A Query Language for EER Schemas

Michael Lawley Rodney Topor

CRC for Distributed Systems Technology[*]
and
School of Computing and Information Technology
Griffith University
Nathan, Queensland 4111, Australia
`{lawley,rwt}@cit.gu.edu.au`

Abstract

We present a proposed query language for extended entity relationship schemas. The language improves on previous proposals by using only concepts explicitly in a given schema. It includes quantifiers and aggregates to allow complex queries to be expressed, and it allows derived subtypes, attributes and relationships to be defined and used in queries. Further extensions are discussed.

1 Introduction

The usual database design process is to design conceptual schema (e.g., an entity-relationship schema) which is then mapped into an implementation schema (e.g., a relational schema) for representation in a particular database management system. Application software, in particular queries, are then expressed in terms of the implementation schema (e.g., using SQL). Why is this so?

Why can't the database designer avoid mapping the conceptual schema into the implementation schema, and write application software for the conceptual schema directly? In particular, why can't casual users express queries in terms of the conceptual schema? This would avoid the mapping task and, more importantly, would allow users to express queries in more abstract, familiar, application-oriented terms.

Presumably the reason is that there is no standard data manipulation language for conceptual schemas.

[*]The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

Many other researchers have proposed query languages for the extended entity-relationship (EER) model, but none of these have become widely accepted. One reason is perhaps that the proposed languages implicitly assumed a relational or network representation of the data which showed through in the language, and thus offered little advantage over the standard relational query language SQL. Another is that these languages are usually introduced via examples and no formal semantics is given for the language. Recently however, there have been substantial efforts in defining such a language and providing a formal semantics, for example [6, 9]. The drawback of this work however is its size and complexity.

Our intended contribution in this paper is to present a new query language, ERQL, for the EER model. Our language is characterised by the facts that it only uses abstract concepts that appear in the EER schema, that is does not depend on any implicit relational representation of the EER schema, that it has a simple declarative semantics, and that it is easily extendible. To this end, we have paid particular attention to Date's critique of SQL [4].

The implementation of such a language would form just one step in our programme of hiding the implementation model and schema from users completely. Other steps would involve extending the language to include updates, incorporating it into an appropriate host language for applications, specifying dependencies and access paths in the conceptual model, and developing theory and tools for performing database design at the conceptual level.

In Section 2 we summarise the EER model that is assumed in the language definition. In Section 3 we informally describe ERQL by a series of examples that gradually introduce new features to deal with increasingly complex queries. In Section 4 we give a more formal description of the syntax and semantics of the language. Section 5 summarises our achievements and suggests some further work.

2 The EER model

We assume a fairly standard extended entity-relationship model, as described for example in the texts by Batini, Ceri and Navathe [1] and Elmasri and Navathe [5]. An example of a schema in this model, taken from [5], is shown in Figure 1.

In this model, a schema is constructed primarily from entity types (e.g., employee), attributes (e.g., name), relationship types (e.g., works_on). Entity types may be either strong (e.g., employee) or weak (e.g., dependent). Entity types may have subtypes (e.g., manager is a subtype of employee) and generalisation hierarchies (e.g., an employee may be a clerk or a buyer or a salesperson or a manager, not shown in Figure 1). Generalisation hierarchies may be either overlapping or exhaustive, and either total or partial. Attributes may be composite (e.g., name) or multi-valued (e.g., locations). Relationship types may have two or more participating entity types. If two or more of the participating entity types are equal, role names are associated

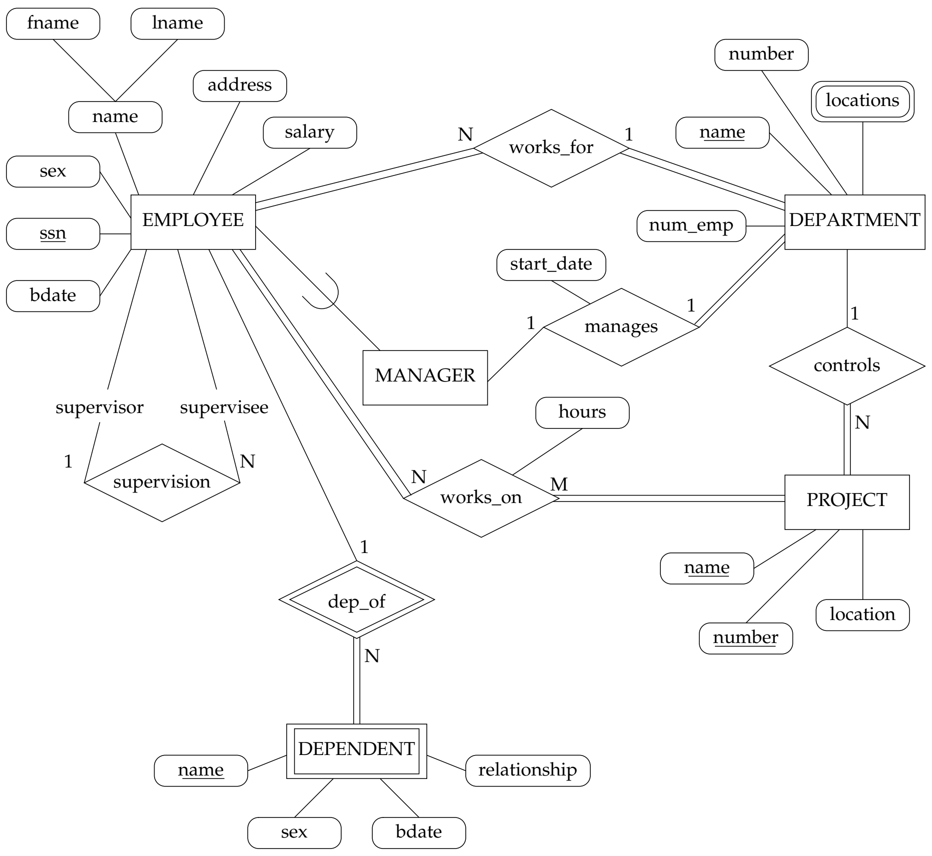

Figure 1: Example EER schema

with the links from the entity type to the relationship type to distinguish the two links (e.g., supervisor and supervisee).

Each entity type must have a primary key (e.g., ssn of employee), and each participation of an entity type in a relationship type must have a cardinality ratio (one-to-one, one-to-many, or many-to-many) and a participation constraint (total or partial).

We assume that derived (entity) subtypes, attributes, and relationship types can all be defined in terms of concepts in the schema. For example, the number of employees who work for a department is a derived attribute of department, the group of employees who supervise some employee is a derived subtype (supervisor) of employee, and the relationship between dependents and departments determined by the existence of an "intermediate" employee is a derived relationship between dependent and department.

Finally, a *database* for an EER schema S is a graph. The nodes in the graph are entities, relationships, and attribute values. For each entity e of type E, and for each attribute A of E, there is an edge from e to some value of A. If A is a multi-valued attribute there may be several edges from e to values of A. For each value of a composite attribute, there is an edge to a value of each of its component attributes (and possibly several edges for a multi-valued composite attribute). For each relationship r of type R, there is an edge from r to an entity e of type E (or subtype of E) for each entity type E participating in R, and there is an edge from r to a value of each attribute of R (and possibly several edges for a multi-valued attribute).

We use the EER schema of Figure 1 and the extensions described above in the examples used to illustrate the language in the following sections.

3 Informal Language Description

3.1 Simple Path Expressions

We use path expressions for navigation around EER schemas. Similar path expressions have previously been used for navigating around object-oriented database schemas [2, 10] and federated schemas [11].

Each path expression evaluates to a bag of tuples. The values comprising these tuples may be a simple value such as the name of a department, a complex value such as an employee's name, or an object such as an entity or relationship.

The reader is invited to compare the following example ERQL queries with the equivalent SQL queries on an appropriate relational database. We feel the simplicity, regularity, and power of ERQL will then be evident.

A path expression may refer to just an entity or a relationship. To find all the departments we would give the query:

`department` (1)

We can follow links in the EER schema. To find the starting dates of all the manages relationships we would give the query:

`manages.start_date` (2)

If an attribute is multi-valued, the result is flattened. To find all the locations of all the departments we would give the query:

`department.locations` (3)

Note, this query returns a bag of locations rather than a bag of sets.

When a relation links an entity with itself, roles must be specified for the links and these are used in the query; otherwise we would not know which link was meant to be traversed. To find all the employees who have a supervisor we would give the query:

`employee.supervisor.supervisee` (4)

To find the cartesian product of the department names and the project names we would give the query consisting of two sub-queries:

`department.name, project.name` (5)

A query on a subtype can directly reference attributes and relationships from any of its supertypes. To find the names of all managers we would give the query:

`manager.name` (6)

A term enclosed in square brackets can be used to bind a variable or to restrict a path. Such terms are called constraints. To find all the male employees and the constant `'Male'` we would give the query:

```
employee[E], E.sex['Male']                                          (7)
```

To find the social security number, address, and salary of all employees we would give the query:

```
employee[E].ssn, E.address, E.salary                                (8)
```

A variable appearing by itself in a path expression must be bound (i.e. occur in a constraint) somewhere else in the query for the query to be range restricted. Thus, query 8 is equivalent to the query:

```
employee[E].ssn, employee[E].address, employee[E].salary            (9)
```

3.2 Qualified Path Expressions

Path expressions can be qualified by use of a `WHERE` clause. To find the social security numbers of all employees with first name `'Xavier'` we would give the query:

```
employee[E].ssn WHERE E.name.fname['Xavier']                       (10)
```

Similarly, to find the SSNs and names of employees who work on some project we would give the query:

```
E.ssn, E.name WHERE employee[E].works_on                           (11)
```

If we wish to restrict query 11 to employees who work exactly 20 hours on some project, we would give the query:

```
E.ssn, E.name WHERE employee[E].works_on.hours[20]                 (12)
```

Here, a query appearing as a condition evaluates to true if it is a non-empty bag, and to false otherwise.

3.3 Boolean Operations

Queries appearing in a `WHERE` clause and treated as booleans can be combined with the usual boolean operators, `AND`, `OR`, and `NOT`. To find the SSNs of employees who have a dependent with the same birth date we would give the query:

```
employee[E].ssn WHERE
      E.bdate[B] AND E.dependents_of.dependent.birthdate[B]        (13)
```

To find the ssn of employee `'John Smith'` we would give the query:

```
employee[E].ssn WHERE E.name[N].fname['John'] AND N.lname['Smith'] (14)
```

To find the names of employees who have the same first name as one of their dependents we would give the query:

```
employee[E].name[N]
      WHERE E.dependents_of.dependent.name[DN] AND N.fname[DN]     (15)
```

Consider an EER schema containing a ternary relationship, supply, between entities supplier, project, and part.

To find all the part numbers of parts supplied to project `'Bicycle'` by supplier `'Repco'` we would give the query:

```
S.part.partno WHERE project[PR].supply[S].supplier.name['Repco']
      AND PR.name['Bicycle']                                    (16)
```

3.4 Comparison Operations

The usual comparisons of less-than, etc., are supported together with appropriate quantifiers for comparing bags of values. The default quantification is existential.

To find the SSNs of employees who work more than 10 hours on some project we would give the query:

```
employee[E].ssn WHERE E.works_on.hours > 10                     (17)
```

The intuitive meaning of this query is find the SSNs all employees such that they participate in some works_on relationship where the value of attribute hours is greater than 10.

To find the SSNs of employees who, for every project they work on, work more than 10 hours we, would give the query:

```
employee[E].ssn WHERE E.works_on.hours ALL> 10                  (18)
```

To find the SSNs of employees who work more hours on some project than anyone works on the `'Alpha'` project we would give the query:

```
employee[E].ssn WHERE project[P].name['Alpha']
      AND E.works_on.hours >ALL P.works_on.hours                (19)
```

In fact, constraints are a shorthand way of writing explicit equality comparisons, so the query

```
employee[E].ssn WHERE E.name.fname['Xavier']                    (20)
```

is a shorthand for the equivalent query with an explicit comparison.

```
employee[E].ssn WHERE E.name.fname[Name] AND Name = 'Xavier'    (21)
```

3.5 Bag Operations

Bag operations corresponding to the usual set operations of union, intersection, difference, etc. are provided. To find all the birthdates of employees and dependents we would give the query:

```
employee.bdate UNION dependent.bdate                            (22)
```

To find the employees who work on both the `'Alpha'` and the `'Beta'` projects we would give the query:

```
(project[P].works_on.employee WHERE P.name['Alpha'])
      INTERSECT (project[P].works_on.employee WHERE P.name['Beta']) (23)
```

Alternatively, we could give the equivalent query:

```
E WHERE employee[E].works_on.project.name['Alpha']
      AND E.works_on.project.name['Beta']                       (24)
```

3.6 Quantification

Quantification can be applied to any variables appearing in a condition. For example, to find employees who work on all projects we would give the following query.

```
employee[E] WHERE ALL P project[P].works_on[W].employee[E]           (25)
```

Variables not appearing on the left of the `WHERE` clause and not explicitly quantified are implicitly existentially quantified at the innermost enclosing level. (The constraint `[W]` would normally be omitted from the above query.) Thus, the above query is equivalent to the following query.

```
employee[E] WHERE ALL P SOME W project[P].works_on[W].employee[E]   (26)
```

3.7 Regular Expressions

Path expressions can also contain regular expressions. This allows queries requiring linear recursion (e.g., transitive closure). For example, the query

```
E.(supervisee.supervisor)+.bdate
      WHERE employee[E].name.lname['Wong']                          (27)
```

returns the birthdates of all employee Wong's direct and indirect supervisors.

The `'+'` indicates that at least one path must be traversed. For zero or more paths, the `'*'` is used. For example, the query

```
E.(supervisee.supervisor)*.bdate
      WHERE employee[E].name.lname['Wong']                          (28)
```

returns the same birthdates as query 27 together with Wong's birthdate.

The path expression enclosed by the parentheses must begin and end at the same point in the EER schema and may not contain any constraint terms.

3.8 Aggregate functions

The usual functions `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX` are available. The function `SET` removes duplicates from a bag (cf. the keyword `DISTINCT` in SQL). Unlike SQL, aggregate functions may be applied to arbitrary queries. For example, to find the average number of hours spent on projects by members of the `'Research'` department we would give the query:

```
AVG(employee[E].works_on.hours
      WHERE E.works_for.department.name['Research'])                (29)
```

To find project names and total number of hours worked on each project we would give the query:

```
P.name, SUM(P.works_on.hours) WHERE project[P]                      (30)
```

Note that there is no `GROUP BY` (or `HAVING`) clause in ERQL, and none is needed because the scoping rules for aggregates make them unnecessary.

In some cases, the use of `DISTINCT` in SQL, inside aggregates, can be avoided. For example, to find the names of employees who work on more than two projects, the number of projects they work on, and the average salary of the employees they supervise, we would give the query:

```
E.name, COUNT(E.works_on), AVG(E.supervisor.supervisee.salary)
      WHERE COUNT(E.works_on) > 2                                    (31)
```

Finally, aggregates can occur inside other aggregates. For example, to find the average number of employees assigned to projects we would give the query:

```
AVG(COUNT(P.works_on) WHERE project[P])                              (32)
```

It is not possible to express this query in SQL.

3.9 View Definitions

As with SQL, ERQL can be used to define views. Three kinds of views can be defined: relationships, entity subtypes, and attributes. To define a derived relationship, senior, between the entity employee and itself we would give the definition:

```
DEFINE RELATIONSHIP senior
      BETWEEN employee(superior), employee(inferior)
      AS S,I WHERE employee[S].(supervisor.supervisee)+[I]           (33)
```

Here, because the relationship involves an entity more than once, the role names, superior and inferior, must be specified. The entities participating in the relationship are determined by the query following the `AS`.

To define a derived entity, supervisor, as a subtype of the entity employee we would give the definition:

```
DEFINE ENTITY supervisor SUBTYPE OF employee
      AS E WHERE employee[E].supervisor                              (34)
```

Here, those entities satisfying the query following the `AS` are members of the derived subtype supervisor.

To define a derived attribute, tot_hours, of the entity employee we would give the definition:

```
DEFINE ATTRIBUTE tot_hours OF employee[E]
      AS SUM(E.works_on.hours)                                       (35)
```

Here, the derived attribute, tot_hours, gets its value from the query following the `AS`.

A query involving a view such as

```
E.superior.inferior
      WHERE employee[E].works_for.department[D].name['Research']     (36)
```

is evaluated as if it were rewritten as

```
I WHERE employee[E].(supervisor.supervisee)+[I]
      AND employee[E].works_for.department[D].name['Research']       (37)
```

4 Towards a formal definition of ERQL

For brevity, we present only the main features of a formal definition of ERQL. A complete description will appear in a forthcoming report.

4.1 Syntax

The grammar of ERQLs been omitted due to space restrictions, however it should be reasonably clear from the above examples. Readers should note the lack of arbitrary restrictions in the language, compared to those of SQL.

What may not be clear from the examples, however, are the many natural context sensitive conditions. The most important of these conditions are the following.

- Every name following a dot in a path must be the name of an entity type, attribute, relationship type, or role that is adjacent in the schema to the node determined by the preceding path.
- Every name list in a closure must start and end at the same entity type or relationship type.

To ensure that every query is well-defined, and returns only a finite set of answers, we require every query to be range restricted. For example, the queries `E WHERE E` and `E WHERE employee[E] OR department[F]` are not range restricted. To define the class of range restricted queries, we need some preliminary definitions.

A variable X *occurs free* in a query Q if X occurs at the head of a path expression in Q.

A variable X is *bound in a query* Q if X occurs in a constraint in the head of Q or is bound in the condition of Q.

A variable X is *bound in a condition* C if

- C is a path expression, and X occurs in a constraint in C;
- C has the form `SOME Y` C', and X is bound in C';
- C has the form C_1 `AND` C_2, and X is bound in C_1 or in C_2; or
- C has the form C_1 `OR` C_2, and X is bound in C_1 and in C_2.

Finally, a query Q is *range restricted* if every variable that occurs free in the head of Q or a subquery of Q is bound in the query or some enclosing query.

To simplify the description of the semantics of queries, it is convenient to assume that queries are in normal form, which we now define.

A query Q is in *normal form* if Q and every (simple) query that occurs in Q satisfies the following conditions.

- Every path expression in the head of the query is a variable.

- If some expression in the head of the query is a variable, then the query has a `WHERE` clause.

There is a straightforward transformation from arbitrary queries to normal form queries. It is arguable that normal form queries are clearer and that users should write only normal form queries.

4.2 Semantics

We can now define the meaning of a (normal form) query. This involves the definition of several mutually dependent concepts. For the remainder of this section, we assume that D is a database and that all queries are in normal form. Without loss of generality we also assume that each node in each path expression has a constraint, by adding constraints consisting of new variables, if necessary.

A variable assignment v *satisfies* a condition C wrt a database D if

- C is a path expression E, and some ground instance of $v(E)$ occurs in D;
- C is $Q_1 < Q_2$, where Q_1 and Q_2 are scalar queries, and for some answers A_1 to Q_1 wrt D and A_2 to Q_2 wrt D, $A_1 < A_2$ (and similarly for other relational operators);
- C is Q_1 `SUBBAG` Q_2, where Q_1 and Q_2 are simple queries, and the bag of answers to Q_1 wrt D is a subbag of the bag of answers to Q_2 wrt D (and similarly for other bag operators);
- C is `ALL` $X\,C'$, where C' is a condition, and for all bindings of objects o to X, $v[X/o]$ satisfies C' wrt D;
- C is `NOT` C', where C' is a condition, and v does not satisfy C' wrt D;
- C is C_1 `AND` C_2, where C_1 and C_2 are conditions, and v satisfies both C_1 and C_2 wrt D (and similarly for other logical operators); or
- C is the boolean constant `TRUE`.

Let E be a ground instance of a path expression. Then E either has the form $o.n_1[c_1].\ \ldots\ .n_k[c_k]$ or $n_0[c_0].n_1[c_1].\ \ldots\ .n_k[c_k]$, where each n_i is a node, and o and each c_i is an object. We say E *occurs* in D if there exists a sequence of objects $o_0, o_1, \ldots, o_m$ in D such that the following conditions hold.

- Either E has the form $o.n_1[c_1].\ \ldots\ .n_k[c_k]$, and o is o_0, or E has the form $n_0[c_0].n_1[c_1].\ \ldots\ .n_k[c_k]$, and $n_0[c_0]$ matches o_0.
- For each $i \in [1..k]$, $n_i[c_i]$ matches the pair (o_{i-1}, o_i).

The node n_0 is either a name or a closure. The name-constraint pair $n_0[c_0]$ *matches* the object o_0 if n_0 is the name of the entity type or relationship type to which o_0 belongs, and c_0 is o_0.A *database* for an EER schema S is a graph. The nodes in the graph are entities, relationships, and attribute values. For each entity e of type E, and for each attribute A of E, there is an edge from e to some value of A. If A is a multi-valued attribute there may be several edges from e to values of A. For each value of a composite attribute, there is an edge to a value of each of its component attributes (and possibly several edges for a multi-valued composite attribute). For each relationship r of type R, there is an edge from r to an entity e of type E (or subtype of E) for each entity type E participating in R, and there is an edge from r to a value of each attribute of R (and possibly several edges for a multi-valued attribute).

The closure $(n_{11}. \ \ldots \ .n_{1p})*$ (resp., $(n_{11}. \ \ldots \ .n_{1p})+$) *matches* the object o_0 if n_{11} is an entity type or relationship type, and there exists a ground path expression E' consisting of $k \geq 0$ (resp., $k > 0$) occurrences of $n_{11}. \ \ldots \ .n_{1p}$ such that E' occurs in D and o_0 is an answer to the query X `WHERE` $E'[X]$ wrt D.

Each node n_i in the node-constraint pair $n_i[c_i]$ is also either a name or a closure. The name-constraint pair $n_i[c_i]$ *matches* the pair (o_{i-1}, o_i) if n_i is the name of an entity type or relationship type or attribute adjacent to the entity type or relationship type to which o_{i-1} belongs, if n_i links o_{i-1} and o_i in D, and if c_i is o_i. The closure-constraint $(n_{i1}. \ \ldots \ .n_{ip})*$ (resp., $(n_{i1}. \ \ldots \ .n_{ip})+$) *matches* the pair (o_{i-1}, o_i) if there exists a ground path expression E' consisting of $k \geq 0$ (resp., $k > 0$) occurrences of $n_{i1}. \ \ldots \ .n_{ip}$ such that $o_{i-1}.E'$ occurs in D, if o_i is an answer to the query X `WHERE` $o_{i-1}.E'[X]$ wrt D, and if c_i is o_i.

We now define the value of an expression E under a variable assignment v wrt D of the free variables in E.

- If E is a variable X, the value of E under v wrt D is simply $v(X)$.
- If E is an aggregate expression $f(Q)$, let Q' be the query with no free variables $v(Q)$ and S be the bag of answers to Q' wrt D. Then the value of E under v wrt D is $f(S)$.

Finally we can define the semantics of a query by specifying the set of answers to the query with respect to a database. Let Q be the query $E_1, \ldots, E_n$ `WHERE` C. Let V be the set of variable assignments v such that v satisfies C wrt D. Then a tuple of values $(v_1, \ldots, v_n)$ is an *answer* to Q if v_i is the value of E_i under v wrt D, for each $i \in [1..n]$.

5 Discussion

We have defined and illustrated a simple, regular, powerful query language for the EER data model. Such a language simplifies the task of expressing ad hoc queries to a database by hiding implementation details, and serves as one step in the larger programme of modifying the database design and application activities to work solely in the conceptual model. An additional application of such a language is to serve as a query language for federated schemas that use the EER data model.

While ERQL is based on the concept of navigation by path expressions it is still a declarative language, in contrast to hierarchical and DBTG-network query languages. Furthermore, the navigation involved is equivalent to explicit joins on foreign keys in SQL, but in ERQL the user is relieved from having to know about keys and foreign keys and having to make the joins explicit.

There are several straightforward extensions to our language which we have not considered here. These include boolean queries and derived multi-valued attributes. Neither of these extensions is difficult. Extensions to an update language, and the inclusion of ERQL in a host application language, are more challenging. It would also be desirable to allow meta-queries, where variables may range over attribute, entity, relationship names and other schema information, in the manner of XSQL [10].

Other researchers have also proposed query languages for the entity-relationship model (see [9, 7] and many others cited in these papers). Many of these languages do not deal with EER extensions, or assume an underlying relational implementation, or are graphical in nature. The work most comparable to ours is that of of Hohenstein et al. who define a language called SQL/EER [9]. Their language is more complex than ours. It directly supports lists as well as sets and bags, and also supports user supplied functions and relations. They define an EER calculus, define a semantics for it [6], and define the semantics of SQL/EER by translation into their EER calculus. In other work [8] they describe a translation of their language into SQL.

In contrast, our query language is given a semantics directly in terms of path expressions, which we believe allows the user a clearer understanding of the language's semantics. Furthermore, we provide a simple and convenient means for expressing simple recursive queries such as transitive closures by means of regular expressions; we allow variable constraints in path expressions, simplifying many queries; and we provide a mechanism for defining views.

The work of Cruz et al. [3] describes a graph based query language with regular expressions over paths. It has the drawback of being based on a graph data-model which is likely to be unfamiliar to the user, and it is a graphical language which, while quite powerful, is likely to be difficult to use for complex queries.

We believe that our proposed language provides a good balance between simplicity and expressive power, and provides a promising basis for further work towards the goals listed above.

References

[1] C. Batini, S. Ceri, and S. Navathe. *Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1989.

[2] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: the notion and the issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–37, June 1992.

[3] I. Cruz, A. Mendelzon, and P. Wood. G+: Recursive queries without recursion. In *Proc. 2nd Intl Conf. on Expert Database Systems*, pages 645–66, 1989.

[4] C. Date. A critique of the SQL database language. *ACM SIGMOD Record*, 14(3), Nov. 1984.

[5] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[6] M. Gogolla and U. Hohenstein. Towards a semantic view of an extended entity-relationship model. *ACM Transactions on Database Systems*, 16(3):369–416, 1991.

[7] J. Grant, T. Ling, and M. Lee. ERL: Logic for entity-relationship databases. *Journal of Intelligent Information Systems*, 2(2):115–147, 1993.

[8] U. Hohenstein. Automatic translation of an entity-relationship query language into SQL. In F. Lochovsky, editor, *Entity-Relationship Approach to Database Design and Querying*, pages 303–21. Elsevier Science Publishers B.V., 1990.

[9] U. Hohenstein and G. Engels. SQL/EER – syntax and semantics of an entity-relationship-based query language. *Information Systems*, 17(3):209–242, 1992.

[10] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, pages 393–402, 1992.

[11] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. In *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, pages 40–49, 1991.