Data & Knowledge Engineering, North-Holland, Amsterdam (1996)

Drafting ER and OO Schemas in Prototyping Environments

Bernd Meyer [a] Gerd D. Westerman [b] Martin Gogolla [c]

[a] *Monash University, Dept. of Computer Science, Clayton, Vic 3168, Australia, E-Mail: berndm@cs.monash.edu.au, Fax: ++61-3-99 05 90 42*

[b] *sd&m GmbH & Co. KG, Am Schimmersfeld 7a, D-40880 Ratingen, Germany, E-Mail: gerd.westerman@sdm.de, Fax: ++49-21 02-99 57 50*

[c] *Universität Bremen, FB Mathematik und Informatik, Postfach 33 04 40, D-28334 Bremen, Germany, E-Mail: gogolla@informatik.uni-bremen.de, Fax: ++49-4 21-2 18-42 69*

Abstract

The system Queer is a prototype of an information system design tool 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, integrity constraints, and data-manipulation statements into Prolog programs. All features mentioned are implemented in form and extent as described here.

Keywords: Conceptual Modelling; Entity Relationship Model; Object-Oriented Modelling; Logic Programming; Prototyping.

1 Introduction

Conceptual modelling plays the central role among the different steps in database design. The result of the modelling process is a comprehensive and first formal description of the part of the world to be modelled. Usually, this step is done by employing a semantic data model like TAXIS [35], SDM [17], IRIS [32], or IFO [1]. An overview can be found in the excellent survey articles [26,38]. In the database community, Chen's Entity-Relationship Model [5] is nowadays well-tried and widely accepted to be an excellent candidate for this phase of database design. Nevertheless, numerous extensions have been proposed enhancing the expressiveness of the original proposal, among many others [9,33,45].

However, in order to achieve a formal and consistent description of complex application domains it is necessary and desirable to document the resulting details concerning real world in a systematic way. Therefore from the very beginning, database designers claimed to support the design process by appropriate tools. In this paper we introduce the system QUEER which directly supports an extended Entity-Relationship model on its front-end. QUEER stands for *QU*ery system for an *E*xtended *E*ntity *R*elationship model. QUEER can be seen as a design tool or alternatively as a prototype of a (simple) user-interface for an Entity-Relationship information system. Our approach continues the tradition of various efforts describing information system design tools (see the discussion of related work at the end of this introduction). The advantage of QUEER can be seen in its semantic well-foundedness, in its rich data model, and in its support of an expressive language for queries and constraints. All features mentioned are implemented in form and extent as described here. Work on this project has previously been reported in [16].

Our system is implemented in PROLOG, but our aim has not been to provide a logic programming language with database support, but rather to build a prototype with a clean, calculus-oriented, and highly expressive front-end exploiting the power of logic programming as a compiler and target language. In contrast to related projects, 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. Thus the user does not need to work with PROLOG constructs, but with specialized languages. As the query language we choose a particular calculus, which has been developped especially for the extended Entity-Relationship model [10] and which is semantically well-founded, safe, and relationally complete [21,15,12]. Another powerful calculus for an extended Entity-Relationship model has been proposed in [37]. Interestingly, a corresponding algebra has been implemented in PROLOG, as well. The fact that our query language has a precisely defined semantics makes it easier to implement higher level query languages or other user interfaces. 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 the conceptual model while interacting with the system.

1.1 System Architecture

The basic idea behind the system is to store database objects as PROLOG facts and to translate queries expressed in terms of the extended Entity-Relationship calculus into PROLOG goals, thus using the inference engine for data retrieval.

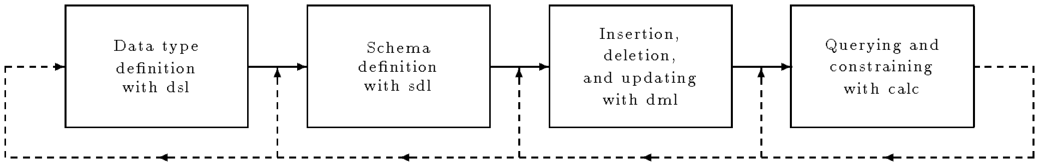
Internally, the data is treated as unnormalized, nested relations, where objects are seen as complex tuples. Relating of objects is done by means of surrogate handling. QUEER consists of a set of compilers, which translate the four different languages necessary for the usage of the system into PROLOG 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, thus providing a precisely defined interface for database interactions. 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. Our system uses the standard PROLOG front-end; from 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 extended Entity-Relationship schemas allowing user-defined data types.

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.

calc Language for queries and constraints on the basis of the extended Entity-Relationship calculus.



A typical session, starting from scratch, consists of (at least) four steps, each of which requires one of these languages. In accordance with a three level specification paradigm of database modelling [8,22] 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. 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 test (in principle arbitrary static) integrity constraints or to define views on the database by means of pre-translated queries. Having been translated into Prolog programs by the *calc* compiler, the 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 database use. 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; the languages are completely based upon the conceptual schema and the extended Entity-Relationship calculus.

1.2 Related Work

There has been a considerable amount of work on database design tools implemented in logic or functional languages and fitting for Entity-Relationship or related data models. [3] introduces the database design expert system SECSI implemented in Prolog. The system first generates a semantic network on the basis of application descriptions given in a subset of natural language, in a formal language or by means of a graphical interface and afterwards simplifies the network to reach flat relations. [4] discusses how ER diagrams are translated into relational and network database schemas. Diagrams and schemas are represented as Prolog facts and rules. KPSP [18] is a general knowledge programming system close to structural semantic networks and implemented in Prolog. [18] also explains how a simple ER model is mapped into KPSP. In [43] an object-relationship-situation (ORS) model is proposed and a system written in the AI language PEARL executing ORS data specifications in order to construct conceptual schemas is described. In [40] a Prolog implementation of the ambitious expert system OICSI for information system design is outlined. Starting with a description in restrained natural language, the system interactively generates conceptual schemas in form of semantic networks. [49] shows how predicate logic can be employed for conceptual modelling and how the logical axioms can be translated into Prolog rules. CHRIS [11,46] is an expert system design tool covering the complete design cycle including conceptual and logical design and a database interface generation phase. It is written in Prolog. TSER [24] is a LISP-based implementation of the Two-Stage Entity-Relationship approach to data modelling. Among other components, the system provides facilities for the modelling constructs and mapping algorithms to yield relational schemas. In [30] the E^2R model and a verfication and transformation system implemented in Prolog is presented. One part of the system tests the correctness of a conceptual database specification; the other part translates the ER schema into a relational one. A conceptual language (CPL) derived from natural language theory for specifying static and dynamic components of a conceptual model is proposed in [7]. The language allows for specialized constraints and is implemented in Prolog. ERMCAT [25] is a rule-based system prototype implemented in OPS5 and automating the ER clustering process, i.e., grouping of entity and relationship types to form entity types at higher levels of abstraction. KORTEX [29] is an expert database shell implemented in LISP and supporting the integrated specification and management of data and knowledge. It is based on an extended Entity-Relationship model [45]. [20] describes a translation of schemas from the extended Entity-Relationship model to the relational model. This approach has the advantage that secondary storage management can be supported by a relational database system. In [44] an expert system providing intelligent assistance in the conceptual modelling phase is discussed and a prototype of the system designed in PROLOG supporting structure modelling is mentioned.

In [2] a translation of an extended Entity-Relationship model into the deductive database language LDL is proposed. MOLOC [27] is a prototype semantic database system based on PROLOG. [27] discusses also the merits of the system in teaching conceptual modelling. The approach presented in [31] also employs PROLOG as a tool for implementing various Entity-Relationship concepts with the focus on weak entity types and multivalued attributes. In [28] the deductive database language Datalog serves as the basis for implementing enhanced Entity-Relationship schemas and database states simliliar to our approach but concentrating on functional and inclusion dependencies as constraints. The design of data dictionaries is the topic of [39]. There, illegal design decisions are revealed by abductive reasoning. The implementation of a quite general constraint language based on the deductive database model is described in [41]. The paper concentrates on constraints and derivation rules based on a data model close to the classical relational one without explicit abstraction mechanisms. The OBSERV approach of [47] focuses on an object-oriented way for building systems from a collection of objects with well-defined and restricted interfaces. The system offers logic programming and graphical specification interfaces. In [42] the design of object schemas is supported by a PROLOG prototype. [34] studies system prototyping in the context of an object-oriented environment. The focus in that approach is on database schema extensibility, in particular on logical and physical extensibility. Recently, a general software specification tool based on the Entity-Relationship model has been put forward in [36].

The rest of the paper is organized 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. The details of the translation are covered in Section 4. 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:

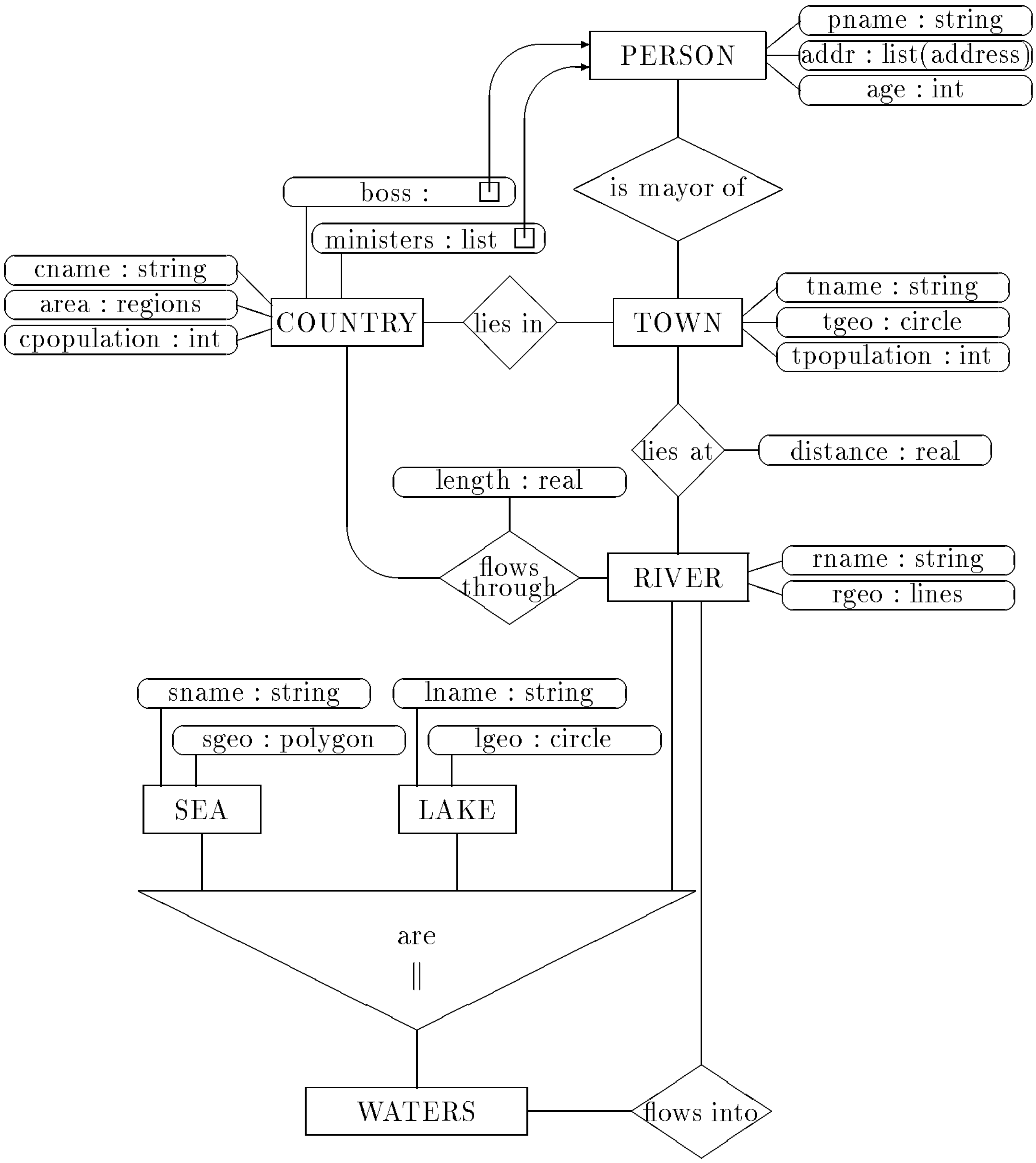
- Every TOWN lies in a COUNTRY and may lie at one or several RIVERS.
- RIVERS flow through COUNTRIES and flow into some kind of WATERS.
- SEAS, RIVERS, and LAKES are such WATERS.
- A PERSON may be the mayor of a TOWN, or the boss or a minister of a COUNTRY.

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 a corresponding Entity-Relationship diagram. 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 subsequent sections; for more motivation concerning the background we refer to [21,15]).

```
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
predicates    cccut  : circle x circle
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_data_types
```



The translation of the above input into PROLOG is 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 on 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).
```

Likewise, there is a predicate `db_pred` that specifies the syntactic properties of the user-defined predicate `cccut`.

```
db_pred(cccut, [circle, circle]).
```

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`. The semantics of the function `pdist` (determining the distance between two points) is a PROLOG rule which evaluates the function for given `P1, P2`.

```
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).
```

In very much the same way a PROLOG-predicate defining the semantics of every user-defined predicate is derived from the specification. In the case of `cccut` which tests if two circles are intersecting this definition has the form

```
cccut(C1, C2) :-
   center(C1, X1), center(C2, X2), pdist(X1, X2, D),
   radius(C1, R1), radius(C2, R2), D <= R1 + R2.
```

Like the data types, Entity-Relationship schemas are given to the 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,
                   components     boss       : person,
                                  ministers  : list(person);

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

  construction types are input    sea, lake, river;
                         output   waters;
end_schema country_town_waters.
```

In PROLOG, we use 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) ],
             [ ( 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. Entities are represented by ground facts of the predicate `entity` matching the entity type definition, e.g.:

```
entity(country,
       [country42,"france",...,52900000,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 = 52900000`.

Each relationship type is 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] ).
```

As mentioned above, the query language we use is the extended Entity-Relationship calculus [21,15], 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, arbitrary nested subqueries and aggregation functions are allowed.

Summarizing, it can be observed that an extended Entity-Relationship model has been employed for the conceptual modelling process and the result of the modelling process has been translated 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. Secondly, 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 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 [21,15] for details and the section on calculus expressions for an example).

3 The Extended Entity-Relationship Model and Calculus

Here, we shortly recapture the fundamental notions of the underlying extended Entity-Relationship 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 [21,15]. There, also full motivation for all concepts can be found.

3.1 The Extended Entity-Relationship Model

3.1.1 Data Types

The extended Entity-Relationship 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, PREDS giving the names of the data sorts, operation and predicate symbols, resp., together with
- functions $source : \text{OPNS} \to \text{DATA}^*$, $destination : \text{OPNS} \to \text{DATA}$, and $arguments : \text{PREDS} \to \text{DATA}^+$ 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

$$list(\mathrm{s}),\ bag(\mathrm{s}),\ set(\mathrm{s}),\ \text{and}\ tuple(\mathrm{s}_1, \ldots, \mathrm{s}_n).$$

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

$$point = tuple(real,real);\ circle = tuple(point,real);\ lines = list(point)$$

Additionally, functions and predicates can be defined, e.g.:

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

3.1.3 Aggregation Functions

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

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

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 $\mathrm{d} \in \mathrm{DATA}$. 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 extended Entity-Relationship model is the extended Entity-Relationship schema. Such a schema gives 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, CONSTRUCTION together with functions
- *participants* RELTYPE → ENTTYPE^+,
- *source* ATTRIBUTE → ENTTYPE ∪ RELTYPE,
- *destination* ATTRIBUTE → { d, *set*(d), *bag*(d), *list*(d) | d ∈ DATA },
- *source* COMPONENT → ENTTYPE
- *destination* COMPONENT → { e, *set*(e), *bag*(e), *list*(e) | e ∈ ENTTYPE }, and
- *input*, *output* CONSTRUCTION → $2^{\text{ENTTYPE}} - \emptyset$.

There are several additional conditions these schemas have to fulfill: 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.

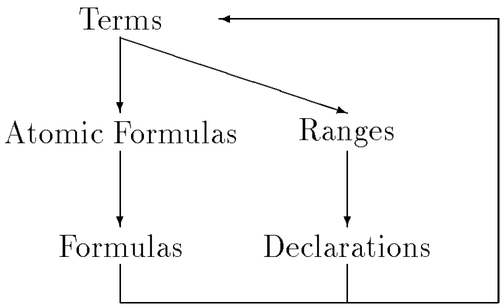
- ENTTYPE := { PERSON, COUNTRY, TOWN, RIVER, SEA, LAKE, WATERS }
- RELTYPE := {*is_mayor_of, lies_in, lies_at, flows_through, flows_into* }
- ATTRIBUTE := { *pname* : PERSON → *string*, *addr* : PERSON → *list*(*address*), ... } This specifies not only the set ATTRIBUTE but the functions *source* and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

The interpretation of an extended Entity-Relationship 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. The basic idea behind construction types is to take (not necessarily all) input entities and to distribute them disjointly among the output types. Technically, 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 supertype) of the specialized entity. In the case of generalization, there may be 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, there is 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 Extended Entity-Relationship Calculus

The extended Entity-Relationship calculus defined for the above introduced extended Entity-Relationship model allows to formulate queries against an extended Entity-Relationship database (i.e., an interpretation or instance of an extended Entity-Relationship 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 forms of terms:

- Variables and applications of operation symbols are terms. Operation symbols in this context are: data operations, standard operations of the parameterized 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* (if $point_1$ and $point_2$ are variables of data type *point*).
- If we have a type construction, then $\mathrm{e}_{in}(\mathrm{t}_{out})$ and $\mathrm{e}_{out}(\mathrm{t}_{in})$ are terms of sort e_{in} or e_{out}, resp., assuming e_{in} is an input type and e_{out} is 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, resp. Additionally, role names are also allowed in this place instead of numbers (e.g., li.TOWN or li.COUNTRY).
- The most important form of terms is the following one.

  -[t_1, ..., t_n | δ_1, ..., $\delta_k \wedge \phi$]-

  The value of this expression is a bag (or multi-set) allowing multiple occurrences of elements. The term has the sort $bag(tuple(\mathrm{sort}(\mathrm{t}_1),...,\mathrm{sort}(\mathrm{t}_n)))$. Here, the t_is are the result terms, the δ_is are variable declarations and ϕ is a qualifying formula. In the case n=1, the sort of the term is simply $bag(\mathrm{sort}(\mathrm{t}_1))$. Of course, one could as well use a more talkative syntax like

  `select` t_1, ..., t_n `from` δ_1, ..., δ_k `where` ϕ

instead of -[t_1, ..., t_n | δ_1, ..., $\delta_k \wedge \phi$]-. 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*)), resp. 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 $circle_2$ 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 *l*ist *t*o a *b*ag).

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 $r \in$ REL-

TYPE 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 *l*ist *t*o a *s*et and *bts* a *b*ag *t*o a *s*et).

We have started with a simple form of declarations for the ease of demonstration. The final form of declarations 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, too. 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 the calculus are 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*)]- $\leq$ 65)

This example reveals the non-hierarchical structure 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 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. This approach was taken for two reasons: first to remove redundancy from the database; second 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 — therefore every expression in the *dml* — has a unique type, independent of the contents of the database. Thus an expression, which is validated at compile time, cannot produce type 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, resp. Not even the procedures used for applying functions to data values and object values have to be structurally different. For example, 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, operations generated by the *sdl* compiler and the *dsl* compiler have quite a similar appearance.

4.1 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 that it 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 (with sorted elements to achieve more efficient implementations of several operations). 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 ($\perp$). In this case, the specific operation has to return the undefined value, and the predicates have to fail.

4.1.1 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.

4.1.2 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 4.3). An example is the operation *start*: $lines \rightarrow point$ which is defined as

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

A clause

```
start(Lines, Res) :- sel(1, Lines, 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

$select(Template,\ Res)$

where *Template* is a tuple with n components whenever the associated tuple sort has n components. *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.

4.1.3 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 such that the usual mathematical notation given in the specification is converted to a chain of predicate calls implementing the arithmetical parts of the predicate evaluation in the usual PROLOG-style. Thus the definition of `cccut` becomes

```
cccut(C1, C2) :- center(C1, X1), center(C2, X2),
                 pdist(X1, X2, D), radius(C1, R1),
                 radius(C2, R2), D <= R1 + R2.
```

4.1.4 Restrictions

An important restriction concerning admissible definitions of sorts, operations, and predicates is that no recursion may be used in either of their definitions. Neither may there be more than one definition of any equation, i.e. case distinctions cannot be used in function definitions. This would, however, be a straight-forward extension of the presented schema.

4.2 Extended Entity-Relationship Schemas

4.2.1 Syntax of Schema Definitions

Like data specifications, extended Entity-Relationship schemas in QUEER are defined textually. A sketch of a sample definition has been outlined in Section 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, resp. 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 must not be nested.

4.2.2 The Meta-Database

The translation of extended Entity-Relationship data into PROLOG clauses resembles that into relational schemas in some respects, but due to the greater expressiveness of horn clauses with function symbols it is somehow more straight forward. Basically, three types of PROLOG clauses are generated for every schema object, which form QUEER's meta-database (a schema object is an entity type, a relationship type or a construction type):

- 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 (analogously for components and participants). 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 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)` consisting 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. As mentioned above, the instance of this predicate for the type *country* in our running example is:

```
entity_type( country,
           [ ( cname, string ), ( area, regions ),
             ( cpopulation, int) ],
           [ ( 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 `], _ ).`
$comp$`(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, i.e., there is an index i such that $V_i = A$. 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, _ ], _ ).
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) .
```

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 ) ],
              [ ( length, 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). An example of such a selection clause is:

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

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 ) .
```

In addition to that, appropriate type declarations for each relationship type are generated that resemble those for normal predicates defined on data values, for relationships are handled like binary predicates.

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 corresponding output entity and vice versa (with `C` and `B` interchanged).

III. As for entity types and relationship types, the group II rules are declared as operators by `db_opn/3`.

A fragment of the rules derived from the definition of the construction type *are* is:

```
cons_type( are, [sea, lake, river], [waters] ) .
...
lake( C, B ) :-
   construction( are, B, C ),
   entity( lake, [C | _], _ ) .

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

4.2.3 Keys

In addition to the predicates described above, clauses are generated that describe the set of key attributes for each entity type. These clauses are used by the *dml* interpreter to verify transactions against the implicit integrity constraints of the extended Entity-Relationship 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, i.e., that part of the schema consisting only of key attributes and components.

4.3 Calculus Expressions

4.3.1 Introductory Example

For the sake of clarity, we will first give a small example to show the necessary steps for the translation of calculus expressions. Consider again the query `-[ pname(P) | (P: person) & age(P) < 18 ]-`. The translated code has been presented in Section 2. The bag-valued expression of the calculus is translated to an application of the PROLOG predicate `bagof/3`. Thus the solution of this query (bound to variable `Result`) will be the list of all instances of `PName` for which the second argument of `bagof` can be satisfied. 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 `age/2` and `pname/2` generated in the translation of the schema definition return the age or the name, resp., 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.

4.3.2 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 which the innermost function application appears as the leftmost subgoal. 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.

4.3.3 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 or the list of surrogates, resp., during the evaluation of its PROLOG counterpart.

4.3.4 Translation of Calculus Constructs

In this section we present the PROLOG counterparts of the language elements of the calculus. In Table 1 and Table 2 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 exception 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 4.3.2) is V_i. Ω is the PROLOG function corresponding to ω. Table 1 and Table 2 are explained in the following:

Terms: For constants and variables a `true` has to be generated as code 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 4.3.2), e.g., the term

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

calculus expression		prolog code	return variable
term	t	T	V
formula	φ	F	-

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($V_1, \ldots, V_n$)	
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	$\varphi_1\ \Longrightarrow\ \varphi_2$	(`not` F_1) `or` F_2	
equivalence	$\varphi_1\ \Longleftrightarrow\ \varphi_2$	(F_1, F_2) `or` (`not` F_1, `not` F_2)	
existential quant.	$\exists\delta : \varphi$	*exists (D, F)*	
universal quant.	$\forall\delta : \varphi$	*not exists (D, not F)*	

Table 1
Translation of calculus terms and formulas to Prolog

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

```
tgeo(T, C1), center(C1, P1), lgeo(L, C2),
center(C2, P2), pdist(P1, P2, 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

calculus expression		prolog code	return variable
declaration	δ	D	V
range	ρ	R	V

Declarations

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

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
setof	setof(r)	`setof(`W`, `R`, S),subset(`V`, S)`	V

Table 2
Translation of calculus declarations and ranges to Prolog

Prolog predicate `is/2` but returns `bottom` ($\perp$) if one of the operands is non-numeric, a division by zero appears, or one of the arguments is `bottom`.

There is no need for a separate handling of the different kinds of operations, which include selection of data-, object-, or multi-valued attributes, selection of entities participating in a relationship, type-conversion functions $s_{in}(t_{out})$, $s_{out}(t_{in})$, and user- and pre-defined operations (including aggregations).

Bag expressions are translated to `bagof` goals. The order of the subgoals inside the `bagof` expression has to be D_1, ..., D_m, F,T_1, ..., T_n, because the free variables inside F have to be bound by D_1, ..., D_m first. Then the instances can be tested for satisfaction of the formula ϕ by the subgoal F. For every instance satisfying F, the return values V_1, ..., V_n are computed afterwards by T_1, ..., T_n.

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. Special attention has to be paid to `exists` and `or`, because the implementation of the bag-valued terms by `bagof` 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 above — inside 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, an existentially 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), !.
```

Under these assumptions, 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 exists` clauses is a PROLOG *and*. `exists` and `not` are declared to be monadic operators in 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. For the setof the PROLOG predicate `setof` generates a set `S` of all Ws where W is the return variable of R, the translation of r. The `subset` predicate generates all subsets of `S`.

4.3.5 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 do not hold 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 necessity 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.

However, `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 in bag-valued terms: their PROLOG counterparts will be placed to the left of a subgoal sequence computing the qualifying formula and the result terms. The PROLOG counterparts of declarations then 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.

4.3.6 Comparison to the Translation into the Relational Calculus

In Section 1 we have surveyed several lines of related work. However, a comparison of our approach to a translation into the relational calculus is of particular interest, for the relational model still is the most widely used database model in real-world applications. Thus, using the relational model has the advantage that DBMS techniques like secondary storage management, transaction management, etc. can be supported at almost no extra cost. On the other hand, a number of high-level features of the EER calculus are notoriously difficult to translate.

In [20] a translation of the same EER model that we are using here into the relational model is described. More precisely, EER calculus queries are translated to relational 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.

4.3.7 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 4.3.4) 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, the following statement is valid in pure Prolog: *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 heuristics has to be developped for another conceivable strategy which re-orders the subgoals by placing the stronger specified ones (e.g., with fewer free variables) to the left.

4.3.8 Examples

Example 1: The following query retrieves the names of ministers having only addresses in towns of their country. This query is formulated in the calculus in the following way.

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

The result of the translation before the optimization phase is as follows.

```
query(Solution) :-
     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 `MinSet` (recall that only set-valued ranges are allowed in the calculus). On backtracking, `member(P, MinSet)` yields every single minister `P` in 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 `Solution`. After the optimization phase the optimized PROLOG code looks as follows.

```
query(Solution) :-
     findall(PName,
             (entity(country, [C | _], [_, Min]),
              lts(Min, MinSet), member(P, MinSet),
              not
                exists (entity(person, [P, _, Addr, _], _),
                        lts(Addr, AddrSet),
                        member((_, City, _, _), AddrSet),
                        not exists
                         (entity(town,
                                 [T, City, _, _], _),
                                 relship(
                                   lies_in, [T, C], [])) ),
              entity(person, [P, PName, _, _], _) ),
             Solution ).
```

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, more abstract example shows how the transitive closure of a binary relationship can be expressed in the calculus and what its translation is. The query is based on the following definition of the transitive closure R^* of a binary relationship R:

$$R^* = \{(x, y) \mid \exists\, x_1, \ldots, x_k \;:\; (x, x_1) \in R \wedge (x_1, x_2) \in R \wedge \cdots \wedge (x_k, y) \in R\}$$

Let the following part of a schema definition be given:

```
entities
        person      attributes      pname : string;
relationships
        ancestor    participants    parent: person,
                                    child : person;
```

Then the following calculus term computes the transitive closure of *ancestor*, i.e. all pairs of persons for which one person is ancestor of the other:

```
-[ pname(child(Start)), pname(parent(End))
        | (Start : ancestor), (End : ancestor)
        & (exists (A : setof(ancestor)))
              End in A & Start in A &
              (forall (Ai : ancestor))
                  Ai in A &
                  (Ai = End or (exists (Aj : ancestor))
                                  Aj in A &
                                  Ai.parent = Aj.child)
]-
```

`setof` yields some subsset of a range and `X in Set` is the $\in$-Relationship. The translation generates:

```
query(Solution) :-
    bagof((X, Y),
          (relship(ancestor, Start, _),
           relship(ancestor, End, _),
           exists(
             setof((ancestor, A_R),
                   relship(ancestor, A_R, _), A_S),
             subset(A_S, A),
             member((ancestor, End), A),
             member((ancestor, Start), A),
             not
              exists(relship(ancestor, Ai, _),
                     not (member((ancestor, Ai), A),
                          (    (ancestor, Ai)
                             = (ancestor, End)
                           or exists(
                               relship(ancestor, Aj, _),
                               member((ancestor,Aj), A),
                               parent((ancestor, Ai) PC),
                               child((ancestor, Aj), PC))
                          ))))),
           child((ancestor, Start), C), pname(C, X),
           parent((ancestor, End), P),  pname(P, Y)
          ),
          Solution
         ).
```

There is some potential for optimization in this query: The calls of the relationship selection predicates `parent` and `child` are unfolded to their definition, therby unifying their arguments `Ai` and `Aj`, resp. The calls of `pname` are unfolded as well, like in the last example.

Note, that optimization as described in the previous subsection does not unify both branches of the `or` with Variables outside of the `or`. If this had been done the result would have been wrong with a clause like "`true or exists ...`".

```
query(Solution) :-
    bagof((X, Y),
          (relship(ancestor, [Start, C], _),
           relship(ancestor, [P, End], _),
           exists(setof((ancestor, A_R),
                        relship(ancestor, A_R, _), A_S),
                  subset(A_S, A),
                  member((ancestor, End), A),
                  member((ancestor, Start), A),
                  not exists(
                   relship(ancestor, [PC, _], _),
                   not (member((ancestor, [PC, _]), A),
                       ( (ancestor, [PC, _])
                           = (ancestor, End)
                         or exists(
                           relship(ancestor, [_, PC], _),
                           member((ancestor, [_, PC]), A),
                           relship(ancestor, [PC, _], _),
                           relship(ancestor, [_, PC], _),
                       ))))),
           entity(person, [C, X]),
           entity(person, [P, Y]) ),
          Solution ).
```

5 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.

5.1 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[1]. The values for non-key attributes may be specified or set to *bottom*.

```
insert into town values: ('Berlin',
                           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'>,   <'Germany'>,   800);
        (<'Main'>,    <'Germany'>,   300);
        (<'Isar'>,    <'Germany'>,   150);
        (<'Elbe'>,    <'Germany'>,   400);
```

[1] key components are specified by a list of their key attribute values enclosed in pointed brackets like in the example in section 5.3

```
        (<'Wolga'>,   <'Russia'>,   1200);
```

This adds five 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) & waters(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 = 'Germany'
        ]-
```

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 := 'Germany', C.boss := P |
           (C : country), (P : person) &
           P.name = 'Kohl' & C.cname = 'Deutschland'
        ]-
```

assigns to the country named 'Deutschland' — or perhaps the countries, if there is more than one — the new name 'Germany' 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.

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 them must of course be deleted.

By means of a cross-reference 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 stated above is rejected.

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[2] 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

[2] directed acyclic graph

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.

6 Conclusion

The design and implementation of QUEER has shown that it is possible to implement a prototype of a database system design tool in a very short time (less than one year). Our project has derived great benefit from employing PROLOG as a compiler and target language. The system supports all concepts of the extended Entity-Relationship model and calculus which has initially been planned to provide a better theoretical framework for database design. In addition, the system has a data manipulation component whose language is very close to the calculus. Nevertheless, the system is far from being a perfect professional tool. At present, the following topics should be studied in depth for our current system:

- The optimization phase must be improved significantly.
- A user-friendly interface is needed. The PROLOG interpreter's interface is insufficient. A better conceivable query language than the (sometimes complicated) calculus has to be implemented. This could at least be an SQL-like language [23] incorporating the enhanced capabilities of the extended Entity-Relationship calculus.
- The integrity monitoring capabilities for (at least) static integrity constraints should be enhanced and integrated into *sdl*.

Further work in this area has been done in the KORSO project [48]. A part of this project was the development of the object description language TROLL *light* [13,19] whose static component corresponds to the extended Entity-Relationship model enhanced by inference rules and an SQL-like calculus for queries, inference rules, and static integrity constraints. An animator [14] can be used for certification of a specification and a proof support system helps to verify object properties formally [6].

Acknowledgement

Thanks to our colleagues Martin Erwig, Christine Müller, and Karl Neumann for their honest words on a draft version of this paper and to Uwe Hohenstein for earlier common work on the calculus.

References

[1] S. Abiteboul and R. Hull. IFO – A Formal Semantic Database Model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[2] D. Ackley, R.P. Carasik, T. Soon, D. Tryon, E. Tsou, S. Tsur, and C. Zaniolo. System Analysis for Deductive Database Environments: An Enhanced Role for Aggregate Entities. In Hannu Kangassalo, editor, *Proc. 9th Int. Conf. on Entity-Relationship Approach*, pages 129–142, 1990.

[3] Mokrane Bouzeghoub, Georges Gardarin, and Elisabeth Matais. Database Design Tools: An Expert System Approach. In A. Pirotte and Y. Vassiliou, editors, *Proc. 11th Conf. Very Large Data Bases*, pages 82–95, 1985.

[4] H. Briand, H. Habrias, J.-F. Hue, and Y. Simon. Expert System for Translating an E-R Diagram into Databases. In King-Sun Fu, editor, *Proc. 4th Int. Conf. on Entity-Relationship Approach*, pages 199–206. IEEE, 1985.

[5] P.P. Chen. The Entity-Relationship Model – Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[6] S. Conrad. On Certification of Specifications for TROLL *light* Objects. In H. Ehrig and F. Orejas, editors, *Proc. 9th Workshop on Abstract Data Types – 4th Compass Workshop (ADT'92)*, pages 158–172. Springer, Berlin, LNCS 785, 1994.

[7] Frank Dignum, T. Kemme, W. Kreuzen, Hans Weigand, and Reind P. van de Riet. Constraint Modelling Using a Conceptual Prototyping Language. *Data & Knowledge Engineering*, 2:213–254, 1987.

[8] 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.

[9] R. Elmasri, J. Weeldreyer, and A. Hevner. The Category Concept: An Extension to the Entity-Relationship Model. *Data & Knowledge Engineering*, 1:75–116, 1985.

[10] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, 9(2):157–204, 1992.

[11] Antonio L. Furtado, Marco A. Casanova, and Luiz Tucherman. The CHRIS Consultant. In Salvatore T. March, editor, *Proc. 6th Int. Conf. on Entity-Relationship Approach*, pages 515–532. North-Holland, 1987.

[12] M. Gogolla. *An Extended Entity Relationship Model. Fundamentals and Pragmatics*. Springer, Berlin, LNCS 767, 1994.

[13] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'92)*, pages 17–32. Springer, Berlin, LNCS 722, 1993.

[14] M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the ER Approach in an OO Environment. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. 12th Int. Conf. on Entity-Relationship Approach (ER'93)*, pages 382–395. Springer, Berlin, LNCS, 1994.

[15] M. Gogolla and U. Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Transactions on Database Systems*, 16(3):369–416, 1991.

[16] M. Gogolla, B. Meyer, and G.D. Westerman. Drafting Extended Entity-Relationship Schemas with QUEER. In T.J. Teorey, editor, *Proc. 10th Int. Conf. on Entity-Relationship Approach*, pages 561 – 585, 1991.

[17] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

[18] Sangki Han and Jung W. Cho. KPSP: A Knowledge Programming System based on PROLOG. In King-Sun Fu, editor, *Proc. 4th Int. Conf. on Entity-Relationship Approach*, pages 2–9. IEEE, 1985.

[19] R. Herzig, S. Conrad, and M. Gogolla. Compostional Description of Object Communities with TROLL *light*. In C. Chrisment, editor, *Proc. Basque Int. Workshop on Information Technology (BIWIT'94)*, pages 183–194. Cépaduès-Éditions, Toulouse, 1994.

[20] U. Hohenstein. Automatic Transformation of an Entity-Relationship Query Language into SQL. In F. Lochovski, editor, *Proc. 8th Int. Conf. on the Entity-Relationship Approach*, pages 309–327, Toronto, 1989.

[21] U. Hohenstein and M. Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In C. Batini, editor, *Proc. 7th Int. Conf. on the Entity-Relationship Approach*, pages 129–148. North-Holland, Amsterdam, 1988.

[22] 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.

[23] Uwe Hohenstein and Gregor Engels. Formal Semantics of an Entity-Relationship-Based Query Language. In Hannu Kangassalo, editor, *Proc. 9th Int. Conf. on Entity-Relationship Approach*, pages 129–142, 1990.

[24] Cheng Hsu, Alvaro Perry, M'hamed Bouziane, and Waiman Cheung. TSER: A Data Modeling System Using the Two-Stage Entity-Relationship Approach. In Salvatore T. March, editor, *Proc. 6th Int. Conf. on Entity-Relationship Approach*, pages 497–514. North-Holland, 1987.

[25] Scott Huffman and Randal V. Zoeller. A Rule-Based System Tool for Automated ER Model Clustering. In Frederick H. Lochovsky, editor, *Proc. 8th Int. Conf. on Entity-Relationship Approach*, pages 345–360, 1989.

[26] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

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

[28] N. Kehrer and G. Neumann. An EER Prototyping Environment and its Implementation in a Datalog Language. In G. Pernul and A.M. Tjoa, editors, *Proc. 11th Int. Conf. on Entity-Relationship Approach (ER'92)*, pages 243–261. Springer, Berlin, LNCS 645, 1992.

[29] Larry Kerschberg, Richard Baum, and Ju Hung. KORTEX: An Expert Database System Shell for a Knowledge-Based Entity-Relationship Model. In Frederick H. Lochovsky, editor, *Proc. 8th Int. Conf. on Entity-Relationship Approach*, pages 174–187, 1989.

[30] Wojtek Kozaczynski and Leslek Lilien. An Extended Entity-Relationship (E^2R) Database Specification and its Automatic Verification and Transformation into the Logical Relational Design. In Salvatore T. March, editor, *Proc. 6th Int. Conf. on Entity-Relationship Approach*, pages 533–549. North-Holland, 1987.

[31] T.-W. Ling and M.-L. Lee. A Prolog Implementation of an Entity-Relationship Based Database Management System. In T.J. Teorey, editor, *Proc. 10th Int. Conf. on Entity-Relationship Approach (ER'91)*, pages 587–606. ER Institute, Pittsburgh (CA), Participants' Proceedings, 1991.

[32] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In K.R. Dittrich and U. Dayal, editors, *Proc. of the Int. Workshop on Object-Oriented Database Systems*, pages 6–17, 1986.

[33] J.A. Makowski, V.M. Markowitz, and N. Rotics. Entity-Relationship Consistency for Relational Schemes. In G. Ausiello and P. Atzeni, editors, *Proc. Int. Conf. on Database Theory ICDT*, pages 306–322. Springer, LNCS 243, 1986.

[34] M.M.A. Morsi and S.B. Navathe. Application and System Prototyping Via an Extensible Object-Oriented Environment. In R. Elmasi and V. Kouramajian, editors, *Proc. 12th Int. Conf. on Entity-Relationship Approach (ER'93)*, pages 25–36. ER Institute, Pittsburgh (CA), Participants' Proceedings, 1993.

[35] J. Mylopoulos and H.K.T. Wong. Some Features of the TAXIS Data Model. In *Proc. 6th Int. Conf. on Very Large Data Bases*, pages 399–410, 1980.

[36] N. Nagui-Raiss. A Formal Software Specification Tool Using the Entity-Relationship Model. In P. Loucopoulos, editor, *Proc. 13th Int. Conf. on Entity-Relationship Approach (ER'94)*, pages 315–332. Springer, Berlin, LNCS 881, 1994.

[37] Christine Parent, Helene Rolin, Kokou Yetongnon, and Stefano Spaccapietra. An ER Calculus for the Entity-Relationship Complex Model. In Frederick H. Lochovsky, editor, *Proc. 8th Int. Conf. on Entity-Relationship Approach*, pages 75–98, 1989.

[38] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[39] F. Pirri and C. Pizzuti. Data Dictionary Design: A Logic Programming Approach. In G. Pernul and A.M. Tjoa, editors, *Proc. 11th Int. Conf. on Entity-Relationship Approach (ER'92)*, pages 210–225. Springer, Berlin, LNCS 645, 1992.

[40] Colette Rolland and Christophe Proix. An Expert System Approach to Information System Design. In H.-J. Kugler, editor, *Proc. IFIP 10th World Computer Congress*, pages 241–250. North Holland, 1986.

[41] Jaume Sistac. The DADES/GP Approach to Automatic Generation of Information System Prototypes from a Deductive Conceptual Model. *Information Systems*, 17(3):195–208, 1992.

[42] Christine Solnon and Michel Rueher. Using a Prolog Prototype for Designing an Object Oriented Scheme. In T.P. Clement and K.-K. Lau, editors, *Proc. International Workshop on Logic Program Synthesis and Transformation*, pages 300–317. Springer, Workshops in Computing, 1992.

[43] Scott M. Staley and David C. Anderson. Executable E-R Specifications for Database Schema Design. In King-Sun Fu, editor, *Proc. 4th Int. Conf. on Entity-Relationship Approach*, pages 160–169. IEEE, 1985.

[44] Branka Tauzovich. An Expert System for Conceptual Data Modelling. In Frederick H. Lochovsky, editor, *Proc. 8th Int. Conf. on Entity-Relationship Approach*, pages 329–344, 1989.

[45] T.J. Teorey, D. Yang, and J.P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[46] Luiz Tucherman, Marco A. Casanova, and Antonio L. Furtado. The CHRIS Consultant - A Tool for Database Design and Rapid Prototyping. *Information Systems*, 15(2):187–195, 1990.

[47] Shmuel Tyszerowicz and Amiram Yehudai. OBSERV - A Prototyping Language and Environment. *Transactions on Software Engineering*, 1(3):269–309, 1992.

[48] N. Vlachantonis, R. Herzig, M. Gogolla, G. Denker, S. Conrad, and H.-D. Ehrich. Towards Reliable Information Systems: The KorSo Approach. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Proc. 5th Int. Conf. on Advanced Information Systems Engineering (CAISE'93)*, pages 463–482. Springer, Berlin, LNCS 685, 1993.

[49] 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.