

\[
query :-
g_i(staff,S), g_i(work,W), g_i(department,D),
\]

\[
re(W,S,-), re(W,D,-),
g_i(professor,P), g_i(head,H),
\]

\[
re(H,P,-), re(H,D,-),
v(S,salary,Sall), v(P,sal,Sal2),
\]

\[
Sall > Sal2.
\]

These predicates can also be combined with other Prolog predicates. One example is the following query to find the department with more than twenty staff members.

\[
query :-
g_i(department,D),
\]

\[
s Moff(W, ((g_i(work,W), re(W,D,-) ), Wset),
\]

\[
countmember(Wset,N),
\]

\[
N>20.
\]

(In this query, the setof predicate is a Prolog predicate that returns a set Wset (the third argument) of all the W's that satisfy the goal in the second argument. The countmember predicate will return the number N of elements in the list Wset.)
2.4 Handling of Is-a Relationships

The system ERLOG automatically handles inheritances and special constraints for entities involved in is-a relationships. An inclusion relationship associates subset and superset entity types. For example, in figure 1, PROFESSOR and DOCTOR are subsets of the superset STAFF. We may speak of PROFESSOR and DOCTOR as specializations of STAFF. If there is no overlap among the subset entity types and if every instance of STAFF is also an instance of one of them, we can refer to STAFF as the generalization of the other two. In any case, the three entity types are said to be is-a linked.

The important characteristic of the is-a relationship is property inheritance [2]. All properties of the superset type are inherited by the subset types. The term "property" is intended to include participation in relationships as well as attributes. Thus, if STAFF has attributes NAME, SALARY and NUMBER, these will be inherited by PROFESSOR and DOCTOR. Also, if STAFF is related to VEHICLE, the subsets will inherit this relationship. This is the usual downward inheritance for classes.

An is-a relationship is defined using the a_isa predicate described in the model section.

When we say "E is-a F", we mean that E is a direct subclass of F. When we say that "E is-a linked to F", we mean that there are direct or indirect is-a linkages between E and F. Thus, any two entities in an is-a network are is-a linked. For example, if PROFESSOR has two subclasses FULL-PROFESSOR and ASSOCIATE-PROFESSOR, then STAFF, PROFESSOR, DOCTOR, FULL-PROFESSOR and ASSOCIATE-PROFESSOR are all is-a linked to one another.

For classes, we have the usual downward inheritance. For instances, a more powerful multi-directional is-a reference is possible. If an instance of a superset is also an instance of a subset, then that superset instance will also have all the properties of the subset instance. This is downward is-a reference. If an instance of a subset, say, PROFESSOR is also an instance of another subset, say, DOCTOR, then that instance of PROFESSOR has all the properties of DOCTOR. This is sideway is-a reference. Hence, for queries involving is-a relationships, we propose a principle of multi-directional is-a reference:

When referring to an entity E, it should be possible to refer directly to attributes and relationships of entities that are is-a linked to E as if these properties were defined for E itself. Furthermore, an entity instance that belongs to multiple types can be referenced by any of its type names.

An object has properties which may be stored in a database in a variety of ways. Property inheritance is a data organization strategy; it does not have any underlying significance in the physical world. In particular, all entity instances which are isa-linked are actually different manifestations of a single real-world object. The user should not have to explicitly specify (or even necessarily be aware of) these different manifestations in order to formulate a query.

Some examples of instance retrievals that use multi-directional is-a reference are the following:

1. g_i(doctor,E),v(E,name,N) - upward is-a reference of attribute.
2. g_i(staff,E),v(E,rank,R) - downward is-a reference of attribute.
3. g_i(doctor,E),v(E,rank,"full") - sideway is-a reference of attribute. This is a combination of upward and downward is-a references.
4. g_i(doctor,E),g_i(work,W),re(W,E,-) - upward is-a reference of relationship.
5. g_i(staff,E),g_i(head,H),re(H,E,-) - downward is-a reference of relationship.
6. g_i(doctor,E),g_i(head,H),re(H,E,-) - sideway is-a reference of relationship.

Some examples of instance modification that use multi-directional is-a reference are the following:

1. a_i(doctor,E),a_v(E,name,"John") - upward is-a reference. This creates an instance of DOCTOR as well as a STAFF instance with the same surrogate key.
2. a_i(staff,E),a_v(E,rank,"associate") - downward is-a
reference. This will create a STAFF instance and a PROFESSOR instance with the same surrogate key.

3. a_(doctor,E),a_(rank,"full") - sideway is-a reference. This will create a DOCTOR instance, a STAFF instance and a PROFESSOR instance, all with the same surrogate key.

4. g_(doctor,E),v(E,name,"Jack"),a_(rank,"associate") - sideway is-a reference. Unlike the previous examples that create new instances, this is a modification of an existing doctor who now has a professor rank of "associate". If E is not already a professor, then a professor instance with the same surrogate key will be created.

2.5 Integrity Constraints

The ER model comes with some in-built integrity constraints. These constraints are automatically maintained by the system. The specific constraints and how they are maintained are described below.

1. A relationship instance cannot exist if the involved entity instances do not exist. In action, this means that when an entity instance is stated as being involved in a relationship instance, that entity instance must already exists. This is done by the a_re predicate. In addition, when an entity instance is deleted, the system will delete all relationship instances that involve this entity instance. The relationship instances may come from many different relationship types, depending on the actual ER model.

2. A relationship type cannot exist if the involved entity types do not exist. In action, when an entity type is stated as being involved in a relationship type, the entity type must be already defined. This is done by the a_r predicate. Also, when an entity type is deleted, the relationship types that involve this entity type will also be deleted. This is done by the d_t predicate.

3. All instances can exist only if the type exist. Thus when an instance is defined, it must be stated to be of a type that has been defined. This is done by the a_i predicate. And when a type is deleted, all the instances of that type will be deleted. This is done by the d_t predicate.

4. All subset instances must exist at the superset level. When an instance of a subset is defined, the system checks that it exist at the superset level, otherwise the system automatically create the same instance at the superset level. This will continue all the way to the root of the is-a hierarchy. This is done by the a_i predicate. When a superset instance is deleted, the system deletes all its subset existence. This is done by the d_i predicate.

External constraints are also being considered. These will allow the user to define constraints to better reflect the constraints of the user's world. An example may be that a staff member's salary cannot be less that a certain minimum.

3. Some Characteristics of ERLOG

The set of predicates have been designed to be consistent and as small as possible. For example, the longer predicate to specify an attribute value, e.g. value(X, color,"blue"), is chosen over a shorter version such as color(X, blue) for the following two reasons. One is consistency, the value predicate has the same arguments as the predicate to assert a new value. The other is to reduce the number of predicates: with the second choice, each attribute will result in a predicate. The proliferation of attributes will also lead to a need to enforce globally unique names for attributes. With the longer predicates, attributes need be unique only within a type.

Another important design consideration is that these predicates should protect users from minor changes in the database. For example, addition of new attributes to an entity should not change any of these predicates. This will minimize program maintenance due to database changes.

The system is simple. For retrievals of instances, there are only three predicates. These corresponds to the three basic facts we can say about instances - that an instance
is of a certain type, that an instance has a certain attribute value, and that two instances are related. For the whole system covering the creation, modification and retrieval of the ER model and the ER instances, there are only 21 predicates, and these are atomic predicates that each concern a single fact. For example, asserting a type and asserting an attribute of a type are two separate predicates.

The system does not impose unnecessary restriction, as compared to the system in [1] where attribute values of instances from different types cannot be compared, and where only instances of one type can be retrieved.

The ER model supported by the present system is quite rich in semantics. For example, compared to [5]'s algebra for a primitive binary ER model, this system allows attributes for relationships, and it allows is-a relationships. It also allows role names which makes it possible to define recursive relationships, i.e. relationships where an entity type has multiple roles. The number of advance concepts covered in this system is not as many as in some other systems, e.g. the system does not allow relationship among relationships. However, ERLOG treats the covered concepts more thoroughly. For example, it treats is-a relationships in a nicer and more comprehensive way than other systems - it allows multi-directional is-a reference to be specified implicitly. It also provides operations for updates and deletions to both types and instances. Moreover, the integrity is maintained for these changes.

Some ER calculus and ER algebra [8,9] allow for complex and multi-valued attributes. It is possible in ERLOG to have these types of attributes. The user can assert multi-values for an attribute, e.g. the following asserts two values for the name of a staff member:

\[
g_i(staff,E),v(E,\text{number},"E123"),
\]

\[
a_v(E,\text{name},["John Smith", "Jon Smith"]).
\]

No special operations are predefined for dealing with multi-valued attributes. However, because of its setting in Prolog, additional operations can be easily defined by the user. For example, if salary is multi-valued and the user wants a staff member with at least one salary getting than 3000, this can be done in the following query:

\[
g_i(staff,E),v(E,\text{salary},S),\text{member}(S_1,S),
S_1@>3000.
\]

It is also possible to assert more complex attributes such as a list of lists, a tree structure, an instance, a list or even a tree of instances. One example is the following where we assume that the type staff has an attribute called "spouse":

\[
g_i(staff,E),v(E,\text{number},"E126"),
g_i(staff,F),v(F,\text{number},"E235"),
a_v(F,\text{spouse},E).
\]

This example asserts the staff instance $E$ as the spouse of staff instance $F$.

This should be added that although the system allows instances to be asserted as attribute values, it does not actively support this. If an instance is deleted, it's surrogate number may still be stored in the attribute of some other instances. For example, continuing with the above example on spouse and staff, if staff $E$ is deleted, the attribute of $F$ will still contain the surrogate number of $E$. A good point, however, is that changes in attribute values are automatically "updated". Because only the surrogate number is stored, the attribute values must be retrieved as needed, this results in the retrieval of the up-to-date values. It also seems quite unnatural to have an instance as the attribute value of another instance. It is more natural to create a relationship and link the two instances through the relationship.

It is common to restrict the calculus, e.g. relational calculus and the ER calculus in [9] to ensure a safe expression that cannot create an infinite number of instances/tuples. The usual example is an expression to create a relation of tuples that are not found in another relation. Given a large domain, this will create a very large relation. In ERLOG this is not possible, partly because of it's setting in Prolog, and partly because it has a closed world assumption. All the known instances are already asserted in the system. It is just not possible to automatically assert unknown instances. But, on the other hand, because it is integrated with a general programming language, it is always possible to write a program to continuously assert new instances with randomly generated attribute values.
How the results from a query are organized depends heavily on the user. If the user preserves the instances and values, then the intermediate results from the use of the ER predicates can be used for further queries. One way to do this is to assert the first query as a Prolog predicate, and then used it in another query, e.g.

```
assert( (highpaystaff(E) :-
    g_i(staff,E),v(E,salary,S),
    S @> 3000.)) )
highpayprofessor(M_name) :-
    highpaystaff(M),g_i(professor,M),
    v(M,name,M_name).
```

In some ER calculus, the user can define a new entity type with instances taken from an existing type, and these instances will continue to have their previous relationship linkages. This is possible in ERLOG, in a more organized manner. The new type must be defined in an is-a hierarchy. It does not make sense to have an arbitrary type that contains the same instances as another type. If one type contains a subset of the instances of another type, then it must be is-a linked to the later type.

In summary, the proposed system has the following characteristics:
- the system is relatively simple as it is an extension of a well known system (Prolog). The ER operations are very few and simple.
- it covers all aspects of model definition and manipulation, and it does not cover database system management functions such as security and recovery.
- although it is possible to have complex entities that have attributes that are list of entities, this is not encouraged and not actively supported.
- the basic ER integrities are maintained by the system.
- the system automatically performs multi-directional is-a reference.
- the system is executable.
- new operations, such as recursions, can be easily added without distorting the original set of ER predicates.

4. Implementation

As stated previously, one major use of the system proposed above is as a clear description of the definition and manipulation of ER elements, i.e. as an ER calculus. It happens that this system can be easily implemented using Prolog. A simple implementation is described here.

A two layer architecture is used to implement the new system. The top layer consists of the ER predicates and the usual Prolog predicates. This is the layer that the user has to deal with. The second layer consists of the physical data storage for the ER database - the Prolog predicates to represent the entity and relationship types and instances. This layer is what the system has to deal with. It is transparent to the user. Some transformation algorithms are also needed to connect the two layers. These algorithms are also transparent to the user.

We now describe the second layer - the physical implementation. It must be emphasized that this is just one possible implementation. Since the implementation belongs to the physical level, it may be modified or totally changed without effecting the ER predicates described in the top layer. This ensures that programs written using the ER predicates are cushioned from changes in the ER-Prolog implementation.

The ER addition to Prolog has been implemented by a very simple method. Knowledge of the entity and relationship types in the ER model is stored with the following predicates:

```
e(entity_name, attribute-list)
```

```
r(relationship_name, entity-role-list, attribute-list)
```

```
isa(isa-type, super-type, sub-type-list)
```

For example, the ER model of Figure 1 will be partially recorded by these predicates:

```
e(staff,[number, name, salary])
e(department,[number, name])
r(work,[[staff,employed_in],[department,employs]],[date])
isas(staff, [doctor,professor]).
```

Knowledge of the entity and relationship instances are recorded by the following predicates, where the attribute values are in the same order as the corresponding predicates above, and each predicate represents one instance.
All the other ER predicates essentially access these facts and change them as and when required.

Once we have decided on the physical storage of the ER model and the ER instances, we need to provide the linkage between the ER predicates used by the users and the actual storage used by the system. The following shows a sampling of what the system will do when it encounters the ER predicates. The full set of predicates could not be listed, due to space consideration. One example is the a_t predicate. This is defined as:

\[
\text{a}_t(e,T) \leftarrow \text{not}((T,J), \text{assert}(e(T,[]))).
\]

In this case, the system checks that the type name does not already exist. Then the system creates a new predicate to record the new type.

Another example is the d_t predicate to delete an entity type. This turns out to be quite complex because of the need to maintain the database integrity. The many follow-up actions include: delete the instances of the deleted type, delete the relationship types that involve this entity type, delete the instances of these relationship types, delete the entity types that are subtypes of this entity type, and these entity type deletions in turn will lead to other follow-up actions. This "d_t" predicate calls a few other predicates (transparent to the user) to perform these actions.

\[
\text{d}_t(T) \leftarrow
c(T,X),
\text{findall}([R,(r(R,ER,A),\text{member}([T,],ER)),Rall),}
\text{d}_t\text{-list}(Rall),
\text{ifthen}((\text{isa}(T,Eall),d\_t\text{-list}(Eall)),
\text{ifthen}((\text{isa}(ISA,Su,EL),\text{member}(T,EL)),
\text{(remove}(T,EL,EL2),
\text{retract}(\text{isa}(ISA,Su,EL)),
\text{ifthen}(EL2 \seteq [\:], \text{assert}(\text{isa}(ISA,Su,FL ?))),
\text{isastree(Tree),}
\text{member}(T,Tree),
\text{remove}(T,Tree,Tree2),
\text{retract}(\text{isastree}(Tree))).
\]

Another example is the a_i predicate, which is defined as:

\[
\text{a}_i(T, I) \leftarrow
\text{nonvar}(T), \text{var}(I),
\text{r}(T,ER,AL),
\text{newname}(S),
\text{formlist}(AL,D), \text{formlist}(ER,D2),
I = [T,S],
\text{assert}(i(T,S, D2 , D)).
\]

This example is another illustration of the amount of work that is required of the system in creating an instance. The first predicate defines a relationship instance. The newname predicate returns a unique system surrogate. The formlist predicate gives a set of default values. Lastly, the a_i_isaup predicate asserts the same instance at the superset levels if it does not already exist.

In this implementation, the instance variable consists of the type and the surrogate value. With this information, the attribute values can be retrieved or updated when necessary. This has the advantage that updates are effective immediately so that subsequent access to the attributes will get the updated values. For example, consider the following program:

\[
g_i(staff,X),
v(X,salary,S),
S2 = S \times 1.1,
a_v(X,salary,S2),
v(X,salary,S3).
\]

Staff X's salary is increased by 10%. The subsequent
access will instantiate S3 with the updated salary.

The database can be saved to the secondary memory by using the "save" predicate provided with the Prolog software. This software also uses memory page swapping. Thus the size of the database is limited only by the disk space. The present implementation is slow, but it is fully functional. We are currently investigating various optimizations to improve the speed of the system. With better speed, the system can be used for real applications.

5. Conclusion

We have described a new logic system (ERLOG) that incorporates an existing logic system (Prolog) and the entity relationship model. It covers all aspects of the definition and manipulation of the ER model and ER instances. Dynamic updates to the ER model are possible. The system takes care of the basic ER integrities. In addition, it supports multi-directional is-a references for the ER instances for both updates and retrievals. This system can serve as a formal notation to describe the definition, modification and retrieval of ER models and their instances.

A two layer architecture is proposed for implementing the system, with the top layer available to the user. The lower layer is invisible to the user and can be changed without effect on the top layer. This system in fact includes a simple entity relationship database management system that frees the user from the physical details of storing the data. The system provides a seamless interface between the database predicates and the normal programming predicates.

References


