Bernd Meyer, Gerhard D. Westerman
Fernuniversität Hagen
LG Praktische Informatik IV
Postfach 940
D-5800 Hagen
meyer@fernuni-hagen.de, westerman@fernuni-hagen.de

Martin Gogolla
Technische Universität Braunschweig
Informatik, Abt. Datenbanken
Postfach 3329
D-3300 Braunschweig
gogolla@infbs.uucp

QUEER: A Prolog-based Prototype for an Extended ER Approach

Bernd Meyer, Gerhard D. Westerman & Martin Gogolla

Bericht Nr. 90-03

Abstract

The combination of logic programming and databases has been given broad attention in recent years. However, mainly the connections between the relational data model and logic programming have been investigated. The application of logic programming languages as a basis for information systems founded on more complex, i.e. conceptual data models has received comparatively less attention. With the system QUEER, we introduce the prototype of a PROLOG-based information system which directly supports an extended entity-relationship model on its front-end and uses a semantically well-founded query and manipulation language based on an entity-relationship calculus. The system basically consists of a set of compilers written in PROLOG which translate data specifications, schema definitions, queries, and data-manipulation statements into PROLOG programs.

Contents

1 Introduction

The combination of logic programming and databases has been given broad attention in recent years [GMN84, Nic85, BMSU86, App87, GDK87, Jas89]. However, mainly the connections between the relational data model and logic programming have been investigated; in this area some efficient systems have been developed. Nevertheless, the application of logic programming languages as a basis for information systems founded on more complex, i.e. conceptual data models has received comparatively less attention [Joh90, Wei86]. Though some so-called object oriented logic languages have been designed [AG87, AKN86, CW89, KL89, KW89, Gua86, Omo86, Kup87], there are only few *systems* that exploit the advantages of logic programming for the implementation of more conceptually oriented databases. On the other hand the growing interest in non-standard systems, e.g. for geographic applications, requires databases, which do neither force the user to remain on the low conceptual level of the relational data model nor to translate a more expressive conceptual model into it. Proceeding this way, another kind of the so-called "impedance mismatch" [Mai84] would be introduced.

With the development of QUEER[1] we hope to take some steps into a conceptually oriented direction. QUEER directly supports an extended ER model [HNSE87] on its front-end, thus overcoming the necessity of transforming schemas between different data models. Our aim has not been to provide a logic programming language with database support, but rather to build a database prototype with a clean, calculus oriented, and highly expressive front-end exploiting the power of logic programming. In contrast to related projects [Joh90, Wei86] we do not only focus on conceptual modelling of data, but also present a formal translation from a query and a data manipulation language to PROLOG. For the query language we choose a particular calculus, which has been developed especially for the EER model and which is semantically well-founded, safe, and relationally complete [HG88, GH91]. The fact that the query language has a precisely defined semantics makes it easier to implement higher level query languages or user interfaces, like second-order-query languages. The calculus can be considered an interim layer, into which the new language is to be translated, so that it does not need to be implemented in terms of a programming language.

Due to this language design no knowledge about the internal structures of the database is required to build and use a database. In any phase the user can completely think in terms of her conceptual model while interacting with the system.

The basic idea behind the system is to store database objects as PROLOG terms and to translate queries expressed in terms of the EER calculus into PROLOG goals, thus using the inference engine for data retrieval. Internally the data is treated as unnormalized, nested relations, where objects are seen as complex tuples and relating of objects is done by means of surrogate handling. The idea to translate the EER model and calculus into PROLOG programs has been first stated in [Gog89]. This approach has been implemented and tested in [Mey90, Wes90, Con90].

QUEER is entirely written in PROLOG and can thus readily be embedded in PROLOG application programs providing a precisely defined interface for database interaction.

The structure of this paper is as follows. In section 2 we present a brief informal introduction to our approach by an example of a simple geo-scientific application. Section 3 introduces the basic concepts of our entity-relationship model and calculus. An overview of the system components of QUEER can be found in section 4 and the details of the translation are covered by section 5. Data manipulation is treated in section 6. Some methodical problems arising from translating the entitiy-relationship model to PROLOG are discussed in section 7 and some concluding remarks are given in the final section.

2 The Basic Idea

This section presents a brief informal description of our approach by an example of a simplified geoscientific application. We want to model information about different kinds of bodies of water, about countries, towns, and the leading people of countries and towns. Therefore, we would like to represent the following facts:

- Every TOWN lies in a COUNTRY and may lie at one or several RIVERS.
- RIVERS flow through COUNTRIES and flow into some kind of WATERS.
- SEAS, RIVERS, and LAKES are such WATERS.

[1] **QUEER** stands for **QU**ery system for an **E**xtended **E**ntity **R**elationship model.

- A PERSON may be the mayor of a TOWN, or the boss or a minister of a COUNTRY.

Beside usual properties of entities, like the name of a town, the population of a country, or the list of addresses of a person, we additionally associate with every geographic object a geometry in the world's co-ordinate system:

- Countries are represented by a set of closed polygons (representing their regions),
- towns and lakes by circles,
- seas by closed polygons, and
- rivers by connected, non-overlapping lines.

We describe this situation with an extended entity-relationship model and give a corresponding entity-relationship diagram in figure 1.
Let us now explain how this entity-relationship schema is mapped into PROLOG code. The non-standard data types used in this example are described textually and the following lines are part of their specification (the details of the specification are explained in following sections; for more motivation concerning the background we refer to [HG88, GH91]).

```
specification geo_data_types
sorts         point, circle
constructors  point  = tuple( real, real )
              circle = tuple( point, real )
selectors     radius : circle -> real
operations    pdist  : point x point -> real
equations     pdist(P1, P2) := sqrt( (P1.x - P2.x)^2 + (P1.y - P2.y)^2 )
end_specification geo_data_types
```

The translation of the above input into PROLOG looks as follows. We recognize for every specified sort a PROLOG ground fact of predicate `sort` giving the name of the sort, a one-step expansion of its definition and its complete expansion.

```
sort(point, tuple([real, real]), tuple([real, real])).
sort(circle, tuple([point, real]), tuple([tuple([real, real]), real])).
```

A correct representation for a circle is for instance `((1.1,2.2),99.9)`. The syntactic information concerning the selector `radius` and the operation `pdist` is represented by a ground fact of predicate `db_opn` specifying the argument types and the result type of these functions.

```
db_opn(radius, [circle], real).
db_opn(pdist, [point, point], real).
```

The semantics of the function `radius` is given by a unit clause selecting the second argument of a circle. E.g. `radius(((1.1,2.2),99.9),Result)` yields `Result = 99.9`.

```
radius((Center, Result), Result).
pdist(P1, P2, Result) :-  x(P1, X1), x(P2, X2), y(P1, Y1), y(P2, Y2),
                          Result := sqrt((X1 - X2) ^ 2 + (Y1 - Y2) ^ 2).
```

The semantics of the function `pdist` (determining the distance between two points) is a PROLOG rule which evaluates the function for given `P1, P2`. Just as the data types, the entity-relationship schema is given to our system in textual form. Part of the given schema looks as follows.

```
schema country_town_waters
entity types  country  attributes key:  cname       : string,
                                        area        : regions,
                                        cpopulation : int,
                                        government  : forms;
```

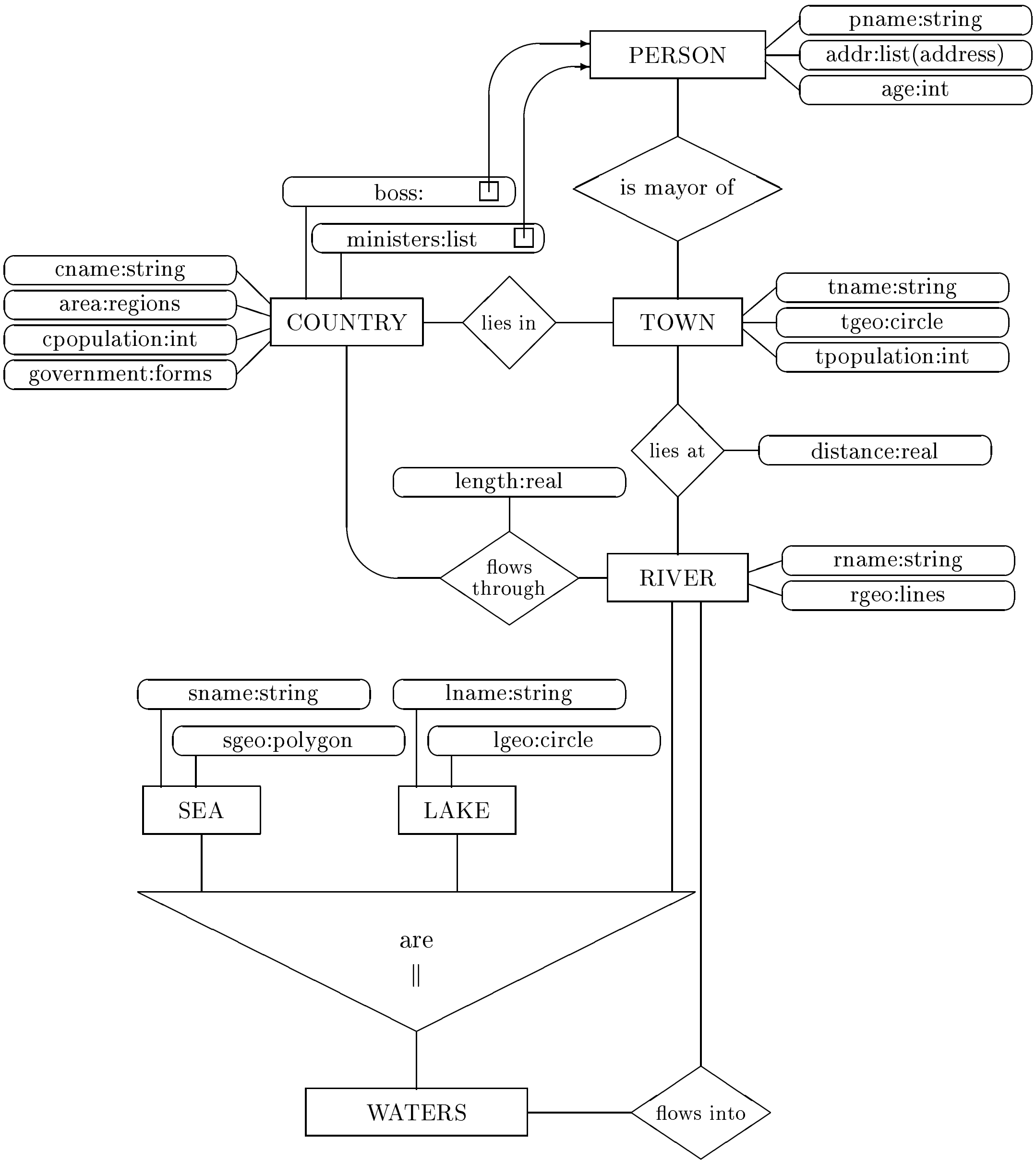


Figure 1: Diagram of the example ER schema

```
                        components      boss        : person,
                                        ministers   : list(person);
    relationship types
            flows_through   participants    river, country;
                            attributes      length : real;
    end_schema country_town_waters.
```

In Prolog, we introduce a ground fact of the predicate `entity_type` for every entity type specifying its name, its attributes together with their data types, and the components together with their types.

```
entity_type( country,
             [ ( cname, string ), ( area, regions ),
               ( cpopulation, int), ( government, forms )
             ],
             [ ( boss, person ), ( ministers, list(person) ) ]
           ) .
```

Entities are internally identified by surrogates. These surrogate values are supervised by the system and are invisible for the user. Thus, if one wants to insert e.g. the country France, a new surrogate value, for instance `country42`, is created which represents France. The actual information concerning an entity is characterized by ground facts of the predicate `entity` matching the entity type definition, e.g.:

```
entity(country,[country42,"france",...,52.900.000,democratic],
               [person17,[person65,person12,...]]).
```

For every attribute and component a rule is generated which computes the value of the attribute or the component, resp. when given a corresponding surrogate.

```
cpopulation(Surrogate, Result) :-
   entity( country, [ Surrogate, _, _, Result, _ ], _ ) .
ministers(Surrogate, Result) :-
   entity( country, [ Surrogate | _ ], [ _, Result ] ) .
```

E.g. `cpopulation(country42,Result)` yields `Result = 52.900.000`. Relationship types are syntactically described by a ground fact of predicate `relship_type` giving the name of the relationship type, the role names of its participating entity types and its attributes together with the attributes' data types.

```
relship_type( flows_through, [ ( river, river ), ( country, country ) ],
                             [ ( len, real ) ]
            ) .
```

Relationships are represented by ground facts of predicate `relship`, e.g.

```
relship(flows_through, [river13,country42], [436]).
```

Here, the surrogate `river13` could represent for instance the river Seine (and `country42` the country France). Attributes and participants of relationship types are translated into rules analogously to attributes of entities.

```
country( (flows_through, [River, Country]), Country) :-
   relship( flows_through, [River, Country], _ ) .
len( (flows_through, SurrList), Len ) :-
   relship( flows_through, SurrList, [Len] ) .
```

The query language we use is the extended entity-relationship calculus [HG88, GH91] which is roughly speaking a tuned version of the well-known tuple calculus suited for the extended entity-relationship model. For instance, if we want the names of all persons under age 18 we denote this query in the calculus in the following way.

```
-[ pname(P) | (P: person) & age(P) < 18 ]-
```

The Prolog equivalent is a rule employing the standard predicate `bagof`. `bagof(Term, Goal, Result)` collects all instances of `Term` satisfying `Goal` in the list `Result`.

```
query(Result) :- bagof( PName,
                        ( entity(person, [Person|_], _),
                          age(Person, Age),
                          Age < 18,
                          pname(Person, PName)
                        ),
                        Result
                      ).
```

Rules of this form are generated automatically from calculus expressions. We emphasize that the complete, rather complex calculus is translated in our approach. In particular we allow arbitrary nested subqueries and aggregation functions. All missing details of this and the previous examples will be explained later.

Summarizing, we observe that we employ an extended entity-relationship model for the conceptual modelling process and we translate the result into PROLOG code. Entities and relationships between entities are represented by ground facts, attributes and other concepts of the model by rules. The list, bag and set constructs are represented as PROLOG lists. Furthermore, we use an entity-relationship calculus to express queries and to formulate ad-hoc integrity constraints which a given database state should fulfill. The calculus allows subqueries and their PROLOG counterparts are computed employing the standard predicate `bagof`. Let us finally give some remarks on the relation between our entity-relationship calculus and the classical tuple or domain calculus. Firstly, everything that can be expressed in the classical calculi can be formulated in this language as well. But the expressiveness of the entity-relationship calculus goes beyond those possibilities, because due to the existence of aggregation functions we can compute e.g. the cardinality of a relation. Another main difference is that the classical calculi are flat in the sense that set expressions can appear only on the top level of a query. In the entity-relationship calculus, set and especially bag expressions are allowed as subqueries in result terms and in formulas, and they can be used to bind variables (see [HG88, GH91] for details).

3 The Extended Entity-Relationship Model and Calculus

Here, we shortly recapture the fundamental notions of the underlying EER model and calculus. We only present a sketch of the syntax and give examples, because all technical details and the formal semantics can be found in [HG88, GH91]. There, also full motivation for all concepts can be found.

3.1 The EER Model

3.1.1 Data Types

Our EER model does not only allow standard predefined data types like *int*, *real*, or *string*, but also arbitrary application dependant types. For instance, it is possible to define and use geometrical types like *point*, *circle*, or *lines* together with appropriate operations and predicates.

Formally, we have a data signature DS consisting of

- finite sets DATA, OPNS, PRED giving the names of the data sorts, operation and predicate symbols, respectively, together with
- functions *source* : OPNS $\rightarrow$ DATA*, *destination* : OPNS $\rightarrow$ DATA, and *arguments* : PREDS $\rightarrow$ DATA$^+$ giving the source and destination types of function symbols and argument types of predicate symbols.

Roughly speaking, data signatures are interpreted by sets, (partial) functions, and predicates.

3.1.2 Parametrized Data Types

For modelling purposes flat data types do not suffice. Certain standard constructors are useful in the context of data types (and also in the context of entity types as we will see later). Therefore, we allow to use tuple-, set-, list-, and bag-valued parametrized types to be applied to already defined types; bags (or multi-sets) allow that one element occurs more than once in the collection of elements. Again, these types are characterized by (the names of) their sorts, operations, and predicates. Given already defined types s, s_1, ..., s_n (or predefined types like *nat*, *int* or *string*) we introduce derived sorts *list*(s), *bag*(s), *set*(s), and *tuple*(s_1, ..., s_n).

These constructors are used to define application dependant types, e.g.

$$\begin{aligned} &point &&= tuple(real,real) \\ &circle &&= tuple(point,real) \\ &lines &&= list(point) \end{aligned}$$

Additionally, functions and predicates are allowed, e.g.

$$\begin{array}{lll} x,\ y & : point & \rightarrow real \\ center & : circle & \rightarrow point \\ point\_distance & : point \times point & \rightarrow real \\ circle\_circle\_cut & : circle & \rightarrow circle. \end{array}$$

3.1.3 Aggregation Functions

For the computation of aggregated values some standard functions are provided:

$$\begin{array}{lll} cnt & : list(\mathrm{s}) & \rightarrow int \\ sel & : list(\mathrm{s}) \times nat & \rightarrow \mathrm{s} \\ lts & : list(\mathrm{s}) & \rightarrow set(\mathrm{s}) \\ apl_\sigma & : list(\mathrm{d}) & \rightarrow \mathrm{d} \end{array}$$

cnt counts the number of elements in a list, *sel* selects the i-th element of a list, *lts* converts a list to a set, and apl_σ applies an operation σ to a list, where $\sigma : \mathrm{d} \times \mathrm{d} \rightarrow \mathrm{d}$ is a binary operation on a data type $\mathrm{d} \in$ DATA. Most functions (e.g. *cnt* and *apl*) are used analogously for the bag- or set-valued case. The *apl* functions are employed in order to define aggregation functions. For instance, if we want to define the *sum* of a set x of integers, we simply specify

$$sum(\mathrm{x}) \equiv apl_+(\mathrm{x})$$

In the same way we use the aggregation functions *min*, *max*, and *avg* computing the minimum, maximum, and average of a collection of numerical values. There are also standard predicates like $in : \mathrm{s} \times list(\mathrm{s})$, which determines whether a given element is element of a given list (analogously for bags and sets).

3.1.4 Extended Entity-Relationship Schemas

The central notion of the EER model is the extended Entity-Relationship schema. Such a schema provides names for the entity (or object) types, for the relationship types between these entity types, and for the attributes of entities. It is also possible to define components (or in other words object-valued attributes). Attributes and components are allowed to be list-, bag-, or set-valued.

Formally, an extended Entity-Relationship schema consists of

- finite sets ENTTYPE, RELTYPE, ATTRIBUTE, COMPONENT, and CONSTRUCTION together with functions
- *participants* : RELTYPE $\rightarrow$ ENTTYPE$^+$,
- *source* : ATTRIBUTE $\rightarrow$ ENTTYPE $\cup$ RELTYPE,
- *destination* : ATTRIBUTE $\rightarrow \{\ \mathrm{d},\ set(\mathrm{d}),\ bag(\mathrm{d}),\ list(\mathrm{d}) \mid \mathrm{d} \in \mathrm{DATA}\ \}$,
- *source* : COMPONENT $\rightarrow$ ENTTYPE,
- *destination* : COMPONENT $\rightarrow \{\ \mathrm{e},\ set(\mathrm{e}),\ bag(\mathrm{e}),\ list(\mathrm{e}) \mid \mathrm{e} \in \mathrm{ENTTYPE}\ \}$, and
- *input*, *output* : CONSTRUCTION $\rightarrow 2^{\mathrm{ENTTYPE}} - \emptyset$.

There are several additional conditions these schemas have to fulfil: for two distinct construction types the output types must be disjoint; no cycles are allowed for construction types, i.e. it is not allowed that there is an entity type being constructed from itself. These conditions guarantee a unique construction path for every entity.

Let us explain these notions using our running geographic example. Here, we have the following identities.

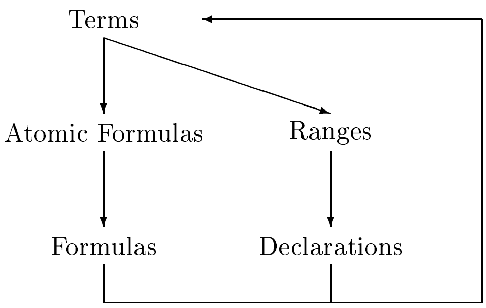
- ENTTYPE := {PERSON, COUNTRY, TOWN, RIVER, SEA, LAKE, WATERS}
- RELTYPE := {*is_mayor_of, lies_in, lies_at, flows_through, flows_into*}
- ATTRIBUTE := {*pname* : PERSON $\rightarrow$ *string*, *addr* : PERSON $\rightarrow$ *list*(*address*), ...}
  The above line specifies not only the set ATTRIBUTE but the functions *source* and *destination* for ATTRIBUTE as well.
- COMPONENT := {*boss* : COUNTRY $\rightarrow$ PERSON,
  ministers : COUNTRY $\rightarrow$ *list*(PERSON)}
- CONSTRUCTION:= {*are* (SEA, LAKE, RIVER ; WATERS)}

The interpretation of an EER schema associates finite sets of surrogate values to the entity types and finite subsets of the cartesian products (of the corresponding surrogate values) to the relationship types. Attributes are interpreted by finite functions from the surrogate values of the entity types into (the interpretation of) the (possibly multi-valued) data types. Components are finite functions from the surrogate values of the source entity types into surrogate values or bags, lists, or sets of surrogates of the destination entity types. Construction types correspond to injections from the union of the output surrogates into the union of the input surrogates. Thus, for example, the injection for the construction *are* gives the SEA, RIVER, or LAKE corresponding to a given entity of type WATERS. All functions corresponding to attributes and components are partial: apart from data values or surrogates they may yield an undefined value $\perp$.

Let us give a short, explaining remark concerning construction types, which support the well-known abstraction concepts specialization, generalization, and partition. These concepts are covered as special cases of this more general approach. In the case of specialization, we have one input type and one output type, and the injection gives the corresponding entity (in the superclass) of the specialized entity. In the case of generalization, there are more than one input type (like the above SEA, RIVER, and LAKE) and one output type (like the above WATERS). In the case of a partition, we have one input type, e.g. PERSON, and more than one output type, e.g. WOMAN and MAN. The injection from the union of the entities of the types WOMAN and MAN into the entities of type PERSON guarantees that each PERSON is either a WOMAN or a MAN.

3.2 The EER Calculus

The EER calculus defined for the above introduced EER model allows to formulate queries against an EER database (i.e. an interpretation or instance of an EER schema). But it also can be employed to formulate integrity constraints. Its main building blocks are terms, atomic formulas and formulas analogously to the classical domain or tuple calculus.



However, the main difference is that our calculus is not hierarchical: we additionally introduce ranges, which are special set-valued terms, and we allow to bind variables in declarations to these ranges; on the one hand ranges and formulas can again be used for the formulation of bag-valued terms, on the other hand there are special aggregation functions converting multi-valued terms into single-valued ones. This will be explained by an example at the end of this section. We now introduce the different syntactic categories of the calculus and give some examples.

3.2.1 Simple Declarations

We start with a first simple form of declarations binding variables to finite sets. The allowed forms are $v : e$ with $e \in$ ENTTYPE or $v : r$ with $r \in$ RELTYPE. Thus, p : PERSON or li : *lies_in* are correct declarations. Throughout the following examples, we assume variables for the entity types of our running example consisting of a single lower case letter like the above p and variables for the relationship types consisting of at least two lower case letters like the above *li*.

3.2.2 Terms

There are various possibilities for the formulation of terms.

- Variables and application of operation symbols are terms. Operation symbols in this context are: data operations, standard operations of the parametrized types (like *cnt* or *lts*), attributes and components. For instance, *age*(p)+42 is a term of sort *int*, *cnt*(*addr*(p)) is of sort *int*, *boss*(c) is of sort PERSON, and *point_distance*($point_1$,$point_2$) is of sort *real* (with $point_1$ and $point_2$ variables of data type *point*).

- If we have a type construction, then $e_{in}(t_{out})$ and $e_{out}(t_{in})$ are terms of sort e_{in}, respectively e_{out} assuming e_{in} is an input type and e_{out} an output type of the construction. For example, RIVER(w) is a term of sort RIVER assuming a variable w of type WATER.

- If r is a relationship type with n argument types and v is a relationship variable for r, then $v.i$ is a term for $i \in 1..n$. For instance, *li.1* and *li.2* are terms of sort TOWN and COUNTRY, respectively. Additionally, role names are also allowed in this place instead of numbers (e.g. *li*.TOWN or *li*.COUNTRY).

- The most important form of term is the following.

  -[t_1, ..., t_n | d_1, ..., d_k $\wedge$ ϕ]-

  The value of this expression is a bag (or multi-set) allowing multiple occurrences of elements. The term has the sort *bag*(*tuple*(sort(t_1),...,sort(t_n))). Here, the t_is are the result terms, the d_is are variable declarations and ϕ is a qualifying formula. In the case n=1, the sort of the term is simply *bag*(sort(t_1)). For example,

  -[*age*(p) | p : PERSON]-
  -[*tname*(t), *cname*(c) | t : TOWN, c : COUNTRY $\wedge$ *lies_in*(t,c)]-

  are terms of sort *bag*(*int*) and *bag*(*tuple*(*string*,*string*)), respectively. They compute the multiset of ages of currently existing persons and the multiset of pairs <town name, country name>, such that the town lies in the country. If the town name is not unique within one country then the pair <town name, country name> appears more than once in the bag. Only declared variables are allowed to appear in result terms and the cardinality of the resulting bag is mainly determined by the different bindings for these declared variables.

Please note, terms are allowed to have free variables.

3.2.3 Formulas

Most of the rules for building formulas in the calculus are quite standard.

- The application of predicate symbols to terms yields correct formulas. Predicate symbols in this context are data predicates, standard predicates of the parametrized types, or names of relationship types. For example, *circle_circle_cut*($circle_1$,$circle_2$) with $circle_1$ and $circle_2$ variables of data type *circle*, *p in ministers*(c), or *lies_in*(t,c) are formulas.

- For a term t, *undef*(t) is a formula testing the definedness of the term. *undef*(*boss*(c)) is a correct formula and yields true, if the *boss* of country c is unknown (in the current interpretation).

- In the context of type constructions t_{in} *is* t_{out} is a formula. For instance, *r is w* yields true, if the value of the variable r of type RIVER is equal to the value of the variable w of type WATERS.

- Apart from the above mentioned possibilities for formulas, the calculus also allows equality between terms with equal types and the usual logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) and quantifiers ($\forall$, $\exists$). Thus for instance,

  $ltb(ministers(c))$ = -[p | p : PERSON $\wedge$ $age(p) > 50$]-
  ($\forall$ t : TOWN) ($\exists$ c : COUNTRY) $lies\_in(t,c)$

  are formulas (*ltb* is another standard operation: it converts a List To a Bag).

Please note, like terms, formulas are allowed to have free variables.

3.2.4 Ranges and Final Form of Declarations

Ranges will be used to restrict the domain of a variable to a finite set of currently existing values.

- The first two simple forms of ranges have already been introduced in the context of simple declarations: e with $e \in$ ENTTYPE or r with $r \in$ RELTYPE are ranges.

- The other possibility for ranges are set-valued terms. For example,

  $lts(ministers(c))$
  bts -[$age(p)$ | p : PERSON $\wedge$ $age(p) \geq 21$]-

  are ranges (recall *lts* converts a List To a Set and *bts* a Bag To a Set).

We have started with a simple form of declaration for the ease of demonstration. The final form of declaration now employs ranges in order to bind variables to arbitrary finite domains, not just entity sets or relationships as in the preliminary form of declaration.

- Variables can be bound to the union of a finite number of ranges with equal types. For example,

  p : $lts(ministers(c))$
  $string_1$: bts -[$rname(r)$ | r : RIVER]- $\cup$ bts -[$sname(s)$ | s : SEA]-

  are declarations.

- Sequences of declarations of the first form are allowed as well. The declared variables of declarations on the right may occur free in declarations on the left. Thus for instance,

  t : -[t' | t' : TOWN $\wedge$ $lies\_in(t',c)$]- ; p : $lts(ministers(c))$; c : COUNTRY

  is a correct declaration of the variables c, p and t. First, the variable c is bound to a currently existing country (surrogate), then the variable p is bound to a minister of this country and the variable t to a town lying in this country.

Please note, like terms and formulas, ranges and declarations are allowed to have free variables.

3.2.5 Queries and Integrity Constraints

Queries in our calculus are now simply terms over data types without free variables and integrity constraints are formulas without free variables. As a simple example, consider the following integrity constraint requiring that the average age of a country's ministers has to be less than or equal to 65.

$\forall$ c : COUNTRY (avg -[$age(p)$ | p : PERSON $\wedge$ p *in* $ministers(c)$]- ≤ 65)

This example reveals the non-hierarchical nature of the calculus: the formula 'p *in* $ministers(c)$' appears inside the term '-[$age(p)$ | p : PERSON $\wedge$ p *in* $ministers(c)$]-'; this term in turn is part of the complete formula.

4 Overview of the System Components

QUEER consists of a set of compilers, which translate the four different languages necessary for the usage of the system into PROLOG programs. Once compiled these programs can be executed by a standard PROLOG system. PROLOG has been used as the implementation language as well as the target language of each compiler. Therefore it becomes trivial to embed the system in application programs written in PROLOG; thereby the "impedance mismatch" [Mai84] is overcome. The drawback of this technique is that we obtain a pure main storage prototype, supporting secondary storage only to the extent the underlying PROLOG implementation does. However, we feel that this problem should be solved on the level of PROLOG implementations, otherwise the major advantages of using PROLOG would have to be given up. Enhancements of the language in this respect have been claimed by several other research groups, too [P+84, BJ84, AB85, App87, Jas89].

No specialized user interface is provided, instead the standard PROLOG front-end is used, there the compilers can be invoked by predefined PROLOG predicates. The four different languages of QUEER are the following:

DSL Data specification language for arbitrary user-defined data types, operations, and predicates having these types as domains.

SDL Schema definition language for the description of EER Schemata using user-defined data types.

CALC Query language on the basis of the EER Calculus.

DML Data manipulation language, used for updating entities, relationships, and constructions. In contrast to the other languages, this one is not completely compiled, but partially interpreted, in order to be able to react interactively to violations of integrity constraints.

A typical session, starting from the scratch, consists of four steps, each of which requires one of these languages.

In accordance with a three level specification paradigm of database modelling [EDG86, HNSE87] the first step to be taken is to formulate the data type specification in *dsl*. These definitions are translated into a PROLOG program by the *dsl* Compiler. Following this, the schema specification together with the output of the *dsl* compiler is translated by the *sdl* compiler, obtaining a complete specification of the database schema expressed in PROLOG terms. Due to its complexity, the development layer, i.e. the specification of dynamic integrity constraints, is not provided. A concrete database may then be created and changed using *dml* commands. Likewise, *dml* commands could be translated in advance, thus establishing the action layer of the system by a set of fixed manipulations. The system does not support this by now, however, it would be easy to implement such a feature. *calc* is used to query the database state in the first place, but it can be used to define views onto the database by means of pretranslated queries or to test integrity constraints as well. Having been translated into PROLOG programs by the *calc* compiler, these queries can be run on the PROLOG machine without further intervention of any part of QUEER.

Roughly speaking, the first two steps are used to generate the representation of the database schema (or the *meta-database*), while the last two steps are the normal way of using the database. The first two steps have to be repeated only if the schema or the data types have to be changed.

Note that none of the above steps requires any knowledge of the internal structure of the database, rather the languages are completely based upon the conceptual schema and the EER calculus.

5 Details of the translation

While the compilers for the *dsl* and *sdl* produce static code, i.e. code which is generated only once during the creation of the database schema and is not changed afterwards, the *dml* interpreter changes the database while running the system. We therefore divide the database into two distinct parts: the *meta database* and the *factbase*.

The factbase contains information on the contents of the database without any structural information, i.e. information on the schema and data types are stripped. In contrast, the meta database contains only this structural information on the schema and data types, though a small amount of dynamic *structural* information, like surrogate and crossreference tables, is covered there, too. This approach was taken for two reasons: At first to remove redundancy from the database; secondly to facilitate an efficient way of type checking without having to access the actual database. Type checking is entirely done at compile time, because there are no objects in the languages that can change their types at run time — not even in the data manipulation language. Every term of the calculus — and with this, every expression in the *dml* — evaluates to a unique type, independent on the contents of the database. Thus an expression, which is validated at compile time cannot produce typing errors at run time.

The meta database and the factbase are organized in an orthogonal fashion for data types and object types. Therefore the same procedures can be used for verifying the types of data and objects, respectively. Not even the procedures used for applying functions to data values and object values have to be structurally different. E.g. there is no difference between the structure of the procedures required to select a component of a tuple and of those used for selecting an attribute of an entity.

Therefore functions generated by the *sdl* compiler and the *dsl* compiler have quite a similar appearance.

5.1 Data Specifications

The data specification language allows to define any data type derived from the base types *int*, *real*, and *string*. Every sort consists of nested applications of the type constructors *list*, *set*, *bag*, and *tuple* to some sorts defined before. Enumeration sorts can be specified by means of operation definitions without argument sorts.

Due to the fact that PROLOG is a type-free language and offers no concept of sets or bags at all, it is impossible to generate type declarations from such definitions, which can be handled by the PROLOG interpreter itself. Therefore, sets and bags have to be implemented by lists[2]. The definitions have to be translated into sort information in the form of PROLOG terms, which can be checked by the compilers for schema definitions and queries explicitly. Thus the type consistency of definitions, queries, and manipulations can be checked already at compile time.

Beside this sort information executable PROLOG code has to be generated from the operation and predicate definitions as well as the declarations that specify the argument (and destination) sorts. For each of the operations and predicates a default clause is generated that handles the case that some of the arguments are undefined ($\perp$). Then the specific operation has to return the undefined value, the predicate must fail.

Before discussing the translation of a data specification in detail we present a short example of a specification and its translation.

Input

```
specification geo

sorts   point, circle, lines, forms;

constructors
        point   = tuple( real, real ),
        circle  = tuple( point, real ),
        lines   = list( point );

selectors
        x              : point         -> real,
        y              : point         -> real,
        center         : circle        -> point,
        radius         : circle        -> real,

operations
        socialistic    :               -> forms,
        democratic     :               -> forms,
        monarchic      :               -> forms,
        pdist          : point x point -> real,   /* point-point distance */

predicates
        cccut          : circle x circle;         /* circle-circle cut    */
```

[2] with sorted elements to achieve more efficient implementations of several operations

```
equations
        pdist( P1, P2 ) := sqrt( (P1.x - P2.x)^2 + (P1.y - P2.y)^2 ),

conditions
        cccut( C1, C2 ) iff
                pdist( C1.center, C2.center ) <= C1.radius + C2.radius;

end_specification geo.
```

Output

```
sort(real, real, real).
sort(int, int, int).
sort(string, string, string).
sort(point, tuple([real, real]), tuple([real, real])).
sort(circle, tuple([point, real]), tuple([tuple([real, real]), real])).
sort(lines, list(point), list(tuple([real, real]))).
sort(forms, forms, enum([dictatorial, socialistic, democratic, monarchic])).

db_opn(x, [point], real).
db_opn(y, [point], real).
db_opn(center, [circle], point).
db_opn(radius, [circle], real).

db_opn(socialistic, [], forms).
db_opn(democratic, [], forms).
db_opn(monarchic, [], forms).
db_opn(pdist, [point, point], real).

x((X, _), X).
y((_, Y), Y).
center((Center, _), Center).
radius((_, Radius), Radius).

monarchic(monarchic).
democratic(democratic).
socialistic(socialistic).

pdist(P1, P2, Res)      :-  x(P1, _0),
                            x(P2, _1),
                            y(P1, _2),
                            y(P2, _3),
                            Res := sqrt((_0 - _1) ^ 2 + (_2 - _3) ^ 2).

db_pred(cccut, [circle, circle]).

cccut(C1, C2)           :-  center(C1, _0),
                            center(C2, _1),
                            pdist(_0, _1, _2),
                            radius(C1, _3),
                            radius(C2, _4),
                            _2 .<=. _3 + _4.
```

This example will be explained in the following.

5.1.1 Sorts

The majority of access during the translation of queries and manipulation statements consists of sort conversions by one level, e.g. accessing the element sort of a list sort or vice versa, since the calculus expressions are parsed recursively. Thus the sort information to be generated is the sort name itself, its one-step expansion, and its complete expansion, which is needed in another transformation step. This Information is represented by a PROLOG fact

```
sort(SortName, OneStepExpansion, CompleteExpansion).
```

E.g. for the sort *circle* the fact

```
sort(circle, tuple(point, real), tuple(tuple(real, real), real)).
```

is constructed. Formally, for any sort $sort_i \in DATA$ a clause

$$sort(sort_i,\ sexpr_i,\ expand(sexpr_i))$$

is generated, where *expand*: $SORT\text{-}EXPR_{DS} \rightarrow SORT\text{-}EXPR_{DS}$ is defined by

$$expand(s) = \begin{cases} s & \text{if } s \in \{int, real, string\} \\ set(expand(s_0)) & \text{if } s = set(s_0) \\ bag(expand(s_0)) & \text{if } s = bag(s_0) \\ list(expand(s_0)) & \text{if } s = list(s_0) \\ tuple([expand(s_0), \ldots, expand(s_n)]) & \text{if } s = tuple(s_0, \ldots, s_n) \end{cases}$$

In addition, the third argument of `sort` can be of the form

$$enum([r_1, \ldots, r_n])$$

for enumeration types.

5.1.2 Operations

For a component selector of a tuple sort (specified in the **selectors** part) or every other operation (specified under **operations**) a declaration of the form

```
db_opn(Operation, Source, Destination)
```

is generated which gives the operation name *Operation*, the list of its argument sorts *Source*, and its destination sort *Destination*, e.g. for the selector *radius* or the operation *pdist*

```
db_opn(radius, [circle], real).

db_opn(pdist, [point, point], real).
```

This information is needed during the translation of calculus terms to check whether the operands have the appropriate sorts.

The **equations** part of the data specification contains a function definition for each operation. The right-hand side of these equations are calculus terms without quantifiers and without free variables other than the parameter variables. Thus the operations can be translated to PROLOG predicates with one additional argument which is bound to the result of the operation on evaluation. The result variable in the rule head is unified with the result variable of the right-hand side term, which is translated in the same way as any calculus term (see section 5.3). An example is the operation *start*: $lines \rightarrow point$ which is defined as

$$start(l) := SEL(1, l).$$

A clause

```
start(L, Res) :- sel(1, L, Res).
```

will be generated in this case.

In the case of a selection operation a binary predicate is generated that simply executes a unification and takes the form

$$sel_i(Template,\ Res)$$

where *Template* is a tuple with n components whenever the associated tuple sort has n components if the tuple sort has n component sorts. *Res* appears at the position of the selected component, all other positions are anonymous variables. Thus for the selection of the component *radius* from the tuple sort *circle* the following code is generated:

```
radius((_, Radius), Radius).
```

It is assumed that for every tuple sort selectors are specified in the order of the arguments of the tuple.

Operations without arguments have to be considered as constants either of an enumeration sort or of another sort. In the latter case a term for their value has to be given on the right-hand side of an equation. This value is computed from the translated term. In order to be able to treat all operations uniformly, facts of the following form are generated for constants:

$op(value)$

If *op* is an enumeration constant *value* is equal to *op*, e.g. `monarchic(monarchic)`.

5.1.3 Predicates

Predicates can be handled analogously to operations. For each of them a declaration of the form

```
db_pred(Predicate, Arguments)
```

is generated that specifies for the predicate *Predicate* its argument sorts *Arguments*.

Analogously to operations, a predicate is defined by a condition. For each of these conditions a rule with the same arity as the defined predicate is generated. The body of the rule consists of the translation of the calculus formula on the right-hand side.

5.1.4 Context-sensitive Conditions

Besides the syntactical correctness of a data specification there are some more conditions that have to be satisfied:

- The names of sorts, selectors (of a specific tuple sort), operations, and predicates resp. have to be unique.
- Any derived sort has to be specified under **sorts**.
- Cyclic sort definitions are not permitted, the sorts can only be constructed hierarchically.
- The argument (and destination) sorts of operations and predicates have to be defined.
- Only one equation is a admitted for a single operation.
- The parameter names of operation and predicate definitions have to be unique.
- The terms (formulas) on the right-hand sides must not contain any free variable other than the parameter variables, and every parameter variable has to appear in the terms (formulas).
- No recursion is allowed for the definitions of operations or predicates; only operations or predicats defined before the current one may be used on the right-hand side.

If any of these conditions is violated, a warning is given, that the data specification is incorrect. Nevertheless, the translation is performed to facilitate debugging by comparing the translated to the source code.

5.2 Translation of EER Schemas

5.2.1 Syntax of Schema Definitions

Like data specifications, EER schemas in QUEER are defined textually. A sample definition is outlined in figure 2. Entity types are defined by their attribute and component names and their domains, relationship types by the names and domains of their participants and data-valued attributes, respectively. To define a construction type, the types of the input and output entities must be given.

An attribute or component in an entity-type definition may be preceded by the reserved word `key:`, indicating that this attribute is an element of the set of key attributes of that entity type. The set of key attributes must not be empty for basic, i.e. non constructed, entity type. Every data type which is defined in the corresponding data definition may serve as an attribute domain. In addition non-atomic attributes may be specified by means of the constructors *set, bag* and *list*, which may not be nested.

```
schema country_town_waters

entity types
        country attributes      key:    cname           : string,
                                        area            : regions,
                                        cpopulation     : int,
                                        government      : forms;
                components              boss            : person,
                                        ministers       : list(person);
        town    attributes      key:    tname           : string,
                                        tgeo            : circle,
                                        tpopulation     : int;
        river   attributes      key:    rname           : string,
                                        rgeo            : lines;
        lake    attributes      key:    lname           : string,
                                        lgeo            : circle;
        sea     attributes      key:    sname           : string,
                                        sgeo            : polygon;
        waters
        person  attributes      key:    pname           : string,
                                key:    addr            : list(address),
                                        age             : int;

relationship types
        flows_through   participants    river, country;
                        attributes      length          : real;
        is_mayor_of     participants    person, town;

construction types
        are     input           sea, lake, river;
                output          waters;


end_schema country_town_waters.
```

Figure 2: Sample schema definition

5.2.2 The Meta Database

In some respects the translation of EER data into PROLOG clauses resembles that into relational schemas, but due to the greater expressiveness of horn clauses with function symbols it is somehow more straight forward. The translation into relational schemas has well been investigated [Hoh88]. There, every entity type comprising multivalued attributes or components has to be translated into multiple tuples with atomic attributes[3]. In our approach it can be represented in a single PROLOG term, using complex terms with function symbols for the representation of non-atomic attributes. The concept of handling components, relationships and constructions by means of surrogates is analogous to [Hoh88].

Basically three types of PROLOG clauses are generated for every schema object[4], which form QUEER's meta-database:

I Type declarations

II Selection and conversion rules

III Operator and predicate declarations

Group I predicates are used by the query compiler and the *dml* interpreter to determine the schema properties of a database object, i.e. its attribute names, positions and domains (components and participants resp.). Group II predicates are used in the evaluation of queries to perform the selection of attributes, components or participants or to execute a conversion between input and output entities in type constructions. Group III

[3] as far as no NF^2 model or the like is used

[4] a schema object is an entity type, a relationship type or a construction type

predicates are generated for technical reasons only. They are required by the other compilers for type checking, when applying group II rules.

Meta-Data for Entity Types: I. Every entity type is described by a single clause, which takes the form

```
entity_type(Entityname, Attributes, Components) .
```

where `Entityname` is instantiated with an atom, the name of the entity type, and `Attributes` and `Components` are PROLOG lists. `Attributes` contains for every attribute position one tuple `(Attrname, Attrtype)`, where `Attrname` is the name of that attribute and `Attrtype` its domain. This may be a defined sortname or of the form `set(S)`, `bag(S)`, `list(S)`, where `S` is a sortname. Correspondingly, `Components` is a list of tuples `(Compname, Comptype)`, which consist of the component's name and its domain, i.e. the name of a defined entity type or, again, `set(E)`, `bag(E)`, `list(E)`, where `E` is an entity type. The instance of this predicate for the type *country* from fig. 2 is:

```
entity_type( country,
             [ ( cname, string ), ( area, regions ),
               ( cpopulation, int), ( government, forms )
             ],
             [ ( boss, person ), ( ministers, list(person) ) ]
           ) .
```

II. For every attribute and every component a rule is generated that returns the value of that attribute when called with some surrogate of an entity for which this attribute or component has been defined. The general form of these rules is:

attr(`S, A`) `:- entity( Entityname ,[ S,` V_1`, ...,` V_n `], _ ).`

and

comp(`S, A`) `:- entity( Entityname , [ S | _ ], [` V_1`, ...,` V_m `], _ ).`

where *attr* and *comp* are replaced by the attribute's or component's name and `Entityname` by the entity's type name. The V_i are anonymous variables in every position except for the one corresponding to the attribute handled by this rule. These predicates are unit clauses and instances of the predicate `entity/3`, which is used for storing the entities in the database. Therefore selection is done without resolution steps, only by unification. Examples of these clauses are:

```
cpopulation(S, A) :- entity( country, [ S, _, _, A, _ ], _ ) .
```

and

```
ministers(S, A) :- entity( country, [ S | _ ], [ _, A ] ) .
```

III. For every predicate of group II (selection and conversion rules) an operator declaration is generated that declares its main functor as a normal calculus function. These predicates are instances of

```
db_opn(Attrname, [Entityname], Domain).
```

This is used to express that `Attrname` is a function from `Entityname` into `Domain`. `Attrname` is an attribute name, i.e. a functor of a group II predicate, `Entityname` is the name of the entity for which this attribute is defined, and `Domain` is its domain. By this "trick" it is possible to handle selections on the syntactical level just like any data valued function.

Meta-Data for Relationships: The predicates resulting from the translation of relationship types have almost the same appearence as those for entity types.

I. The type declaration of a relationship is an instance of

```
relship_type(Relshipname, Participants, Attributes) .
```

Again, `Relshipname` is an atom, the name of the relationship type, and `Participants` and `Attributes` are PROLOG lists. The elements of `Participants` are tuples `(Rolename, Parttype)`, consisting of a rolename and the entity type for each role. `Attributes` is the list of the relationship's data valued attributes and has the same structure as the corresponding attribute list for entity types. As an example the type definition for *flows_through* is given:

```
relship_type( flows_through,
              [ ( river, river ), ( country, country ) ],
              [ ( len, real ) ]
            ) .
```

When no rolename is specified, the entity type name is taken as default.

II. Consequently, there must be selection predicates for every attribute and participant position of a relationship, too. Their structure is:

rolename((Relshipname, [S_1, ..., S_n]), S_l) :-
        rel_ship(Relshipname, [S_1, ..., S_n], _) .

and

attr((Relshipname, Surr_list), V_l) :-
        rel_ship(Relshipname, Surr_list, [V_1, ..., V_m]) .

where `Relshipname` is replaced with the relationship type's name and *rolename* (*attr*) by a rolename (attribute name respectively). In the definition of *flows_through* some of their instances are:

```
country( (flows_through, [_1, _2]), _2) :-
   relship( flows_through, [_1, _2], _ ) .

len( (flows_through, Surr_list), _1 ) :-
   relship( flows_through, Surr_list, [_1] ) .
```

III. At last, the operator declarations for group II rules are generated. They take the same form as those described for entities, e.g.

```
db_opn( country, [ flows_through ], country ) .

db_opn( len, [ flows_through ], real ) .
```

Because relationships are treated like binary predicates, there must be appropriate declarations for each relationship type, like those for normal predicates defined on data values. One example is:

```
db_pred( flows_through, [ river, country ] ) .
```

Meta-Data for Construction Types: I. There is a schema information predicate for construction types, which gives the defined input and output entity types.

```
cons_type(Consname, InList, OutList) .
```

in which `Consname` is the name of the construction type and `InList` and `OutList` are lists containing the names of the input and output entity types belonging to it.

II. Furthermore, there are the type conversion predicates

entity(C, B) :-
        construction(Consname, B, C),
        entity(*entity*, [B | _], _) .

that transform an input entity to the output entity, for which it stands, and vice versa (with `C` and `B` interchanged).

III. Again, the group II rules are declared as operators by `db_opn/3`.

The predicates shown below are produced from the definition of the construction type *are*:

```
cons_type( are, [sea, lake, river], [waters] ) .

waters( B, C ) :-
   construction( are, B, C ),
   entity( waters, [C | _], _ ) .

lake( C, B ) :-
   construction( are, B, C ),
   entity( lake, [C | _], _ ) .

db_opn( waters, [lake], waters ) .

db_opn( lake, [waters], lake ) .
```

etc.

5.2.3 Keys

In addition to the predicates described above clauses are generated that describe the set of key attributes for each entity type. Their clauses are used by the *dml* interpreter to verify transactions against the implicit integrity constraints of the EER model. The clauses are also used to distinguish by which attributes entities have to be specified in manipulation commands, for an entity is referenced by the values of its key attributes. For this and some other reasons, the key attributes have to form a strict hierarchy, or, in other words, cycles are not allowed in the key signature.

5.2.4 Context-sensitive Tests

Together with the translation of a schema definition some tests are carried out to verify that the given schema is valid, i.e. it matches the inherent schema constraints of the EER model. The conditions which have to be obeyed are:

- Only sorts defined in the current data specification module may be used.
- Every name of a schema object must be unique, i.e. the sets *ENTTYPE*, *CONSTRUCTION* and *RELTYPE* must be disjoint.
- Every name used in the definition of a schema object must be unique within the scope of this definition.
- An entity type must not appear more than once as an input type of the same construction. The same must be true for output types.
- An entity type must not be constructed by more than one type construction.
- Any base entity type must have a non-empty set of key attributes, the set of key attributes for constructed entity types must be empty.
- The key signature must not contain cycles.

A schema definition that is syntactically correct and matches the conditions listed above is permissible and can be translated.

5.3 Queries

5.3.1 Introductory Example

For the sake of clarity we will first give a small example to show the necessary steps for the translation of queries. Note, that the syntax of queries in *calc* is slightly different from the original calculus: every variable name have a capital first letter, all other identifier has to start with a small letter. The correspondence between the operators, connectors etc. should be clear.

Consider the query

```
-[ pname(P) | (P: person) & age(P) < 18 ]-
```

which returns the names of all persons under age 18. The translated code of this query is

```
bagof( PName,
       ( entity(person, [P|_], _),
         age(P, Age),
         Age .<. 18,
         pname(P, PName)
       ),
       Result
     ).
```

The bag-valued expression of the calculus is translated to an application of the PROLOG predicate `bagof/3`. Thus the solution of this query, contained in the variable `Result`, will be the list of all instances of `PName` for which the second argument of `bagof` can be satisfied, i.e. there must be an entity of type `person` whose age is less than 18 and has the name `PName`. On backtracking the variable `P` will be bound to the surrogate of each such entity. The declaration in the calculus expression is translated to the term `entity(person, [P|_], _)` that unifies with facts of the form

```
entity(Name, [Surrogate | Attributes], Components)
```

which is the representation of entities in the database. The “operations” `addr/2` and `pname/2` generated in the translation of the schema definition return the address resp. the name of the entity with this surrogate `P`.

Notice that every calculus variable appears in the PROLOG goal. But several other variables are generated to keep the result of an “operation” since n-ary functional terms have to be translated to $(n+1)$-ary predicates. The parameter at position $n+1$ is the result variable which is necessary because there are no functional concepts in PROLOG.

5.3.2 Functions in Prolog

Except for arithmetic predicates, which have to be evaluated by `is/2`, PROLOG offers no functional concepts. Because of this lack, the functions, operations, and selections of the calculus have to be treated as relations. In general, logic programs representing n-ary functions, are defined by $(n + 1)$-ary relations. The solution of a query with n arguments instantiated and uninstantiated last argument will return the function value as a substitution for the last argument.

Nested applications of functions must be expressed by sequences of subgoals. In these sequences the subgoals to the left of a specific subgoal yield the values of the evaluated “inner” arguments as instantiations of the argument variables of the current subgoal.

The advantage of this relational representation is the possiblity of instantiating other arguments than those mentioned before. Thus a predicate `plus/3` can be used for addition and subtraction of numbers as well. A query with `plus` having all arguments bound is a test whether the sum of the first two arguments is equal to the last one. However the different applications of the same predicate are to be treated very carefully because they are only possible if the implementation is sound and complete.

In the translation of the calculus to PROLOG there are only two cases of interest:

- a call with uninstantiated last argument (all others instantiated), or
- a call with all arguments being instantiated.

For these cases even incomplete implementations will suffice. These may in general be more efficient because tests with `var/2` or the like can be dropped.

5.3.3 Surrogates

The object identity of the entities is ensured by associating a unique surrogate with each entity on insertion. References to entities participating in a relationship can be made by specifying their key attributes when such a relationship is inserted. The PROLOG fact for this relationship contains a list of the surrogates of these entities. Roughly speaking, declared variables of a calculus expression are bound to the surrogates resp. the list of surrogates during the evaluation of its PROLOG counterpart.

5.3.4 Generate-and-test Strategy

Ideally, in a logic programming language the ordering of subgoals in a query should not have any effect on the solutions because of the commutativity of conjunctions. For this reason the calculus subexpressions could be translated in the order they appear in a naive approach. Another advantage is the possibility of subgoal reordering to achieve an optimized query.

Unfortunately these assumptions cannot be made for PROLOG. The operational view of programs has to be taken into account. One reason for this is that the aforementioned arithmetic predicates require instantiated arguments. Thus the subgoals evaluating the arguments must not be placed to the right of these arithmetic subgoals. Another, more serious reason is the neccessity of using the cut ("!"). In general, its use in the standard definition of negation by failure has consequences on the translation order because `not`, called with a non-ground argument, is incomplete in conjunction with other subgoals.

But there is a way out of this dilemma: `not` works correctly with ground arguments. We can make use of this property by placing the `not` subgoal at the end where all variables are bound. To see why they are bound consider the declarations: their PROLOG counterparts will be placed to the left of a subgoal sequence because they bind the free variables to surrogates, surrogate lists, or data values. Operator applications on these arguments will instantiate the result variables. This "generate-and-test" strategy is actually implemented in this form. Thus every use of `not` in our translation has all arguments bound.

5.3.5 Handling of Set and Bag Expressions

One of the most important constructs of the calculus is the bag-valued term

$$-[t_1, \ldots, t_n \mid \delta_1, \ldots, \delta_m \ \wedge \ \varphi]-$$

which retrieves *all* instances of $t_1, \ldots, t_n$ implied by $\delta_1, \ldots, \delta_m \ \wedge \ \varphi$. In PROLOG however, the solution to a query consists of *one* instance which is an implication of the program.

A query of this kind is a second-order query since it asks for a bag of elements with a certain property. It would be impossible to formulate it in pure PROLOG, i.e. without `assert`, `retract`, and the cut. These are the language elements used in the implementation of set predicates like `bagof/3` or `setof/3` which are based on side effects: every solution found on backtracking is stored in the database as a temporary fact. These facts are gathered in a list which is returned as the solution. Unfortunately, these predicates are not standardized. In some cases the returned list is sorted, in others it contains the elements in the order they were found. `bagof` can fail if no solutions are found or it can suceed with the empty list.

In QUEER we need the following semantics: `bagof(Term, Goal, Instances)` never fails (empty list if necessary), it sorts the elements of `Instances`, and considers free variables in `Goal` which do not appear in `Term` to be existentially quantified.

The implementations of the set predicates causes another problem because they are based on failure driven loops. This has effects on the generation of duplicate solutions. The question is for example, whether the terms $-[\ldots \mid \ldots, \delta \ \wedge \ \varphi]-$ and $-[\ldots \mid \ldots \ \wedge \ \exists \ \delta\varphi]-$ have to be translated differently. The variables in the body of a PROLOG clause can be considered to be existentially quantified. So, at a first glance, the translations might be the same. But this is not the case if declarations appear — as in the example — inside of bag expressions. The second term requires only *one* instance of the quantified variable, whereas the first should generate a duplicate of the solution term for every instance of the declared variable that satisfies the formula.

As a consequence, the quantified formula has to be encapsulated in a predicate that prevents the generation of more than one solution on backtracking. This is achieved by

```
exists(X)  :-  X, !.
```

The same problem arises with the logical *or*. A simple translation with the PROLOG "or"(;) would cause an unwanted re-satisfyabilty. Hence, the logical *or* has to be implemented in a way it is satisfiable only once by

```
X or Y  :-  (X ; Y), !.
```

5.3.6 Translation of the Language Constructs

In this section we present the PROLOG counterparts of the language elements of the calculus. In table 1 they are grouped according to the calculus definition in terms, formulas, declarations, and ranges. For every calculus construct the corresponding PROLOG code and the binding of the result variable (with the exeption of formulas) is given.

For notational convenience, we introduce a few abbreviations as follows. For example, if the calculus expression is $t_1\ \omega\ t_2$, the corresponding PROLOG code for t_1 (resp. t_2) is T_1 (resp. T_2). The result variable of T_i (see section 5.3.2) is V_i. Ω is the PROLOG function corresponding to ω.

Table 1 is explained in the following:

Terms: For constants and variables a `true` as code has to be generated which is removed later in the optimization phase. The return variable is bound to the value of the constant or is unified with a PROLOG variable (This is the same for variables with equal names.).

The translation of different kinds of operations has already been discussed in the section on functions in PROLOG (section 5.3.2), e.g. the term

$$pdist(center(tgeo(t)), center(lgeo(l))),$$

which computes the distance between the centers of a town t and a lake l, is translated to

```
tgeo(T, _0), center(_0, _1),
lgeo(L, _2), center(_2, _3),
pdist(_1, _3, Res).
```

If a functional PROLOG operator ω exists for the operation to be translated the generated code contains the predicate `:=/2` which behaves similar to the PROLOG predicate `is/2` but returns `bottom` if one of the operands is non-numeric or a division by zero appears.

There is no need for a different handling of the different kinds of operations, as there are:

1. selection of data-valued attributes,
2. selection of object-valued attributes (components),
3. selection of entities participating in a relationship,
4. selection of multi-valued attributes
5. type-conversion functions $s_{in}(t_{out})$ or $s_{out}(t_{in})$,
6. pre-defined and user-defined operations (including aggregations).

The bag expressions are translated to a `bagof` goal that was discussed in depth in section 5.3.5. The free variables have to be bound in the declarations D_i in the first place and are afterwards tested for the satisfaction of the qualifying formula F. At last, the values of the result terms are computed. The tuples of these terms are gathered in V.

Please note the different types of commas in the table: one is the argument separating comma in the parameter list of a predicate or function, the other is the PROLOG *and*.

Formulas: For the same reasons as in the case of operations the atomic formulas can be treated uniformly as the necessary predicates are pre-defined or generated by the translation of the schema definition. The non-atomic formulas can be translated into their PROLOG equivalents[5]. Implications, equivalences and universally quantified formulas are transformed to their logical equivalents based only on *and*, *or*, *not*, and *exists*.

Please note every comma in the `exists` and `not` clauses is a PROLOG *and*. `exists` and `not` are declared to be monadic operators in PROLOG.

[5] `exists/1` and `or/1` are described in section 5.3.5

calculus expression		prolog code	return variable
term	t	T	V
formula	φ	F	-
declaration	δ	D	V
range	ρ	R	V

Terms

constant	c	*true*	C
variable	v	*true*	V
operation	$t_1\ \omega\ t_2$	T_1, T_2, V := $V_1\ \Omega\ V_2$	V
operation	$t_1\ \omega\ t_2$	T_1, T_2, Ω(V_1, V_2, V)	V
operation	$\omega(t_1, \ldots, t_n)$	T_1, ..., T_n, Ω(V_1, ..., V_n, V)	V
bag	$-[t_1, \ldots, t_n \mid \delta_1, \ldots, \delta_m \wedge \varphi]-$	`bagof`((V_1, ..., V_n), (D_1, ..., D_m, F, T_1, ..., T_n), V)	V

Formulas

predicate	$\pi(t_1, \ldots, t_n)$	T_1, ..., T_n, π(V_1,...,V_n)	
predicate	$UNDEF(t)$	T, `undef`(V)	
predicate	$t_1\ is\ t_2$	T_1, T_2, `equal`(V_1,V_2)	
negation	$\neg\varphi$	`not` F	
conjunction	$\varphi_1 \wedge \varphi_2$	F_1, F_2	
disjunction	$\varphi_1 \vee \varphi_2$	F_1 `or` F_2	
implication	$\varphi_1 \implies \varphi_2$	(`not` F_1) `or` F_2	
equivalence	$\varphi_1 \iff \varphi_2$	(F_1, F_2) `or` (`not` F_1, `not` F_2)	
existential quantifier	$\exists\delta : \varphi$	*exists (D, F)*	
universal quantifier	$\forall\delta : \varphi$	*not exists (D, not F)*	

Declarations

$(v : \rho_1 \vee \ldots \vee v : \rho_n)$	R_1; ...; R_n	$V_1 = \cdots = V_n$
$(v : \rho_1 \vee \ldots \vee v : \rho_n); \delta$	D, (R_1; ...; R_n)	$V_1 = \cdots = V_n$

Ranges

entity type	e	*entity(e, [S | A], C)*	S
relationship type	r	*relship(r, P, A)*	(r, P)
construction type	c	*construction(c, I, O)*	(I, O)
set-valued term	t	*T, member(E, V)*	E

Table 1: Translation of calculus constructs to PROLOG

Declarations: In contrast to the disjunctions in a formula the disjunctive declarations are translated with the standard Prolog disjunction operator "`;`" since all instances have to be considered. The return variables of the ranges are unified with each other as they all refer to the same declared variable.

Ranges: For these language constructs the translation generates subgoals corresponding to their definition in the schema if the ranges are entity, relationship, or construction types. For entities the return variable is bound to the surrogate, for relationships it is bound to a tuple consisting of the relationship name and a list of the surrogates of the participating entities. The relationship name is necessary since the same tuples can be involved in several relationships. The code for the construction type ranges is generated analogously. For set-valued ranges the Prolog predicate `member` binds an element of the result list of the evaluated set term to the declared variable.

5.3.7 Comparison to the Translation into the Relational Calculus

[Hoh88] describes a translation of schemas from the EER model to the relational model. Queries expressed in the EER calculus are translated to tuple calculus. This causes difficulties at several points if the complete EER calculus is admitted as the query language:

- Only the operators and predicates defined in the specific relational language can be used. Therefore only the operations and predicates based on them are admitted in the EER query language.
- Queries resp. the underlying schema are supposed to contain arbitrary object-valued attributes which in addition can be multi-valued. If a multi-valued attribute is referenced in a query a join of the "main" relation with the "attribute" relation has to be performed since only atomic data types are allowed in the relational model.
- Lists (sequences of objects) cause a further problem, since an attribute specifying the position of an object in the original list is needed.

These problems do not appear in the case of translating EER calculus expressions to Prolog because multi-valued attributes can be handled more directly as Prolog lists etc. This avoids the generation of tables and tuples caused by the additional join and position attributes. Most of the language elements of the EER calculus can readily be translated to Prolog. Especially the formulas have direct equivalents in both languages.

5.3.8 Optimization

The optimization phase consists of two essential parts, the simplification of arithmetic or boolean expressions and the unfolding of certain subgoals.

Simplification: The translation of constants and variables (see section 5.3.6) generates a `true` for every occurence of such a term, e.g. `true, true, T1 := 3*4, true, T2 := T1 + A` for the calculus term $3*4+A$. These instances of `true` can be eliminated in conjunctions. After having done these first steps sequences of arithmetic evaluations can be joined to a single one as long as the operators involved have a functional correspondent in Prolog, e.g. the sequence `T1 := A-D, T2 := 2*T1, T3 := sqrt(T2)` can be simplified to `T3 := sqrt((A-D)*2)`.

Unfolding: In general, in pure Prolog the following statement is valid:

If a predicate p *has only one clause in the database, every call to* p *in a goal can be substitued by the body of this clause (with the subgoal term and the rule head being unified with each other).*

Thus a lot of inference steps can be performed during compile time. This strategy is known as *unfolding*. The restriction *in pure* Prolog is essential for the statement. It is not generalizable to full Prolog. This is easily shown by the following example: if the call `exists(Goal)` were unfolded to `Goal, !` the cut would appear in a level where it prunes the search tree at a parent node of the intended one. Nevertheless, this optimization strategy should be applied whenever possible. This is achieved by specifying a table with those predicates which must not be unfolded.

The translation of a selection on entity attributes or components is a unit clause that performs just a unification of the result variable with the corresponding value in the entity term, e.g.

```
tgeo(T, Geo) :- entity(town, [T, _, _, Geo], _).
```

The rule body unifies with the subgoal generated by a declaration translation like

```
entity(town, [T | A], C)
```

This suggests not only to unfold the selection subgoals but to unify them with the declaration subgoals. Thus a complex query like

```
entity(town, [T | Attrs], Comps),
tpopulation(T, Pop),
Pop .=. 100000,
tgeo(T, Geo),
radius(Geo, Radius),
Radius .<. 5000
```

can be simplified to

```
entity(town, [T, _, 100000, (Radius, _)], _),
Radius .<. 5000
```

This strategy cannot be pursued in every case. If the same selection appears in both operands of an *or* the first one could be unified but it would fail with the second one. E.g. the translation of *pname(person) = 'Smith'* $\vee$ *pname(person) = 'Jones'* is

```
pname(Person, 'Smith') or pname(Person, 'Jones').
```

Here the unification succeds with the first operand:

```
entity(person, [P, 'Smith', _, _], _).
```

But then 'Jones' had to be unified with 'Smith'...

A similar problem appears inside a negation. A unification of a selection with a declaration outside the `not` would be semantically wrong.

Unification instead of explicit equality: The equality in QUEER is the same as unifyability in PROLOG. Thus, instead of generating subgoals like `cname(C, N), N = 'USA'`, it is possible to perform the unification at compile time: `cname(C, 'USA')`. This optimization cannot be made inside of a `not` if the variable to be unified is declared outside of it.

Further possibilities: Another possible optimization strategy is subgoal reordering. In our case it resembles the algebraic optimization known from the relational model with rules like "perform selections as early as possible". An example of a bad subgoal order is

```
entity(river, [R | _], _),
entity(town, [T, _, Pop, _], _),
relship(lies_at, [T, R], _),
Pop .>. 100000
```

since at first each town is tested whether it lies at a river, although this would be necessary only for those with a population of more than 100000. A better order is

```
entity(river, [R | _], _),
entity(town, [T, _, Pop, _], _),
Pop .>. 100000,
relship(lies_at, [T, R], _)
```

A heuristic would have to be developed for another conceivable strategy which would reorder the subgoals by placing the stronger specified ones (e.g. with fewer free variables) to the left.

We have not studied these ideas in depth, but this should be further investigated because they may lead to considerable improvements.

5.3.9 A Few Examples

Here, we present a few larger examples giving the results of test runs. These examples (taken from [HG88, GH91]) are based on the running example of the geographic schema and have the following structure:

- a verbal formulation of the query,
- the query formulated in the calculus,
- the result of the translation before the optimization phase, and
- the optimized PROLOG code.

The resulting code can be called with `query(Solution)` from the PROLOG interpreter's interface. For better readability we changed the compiler's output slightly by giving the relevant variables mnemonic names or the corresponding names given in the original query, resp.

Example 1: The following query retrieves the names of ministers having only addresses in towns of their country.

```
-[ pname(P)
        | (P : lts(ministers(C))) ; (C : country)
        & (forall (A : lts(addr(P)))) (exists (T : town))
                (lies_in(T, C)  &  tname(T) = city(A)) ]-

query(Solutions) :-
    bagof(PName,
          (entity(country, [C | _], _),
           ministers(C, Min),
           lts(Min, MinSet),
           member(P, MinSet),
           not exists (addr(P, Addr),
                       lts(Addr, AddrSet),
                       member(A, AddrSet),
                       not exists (entity(town, [T | _], _),
                                   lies_in(T, C),
                                   tname(T, City),
                                   city(A, City)
                                  )
                      ),
           pname(P, PName)
          ),
          Solution
         ).
```

The operation `ministers` retrieves the list of all ministers of a certain country `C`. The list is converted by `lts` to a set[6] `MinSet`. `member` yields on backtracking every single minister `P` out of this set. Similarly the single addresses of each minister are retrieved in the following `not exists` subgoal which is a result of the translation of the `forall` formula (transformed to $\neg\exists\neg\varphi$). In the second `not exists` it is verified that there is no town `T` with name `City` which is the same as the `city` component of one of the minister's address tuples. At last, the minister's name is retrieved by `pname`. All the names are collected in `Solutions`.

```
query(Solutions) :-
    findall(PName,
            (entity(country, [C | _], [_, Min]),
             lts(Min, MinSet),
             member(P, MinSet),
             not exists (entity(person, [P, _, Addr, _], _),
                         lts(Addr, AddrSet),
```

[6] recall that only set-valued ranges are allowed in the calculus

```
                    member((_, City, _, _), AddrSet),
                    not exists (entity(town, [T, City, _, _], _),
                                relship(lies_in, [T, C], [])
                               )
                   ),
            entity(person, [P, PName, _, _], _)
           ),
           Solutions
          ).
```

The attribute or component selections like `ministers`, `addr`, `tname`, and `city` have been replaced by simple unifications. The implicit declaration of P (Type `person`) has been made explicit because an attribute `pname` is selected of these entities. `lies_in` is unfolded to its definition.

Example 2: The following query computes the average of the maximal populations of countries with a certain form of government over all these forms.

Let us shortly explain the situation: Assume we have three countries. The democratic country c_1 has 10,000,000 inhabitants, the democratic country c_2 has 20,000,000 inhabitants, and the socialistic country c_3 has 35,000,000 inhabitants. The query computes for every form of government $F \in \{democratic, socialistic\}$ the maximal population of countries having this form, i.e. 20,000,000 for $F = democratic$ and 35,000,000 for $F = socialistic$. The average of both values is 27,000,000.

```
avg(-{ max (-{ cpopulation(C)  |  (C : country)  &  government(C) = F }-)
                 | (F : bts(-{ government(C)  |  (C : country) }-))        }-)

query(Solution) :-
    bagof(Max,
          (bagof(Gov, (entity(country, [C |_], _), government(C, Gov)), GovBag),
           bts(GovBag, GovSet),
           member(F, GovSet),
           bagof(CPop,
                 (entity(country, [C1 | _], _),
                  government(C1, F),
                  cpopulation(C1, CPop)
                 ),
                 CPopBag
                ),
           max(CPopBag, Max)
          ),
          MaxBag
         ),
    avg(MaxBag, Solution).
```

Similarly to the first example the different forms of government `F` the countries `C` in the current database have by first retrieving the bag of all government forms and then converting it by `bts` to a set. In the second inner `bagof` the population numbers of the countries `C1`[7] having such a form of government `F` are collected in `CPopBag`. The maximum is computed of each such bag, and the solution to the query consists of the average `avg` of the bag of each computed maximum.

```
query(Solution) :-
    findall(Max,
            (setof(Gov, entity(country, [_, _, _, _, Gov], _), GovSet),
             member(F, GovSet),
             findall(CPop, entity(country, [_, _, _, CPop, F], _), CPopBag),
             max(CPopBag, Max)
            ),
```

[7] The variable `C` has already been used in the inner bag expression. In the calculus we have lexical scoping, so the declared `Cs` are different variables.

```
        MaxBag
      ),
  avg(MaxBag, Solution).
```

The first inner *bagof* has been combined with the following *bts* to *setof*. The second *bagof* was replaced by the simpler *findall* because there is no need to order the elements for the *max* operation. The selections of the attributes *government* and *cpopulation* have been replaced by unifications.

6 Data Manipulation

QUEER provides a separate data manipulation language, which is based on the EER calculus. Due to this fact we obtain an orthogonal concept for this language. For several reasons we do believe that the usual front-end of a PROLOG system would be a poor user interface for data manipulation purposes.

- Complex terms used for the representation of database objects are hard to read and to construct.
- Changes made to the database would not be transparent to the system. Thus integrity monitoring would be a very expensive task.
- Syntactically incorrect data structures or errorneous usage of correct data structures would be hard to detect as well.
- The user would have to know about the meaning and usage of internal information like surrogates, which should be hidden on the front-end.

Especially the problem of dependencies between entity types — which we could term the "referential integrity" of the EER model — makes clear that there must be a possibility for the system to verify the result of a manipulation, in order to avoid inconsistent database states. This problem, which does not appear in the manipulation of pure relational data, arises from the fact that every entity, which is used as a component, a participant or as the input or output of a construction, has to be defined as an entity in its own right first, i.e. the instruction creating this entity has to precede the one using it. This problem is resolved by the transaction concept described later.

6.1 Syntax

There are three primitives for altering the database state: **insert**, **delete** and **update**. These commands may be applied to any kind of database object.

Insertions: The insert-command has two different appearances: value-insert and query-insert.

Value-insert: Here the objects to be inserted are specified by the values of their key attributes[8]. The values for non-key attributes may be specified or set to *bottom*.

```
insert into town values: ('Braunschweig',
                          tuple(tuple((22.1,35.5), 12.4),
                          315000
                         );
```

inserts a new entity of the type town. Several objects of the same type may be inserted by a single instruction, like in:

```
insert into flows_through values:
      (<'Rhein'>,  <'FRG'>,  800);
      (<'Main'>,   <'FRG'>,  300);
      (<'Isar'>,   <'FRG'>,  150);
      (<'Elbe'>,   <'FRG'>,  400);
```

[8]key components are specified by a list of their key attribute values enclosed in pointed brackets like in the example in section 6.3

```
(<'Elbe'>,    <'GDR'>,   600);
(<'Elbe'>,    <'USSR'>,  600);
(<'Wolga'>,   <'USSR'>, 1200);
```

This adds seven new relationships of the type *flows_through* between rivers and countries.

Query-insert: In this form the objects to be inserted are composed from values which are the result of a database query, e.g.

```
insert into lies_in:
      -[ T, C | (T:town), (P:lts(ministers(C))) ; (C:country)
                & T.tname = city(sel(P.addr,1))
      ]-
```

states that every city, in which some minister has her primary residence, lies in the country governed by her. Of course, if the result of `city(sel(P.addr,1))` does not match the name of any city entity no relationship is created.

For data manipulation purposes the syntax of queries has been extended so that it is possible to have object-valued terms in the result list of a bag. Thus there is a convenient way to address objects that are to be altered. This possibility has already been used in the last example.

Deletions: By means of these object-valued terms deletions are made very easy. The delete-command simply consists of a query with a result sort taken from the set of schema objects, preceded by the keyword **delete**. Such a command removes the entire set of objects the query yields. The following statement deletes every town with less than 100000 citizens:

```
delete -[ T | (T:town) & T.tpopulation < 100000 ]-
```

More than one object may be deleted in a single command, like in:

```
delete -[ P, C | (C:country), (P:person) &
                 C.cpopulation < 1000000 & P=C.boss ]-
```

This delete command removes every country with less than a million inhabitants and their heads of government simultaneously. If the `P.boss` is undefined, the corresponding country is not deleted. This seems to be contrary to our intuiton, but it is absolutely consistent with the semantics of the calculus. No tuple `(P,C)` can be generated for this country, because `P=C.boss` cannot be fulfilled. Hence, this country is not in the result of the query coressponding to this delete command.

Futhermore, in manipulation commands, construction types may be used as ranges as well. This is the only possibility to delete a construction between two entities. E.g. the instruction

```
delete -[ A | (A:are) & water(A).wname='Rhein' ]-
```

deletes the construction between the river Rhein and the corresponding water. The entity `water(A)` will not be removed by this deletion command, only the construction will be deleted. This is not perfectly consistent with the semantics of constructions, for constructed entities can only exist by means of construction from their base entities. One ad-hoc approach to overcome this problem would be to introduce additional integrity tests on constructed entity types and to reject such a deletion. a better way would be to delete constructed entites simulaneously with their constructions, but this would mean to define a different semantics for calculus terms when used in manipulation commands.

Updates: Update commands look very much like query-inserts with the difference that the list of result terms is substituted by a list of assignments, like in:

```
update  -[ C.cpopulation := 215 + C.cpopulation,
           | (C : country) & C.cname = 'BRD'
        ]-
```

The assigned values may be given by constants or terms that can be computed analogously to the results of the bag term. Of course, the sorts to both sides of the assignment operator must match. As as example the statement

```
update  -[ C.cname := 'FRG', C.boss := P |
           (C : country), (P : person) &
           P.name = 'Kohl' & C.cname = 'BRD'
        ]-
```

assigns to the country named 'BRD' — or perhaps the countries, if there is more than one — the new name 'FRG' and establishes the person with the name 'Kohl' as its head of government. Note, key attributes may be changed like any other attribute, if key integrity is maintained. We consider key attributes to be only a technical requirement; object identity is not given by values of key attributes, rather than by the existence of an object itself. thus, there is no reason to forbid updates of keys.

The database state resulting from an update-command must be unambiguous. For this example this means that there must not exist two different persons with the name 'Kohl' in the database at the time this instruction is executed. Attempts to execute ambiguous update commands will be detected in the course of their evaluation and are rejected.

6.2 Integrity Monitoring

Although there is no possibility for the user to specify explicit integrity constraints, some kind of integrity monitoring is indispensable, because *dml* commands have to be verified for the satisfaction of the inherent integrity constraints of the EER model to avoid inconsistent database states. Of course, the EER calculus could also be employed to express user defined integrity constraints. Nevertheless, on the implementational side this would require a lot of effort to build an "intelligent" integrity monitor, if it should be possible to handle more than a tiny number of constraints. Because our primary goal was to show that logic programming is an appropriate framework for the prototyping of database languages based on semantic data models, we did not treat this aspect of database systems.

The model inherent constraints against which every manipulation is verified are:

- **Schema compatibility**: every object produced by a manipulation must match the schema defined for its type.
- **Key integrity**: there may not be two entities of a type in the database that share the same values in every key attribute position.
- **Cyclic references**: No entity may be constructed from itself, neither directly nor transitively.
- **Undefined references**: No entity that is not already stored in the database may be used as a component, a participant or an input or output entity. This situation can arise from two different patterns of operation:
  1. An object valued constant is specified by a set of key attributes that does not match any existent entity of the type required.
  2. An entity is removed from the database that is referenced by another object without removing this object. In this case the objects containing dangling references are determined and reported to the user. Then she is interactively given the possibility to delete these objects as well. Otherwise the incorrect references are set to the undefined value *bottom* ($\perp$). If the dangling references appear in the position of key attributes the entity containing the must of course be deleted.

By means of a cross-refernce table, which is permanently maintained, the tests are organized in a way such that only the relevant portions of the database are tested.

Every manipulation that fails to match the conditions mentioned above is rejected.

6.3 Transactions

dml instructions may be chained by the keyword "`---`". In this case they are executed as a sequence of individual instructions with the integrity tests being executed upon completion of every single instruction. There is a second way to build complex instructions: several commands, separated by the keyword "`||`", are executed as a single transaction, i.e. there may be inconsistent database states between the execution of their subinstructions, as long as these are resolved by the remainder of the transaction. Such a transaction can only be executed or rejected in its entirety, i.e. the integrity tests are not executed until the entire transaction has finished. Several transactions can in turn be combined by "`---`".

There are two motivations for this concept. On one hand there is a need for complex, indivisable transactions, on the other hand the user is released from the exact knowledge of the database schema. E.g. a sequence of instructions like

```
    insert into country values: ('Z-Country', bottom,
                                 2000000, democratic,
                                 <'Jones',
                                  list(tuple(9999, 'X-City',
                                             'Y-Square', 2))
                                 >,
                                 bottom
                                );
--- insert into person values:  ('Jones',
                                 list(tuple(9999, 'X-City',
                                            'Y-Square', 2)),
                                 60
                                );
```

is not permissible, since the new entity of the type *person* is used as a component prior to its creation by the second instruction. However, replacing "`---`" by "`||`" yields a transaction that is admissible and produces the intended result. This situation becomes even more complicated with the use of query-inserts and updates, where the results may strongly depend on the results of some other instruction.

When executing a complex transaction, the system permutes the order of its atomic instructions. In the resulting instruction sequence every instruction that changes the state of an object type on which the result of some other instruction depends is executed prior to the latter.

In order to determine this permutation a DAG[9] is constructed based upon these dependencies, with atomic instructions as nodes. A (directed) edge from i_1 to i_2 represents the fact that the result of i_2 depends on the result of i_1. The instructions are carried out in ascending order of the levels associated to them by this DAG.

If there are cycles, i.e. no DAG can be constructed, the meaning of the transaction is ambiguous and it is rejected.

7 Methodical Problems

In contrast to the various advantages of a declarative programming language as a compiler and target language there are several problems which have to be solved in translating the concepts of the EER calculus into PROLOG. Of course, all the concepts can be mapped to PROLOG but language constructs beyond the pure kernel are neccessary. The neccessity stems from the set and bag expressions which have to be computed in PROLOG by using predicates like `bagof`. The semantics of these predicates is defined with the cut even if they are built-in predicates [Nai86]. The impure constructs (e.g. the cut and negation as failure) are beyond the scope of *pure* logic programming and have no declarative semantics. This fact in turn, makes a clean formal semantics for the resulting programs difficult.

7.1 Representation of Sets

The EER model contains parametric data types like *list(s)*, *set(s)*, *bag(s)* and *tuple*$(s_1, \ldots, s_n)$. There is no difficulty to represent lists and tuples in PROLOG because it offers these constructs in the kernel. For sets and bags however, no direct corresponding language element exists. For reasons of efficiency we chose to represent sets and bags as ordered PROLOG lists so that single elements can easily be accessed, the empty set/bag can be represented by "`[]`", duplicates can be removed efficiently, and equality tests can be done by simple unification.

[9] directed acyclic graph

This neccessitates a sorting operation to be performed when a new bag or set is generated. But this sorting can be dropped in certain situations, e.g. if the AVG or SUM operation is applied to the set. This optimization step is actually done during the optimization phase.

Another point is that set management is not part of first-order logic. So we have to make use of the impure language constructs like `findall/3` (implemented with `assert/1` and `retract/1`) to be able to retrieve all instances of a term qualifying a given formula as in the bag expressions $-[t_1, \ldots, t_n \mid \delta_1, \ldots, \delta_m \ \wedge \ \varphi]-$.

7.2 Quantification

Quantification cannot be expressed *explicitly* in Horn logic which assumes free variables in the head of a rule to be implicitly universally quantified and the variables appearing only in the body are quantified existentially. The latter fact might lead to an identical (but wrong) translation of $-[\ldots \mid \ldots, \delta \wedge \varphi]-$ and $-[\ldots \mid \ldots \wedge \exists\, \delta\varphi]-$ as was pointed out in section 5.3.5. If pure first-order logic was considered the existential quantifier could be implemented simply as a test of the predicate $\varphi(x)$, supposed x is bound correctly at the time of the call.

7.3 Negation

As there is no negation in pure PROLOG the usual implementation is "negation as failure". This corresponds to the idea of the "closed world assumption" (CWA) which is used in database technology for negation.

This kind of implementation has the disadvantage that `not(X)` cannot bind any variables in X. Therefore, no subgoal of a clause must be negated if bindings are to be exported from it. This is one of the main reasons to use an evaluation strategy in which the variables are bound first and the qualifying formula is tested thereafter, whether it is true with these values. For a detailed discussion of negation in PROLOG see [Nai86].

7.4 Undefined Values

The EER calculus allows functions to yield an undefined value ($\perp$) or to have undefined arguments, due to the fact that $\perp$ is a "value" of every sort. The treatment of $\perp$ has to be done by a special representation in PROLOG. There are two possibilities to do this: treating it as a free variable (if it is regarded as something, that is in no conflict with any other value, i.e. it unifies with any term) or as a special atom (`bottom`). The latter way seems to be preferable. Otherwise each variable has to be tested on being bound to avoid unintended instantiation.

We regard every function in the EER model to be undefined if one or more arguments are undefined, every predicate to be false in this case. This fact has to be considered in the translation of operations and predicates of the data specification to PROLOG. So a special clause is added to the translation of an operation φ (analogously for predicates):

$$\varphi(X1,\ ...,\ Xn,\ \mathit{bottom})\ :-\ \mathit{member}(\mathit{bottom},\ [X1,\ ...,\ Xn]).$$

The PROLOG built-in operators and predicates (`+`,`-`,`*`, etc.) would react with run-time errors, if one of their arguments was `bottom`. So the standard assignment operator `is/2` cannot be used here.

8 Conclusion

The project has shown that it is possible to implement a prototype of a database system in a very short time (less than one year). The system supports all the concepts of the EER model and calculus which has initially been planned to provide a better (theoretical) framework for database design. Over and above that, the system has a data manipulation component whose language is very close to the calculus.

Being only a prototype, the system lacks some of the features conventional database systems offer, i.e. efficient secondary storage management, concurrency control, or a user-friendly interface. There are neither possibilities to specify storage structures like hash tables, b-trees etc. nor such to create indices. But these topics have not been that important for us because our aim was to show, that it is possible to make use of the many advantages of a logic programming language to build a system that supports database design. It is possible to model a database by specifying its types and schema in the semantical data model directly. Given these specifications tests on small amounts of data can immediately be run to check whether the design needs further modifications.

The fact that only the renunciation of the database system features mentioned above facilitates the use of the advantages PROLOG offers, makes it dubious whether PROLOG can serve as an implementation and query language for database systems. But it should be stressed that this point mainly concerns current PROLOG implementations, not logic programming in general.

Here and in related projects it has been shown that the declarative programming paradigma offers a lot of advantages. At present we see two principal points worth further studies in this context: firstly, the improvement of system-related features of declarative programming languages (e.g. secondary storage management) and secondly the development of declarative programming languages based on semantical data models.

Whilst the first point concerns system-related questions, which are not only interesting in the database context but for PROLOG in general, the second point would neccessitate a complete revision of the data model and execution strategies logic programming is currently based on. The EER model and calculus could be a starting point, but they would have to be modified so that e.g. data manipulation or meta queries about the schema could be expressed.

At present, we have the following concrete topics that should be studied in depth for our current system:

- We have not yet offered a *formal* proof that our translation is correct, i.e. the PROLOG queries are consistent with the semantics of calculus expressions.
- The pure main storage prototype is to be enhanced by secondary storage management capabilities.
- The optimization phase must be improved significantly beginning with the starting points mentioned briefly in section 5.3.8.
- A user-friendly interface is needed. The PROLOG interpreter's interface is insufficient. There should be a better conceivable query language, not the (somewhat complicated) calculus. This could at least be an SQL-like language incorporating the enhanced capabilities of the EER calculus.

Acknowledgement
Thanks to our colleagues Christine Müller and Karl Neumann for commenting on a draft version of this paper.

References

[AB85] H.-J. Appelrath and H. Bense. Zwei Schritte zur Verbesserung von Prolog-Systemen: DB-Unterstützung und Meta-Interpreter. In A. Blaser and P. Pistor, editors, *Datenbank-Systeme für Büro, Technik und Wissenschaft, GI-Fachtagung, Karlsruhe*, pages 161 – 176, 1985.

[AG87] S. Abiteboul and S. Grumbach. A logic-based language for complex objects. In *Proc. of the Workshop on Database Programming Languages*, pages 253 – 276, Roscoff, France, 1987.

[AKN86] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185 – 215, 1986.

[App87] H.-J. Appelrath. Das Eureka-Projekt PROTOS. In W. Brauer and W. Wahlster, editors, *Wissensbasierte Systeme, 2. Internationaler GI-Kongreß*, pages 1 – 11, München, 1987.

[BJ84] M. Brodie and M. Jarke. On integrating logic programming and databases. In L. Kerschberg, editor, *Proc. of the First Int. Workshop on Expert Database Systems*, pages 191 – 208, South Carolina, 1984.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Cambridge, MA, 1986.

[Con90] S. Conrad. Erste Erfahrungen mit dem ER-basierten Informationssystem QUEER: Test und Fallstudie. Studienarbeit, TU Braunschweig, 1990.

[CW89] W. Chen and D.S. Warren. C-Logic of complex objects. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369 – 378, 1989.

[EDG86] H.-D. Ehrich, K. Drosten, and M. Gogolla. Towards an algebraic semantics for database specification. In *Proc. of the 2nd IFIP Working Conference on Database Semantics (DS-2).* North Holland, November 1986.

[GDK87] G. Goos, R. Dietrich, and P. Kursawe. Prolog-Arbeiten in Karlsruhe. In W. Brauer and W. Wahlster, editors, *Wissensbasierte Systeme, 2. Internationaler GI-Kongreß*, München, 1987.

[GH91] M. Gogolla and U. Hohenstein. Towards a semantic view of an extended entity-relationship model. In *ACM Transactions on Database Systems*, 1991. To Appear.

[GMN84] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, pages 153 – 185, 1984.

[Gog89] M. Gogolla. Zur Übersetzung eines ER-Kalküls nach Prolog. In *Abstracts des Workshops ,,Grundlagen von Datenbanken", Informatik-Bericht 89-02.* Technische Universität Braunschweig, 1989. Also: Internal Note, Technische Universität Braunschweig, Feb. 1989.

[Gua86] N. Guarino. Representing domain structure of many-sorted Prolog knowledge bases. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Proc. of the Workshop on Foundations of Logic and Functional Programming*, pages 168 – 183, 1986.

[HG88] U. Hohenstein and M. Gogolla. Towards a semantic view of an extended entity-relationship model. Technical Report 88-02, Technische Universität Braunschweig, Braunschweig, 1988.

[HG89] U. Hohenstein and M. Gogolla. A calculus for an extended entity-relationship model incorporating arbitrary data operations and aggregate functions. In *Proc. of the Int. Conference on the Entity-Relationship Approach to Data Modelling, Rome*, 1989.

[HNSE87] U. Hohenstein, L. Neugebauer, G. Saake, and H.-D. Ehrich. Three-level specification using an extended entity-relationship model. In R.R. Wagner, R. Traunmüller, and H.C. Mayr, editors, *Informatik-Fachberichte Band 143*, pages 58 – 88. Springer, 1987.

[Hoh88] U. Hohenstein. Automatic transformation of entity-relationship schemas into relational schemas. Technical Report 88-10, Technische Universität Braunschweig, Braunschweig, 1988.

[Jas89] H. Jasper. Prolog und Datenbanken: Ein Schema für die externe Verwaltung von Prolog-Programmen. In T. Härder, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft, GI/SI-Fachtagung*, pages 216 – 230, Zürich, 1989.

[Joh90] P. Johannesson. MOLOC: Using Prolog for conceptual modelling. In *Proc. of the 9th Int. Conference on the Entity-Relationship Approach*, pages 301 – 314, October 1990.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *ACM-SIGMOD Proc. of the Int. Conference on Management of Data*, pages 134 – 145, 1989.

[Kup87] G.M. Kuper. Logic programming with sets. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 53 – 59, San Diego, 1987.

[KW89] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-Logic revisited). In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379 – 393, 1989.

[Mai84] D. Maier. Databases in the fifth generation project: Is Prolog a database language ? In G. Ariav and J. Clifford, editors, *New Directions for Database Systems*, pages 18 – 34. 1984.

[Mey90] B. Meyer. Ein Prototyp eines ER-basierten Informationssystems: Methodische Grundlagen beim Einsatz von Prolog. Diplomarbeit, TU Braunschweig, 1990.

[Nai86] L. Naish. *Negation and Control in Prolog.* Springer, New York, 1986.

[Nic85] J.-M. Nicolas. Logic databases. In *Wissensbasierte Systeme*, 1985.

[Omo86] E.G. Omodeo. Hints for the design of a set calculus oriented to automated deduction. In M. Boscarol, L. Carlucci Aiello, and G. Levi, editors, *Proc. of the Workshop on Foundations of Logic and Functional Programming*, 1986.

[P+84] D.S. Parker et al. Logic programming and databases. In L. Kerschberg, editor, *Proc. of the First Int. Workshop on Expert Database Systems*, pages 34 – 48, South Carolina, 1984.

[Wei86] H. Weigand. Conceptual models in Prolog. In Jr. T.B. Steel and R. Meersman, editors, *Proc. of the IFIP WG 2.6 Working Conference on Data Semantics (DS-1)*, pages 59 – 69. Elsevier Science Publishers B.V. (North Holland), 1986.

[Wes90] G.D. Westerman. Ein Prototyp eines ER-basierten Informationssystems: Übersetzung des ER-Kalküls nach Prolog. Diplomarbeit, TU Braunschweig, 1990.