A Hybrid Query Language for an Extended Entity-Relationship Model*

(revised version of TR 93-15)

Marc Andries Gregor Engels

Leiden University, Dept. of Comp. Science
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
E-mail: {andries,engels}@wi.leidenuniv.nl

Abstract

We present the hybrid query language HQL/EER for an Extended Entity-Relationship model. As its main characteristic, this language allows a user to use *both* graphical and textual elements in the formulation of one and the same query. We demonstrate the look-and-feel of this query language by means of examples, and show how syntax and semantics of this language are formally defined using programmed graph rewriting systems. Although we present the language in the context of the EER model, the concept of hybrid languages is applicable in the context of other database models as well. We illustrate this claim by discussing a prototype implementation of a Hybrid Query Tool based on an object-oriented approach, namely the Object Modeling Technique (OMT).

1 Introduction

The database research efforts of the past decade have provided us with a wide range of both database models and systems, allowing the user to perform complex manipulations on data structures of high modeling power.

This development has strengthened the need for ad-hoc query languages [6] as well as better end user interfaces, fully exploiting the two-dimensional nature of computer screens. In the late eighties, the observation that object schemes and instances allow for a natural graphical representation, inspired a number of researchers to develop *graph oriented* database models, in which notions from graph theory are used to uniformly define not only the data representation or scheme part of the model, but also its data manipulation and query language(s) [5, 10, 14, 20, 23, 28, 35] (in the sequel, we use the term *operations* to denote both queries and data manipulations). In these models, it is investigated to what extent operations may be expressed in a *purely* graphical way. One may conclude from this research that the expressive power that may be obtained with pure graph based languages is unlimited [4].

However, one also gets the impression that some of this research overshoots its mark in the sense that the pure graphical formulation of an operation quite often looks even more complex than its textual equivalent. The solution to this problem which we present in this article, is to combine the "best of both worlds", that is, to develop *hybrid* languages that allow those parts of an operation that are most clearly specified graphically respectively textually, to be indeed specified graphically respectively textually.[1] In a sense, Zloof's Query-By-Example [36], which is commonly considered to be one of the first attempts at "two-dimensional" query languages,

*Work supported by COMPUGRAPH II, ESPRIT BRWG 7183.

[1] The term "hybrid" was inspired by hybrid syntax-directed editors, where the user can freely choose between a syntax-directed and a free style of editing [16].

already offers facilities along this line. Indeed, while join conditions and selections are entered "graphically" (that is, in table skeletons), complex conditions involving e.g., aggregate functions, should be entered in plain text in a so-called "condition box". Similar facilities are offered in prototype interfaces for semantic database models, like SNAP's "node restriction" [7]. In more recent proposals, like [25, 26], tools are presented which give the user a (limited) choice between graphical and textual specification of operations, limited in the sense that graphics and text may not be *mixed within the same* operation.

A major design decision made in the definition of the hybrid language presented in this article is to consciously restrict the set of language constructs (such as declarations, atomic formulas,...) which may be represented graphically to the set of symbols used for graphically representing schemes. Consequently, concepts such as negation and aggregate functions cannot be incorporated into the graphical part of a hybrid operation. As demonstrated in among others [5, 10, 23, 35], it is perfectly possible to invent a graphical representation for these additional concepts. However, it is our conviction that it is precisely the abundance of graphical primitives in many of these fully graphical languages, which makes expressions in them often harder to understand than their textual counterpart. This very observation led us to considering the notion of hybrid languages.

In summary, it is our aim in this article to introduce a *hybrid query language*, in which queries are expressed by means of a mixture of graphical and textual elements. As a framework for presenting this language, we use an extended version of the Entity Relationship model [15] (named "EER"-model in the sequel), which we choose basically for two reasons. First, EER schemes allow for a natural graphical representation of schemes. A second reason was the availability of a formally defined and highly expressive SQL-like query language, named SQL/EER [22]. This language was inspired by some proposals made for query languages for the Entity Relationship model [8, 13], as well as proposals to extend SQL to cope with features of other database models than the relational one [17, 29]. The Hybrid Query Language for the Extended Entity Relationship model (called HQL/EER in the sequel) is an extension of SQL/EER with graphical alternatives for some of its language constructs, where, as already mentioned, the set of these graphical alternatives is a subset of those employed in the graphical representation of EER schemes.

We stress the fact that we *extended* SQL/EER with *alternatives*, rather than *replacing* textual by graphical constructs. As a major consequence, the user of HQL/EER may freely *choose* between textual and graphical expression of those language constructs for which graphical alternatives are offered.

The graphical part of HQL/EER is formally defined by means of a specification in the language PROGRES, a very high level operational specification language based on the concepts of *programmed attributed graph rewriting systems* [32]. The specification defines both

- the syntax of the graphical part of HQL/EER, that is, the set of graphs that correspond to "valid" queries; and
- the semantics of the graphical part of HQL/EER, by translating it into a (possibly incomplete) SQL/EER statement.

Consequently, this definition is quite analogous to that of SQL/EER itself. Indeed, both syntax and semantics of SQL/EER are defined by means of an attributed string grammar [22]. In the attribute part of this grammar, SQL/EER queries are translated into expressions of a formally defined calculus for the EER model [19].

The remainder of this article is organized as follows. In Sections 2 and 3, we repeat the basic concepts of respectively the Extended Entity Relationship model and its query language SQL/EER. After an informal introduction of HQL/EER by means of examples in Section 4, we show in Section 5 how the language is formally defined in PROGRES, a short overview of which may be found in Appendix A. Finally, in Section 6, we discuss a prototype Hybrid Query Tool currently under development. The fact that in this tool the (object model of) Object Modeling Technique [30] is used in replacement of the EER model, illustrates how the concept of hybrid languages is also applicable in the context of languages for other (object-oriented) database models.

2 The Extended Entity-Relationship Model

In this Section, we sketch briefly the main concepts of the Extended Entity-Relationship (EER) model [15][2]. It is based upon the classical Entity-Relationship model [9] and extended with the following concepts known from semantic data models [24]:

- components, i.e., object-valued attributes to model complex structured entity types:
- multivalued attributes and components to model association types;
- the concept of type construction in order to support specialization and generalization;
- several structural restrictions like the specification of keys, cardinality constraints,..., which are, however, of no interest to this article.

Let us illustrate the EER model and its features by means of a small example. The EER diagram of Figure 1 models the world of surfing people who surf on different kinds of waters.

First of all, one easily recognizes the basic concepts of the ER model. These are *entity types* like `PERSON`, *relationship types* like `surfs_on_river`, and *attributes* like `Name` (of `PERSON`) or `Times/Year` (of `surfs_on_river`).

The concept of *type construction* provides means to construct new entity types (called *output types*) from already existing entity types (called *input types*). This means that each object in the set of instances of a constructed entity type also belongs to the set of instances of the input types. Type constructions are represented by triangles, where all input types are connected by edges with the baseline, and the output types with the opposite point. For instance, the type construction `spec1` represents the special case of a specialization. It has one input type `PERSON` and one output type `SURFER`, i.e. `PERSON` is specialized to `SURFER`. Then, `SURFER` in turn is specialized to `PRO`(fessional), this time by `spec2`. This means that each `PRO` is a surfer and therefore also a person.

A type construction is called *specialization*, if it has only one input entity type and one or more output entity types. Another example is `WATER`, which is specialized into `LAKE` and `RIVER`. In the case of specialization, all attributes are inherited from the input types to the output types. For instance, each instance of type `LAKE` also has the attribute `Name`, defined for the entity type `WATER`. Obviously, we do not allow a constructed entity type to be, directly or indirectly, input type of its own type construction.

Generally, an instance of the input type(s) of a type construction is not necessarily a member of the instance set of (one of) the output type(s). For instance, there could be surfers who are not pros. If a total partition of the input instances is desired, the type construction triangle in the diagram is labeled with '=' instead of '$\supseteq$'. For example, each instance of type `WATER` must be an instance of `LAKE` or `RIVER`, but nothing else.

Complex structured entity types can be modeled by *components*. Roughly speaking, components can be seen as object-valued attributes. For instance, each pro possesses one or more surf-boards. Hence `Boards` is a component of `PRO`, which consists of a list of instances of type `SURF-BOARD`. Both attributes and components can be multivalued, i.e., set-, bag-, or list-valued. Multivalued attributes and components are represented by an oval, including a square into the oval that is connected to the corresponding entity type or atomic value type via an arrow, which is always labeled $\in$.

Single-valued components are also represented by means of an oval, labeled "singl.". As an example, note that both the entity type `PRO` and its direct ancestor in the construction hierarchy (i.e., the entity type `SURFER`) have a `Boards`-component, but with different types. Since an ordinary surfer can only possess one surf-board, the `Boards`-attribute for the entity type `SURFER` is singlevalued. Since professional surfers can own several surf-boards, the `Boards`-attribute for

[2]Not to be confused with other extensions to Chen's original ER model, such as the Enhanced Entity Relationship model discussed in [12].

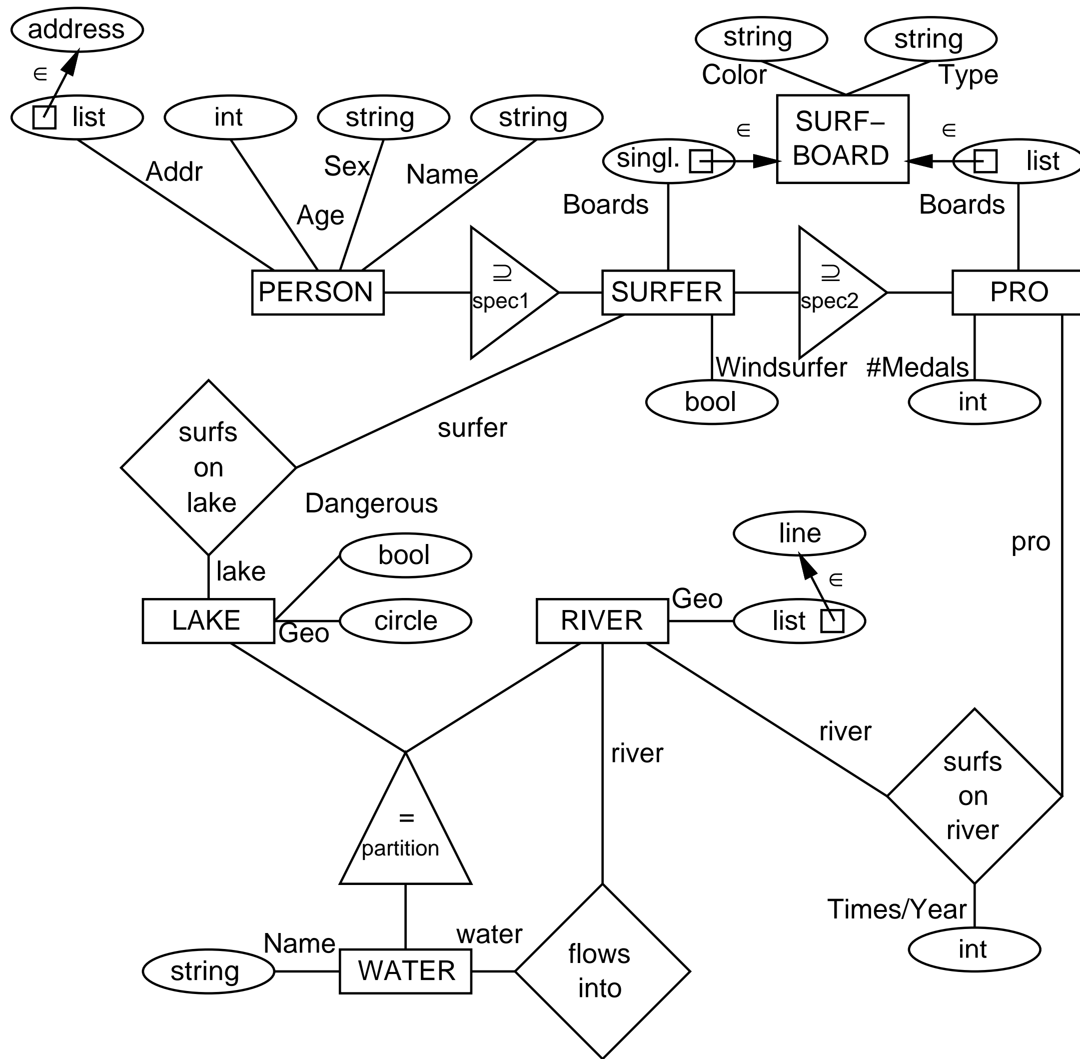


Figure 1: An EER diagram, modeling the world of surfers

the entity type `PRO` is a *list* of surf-boards. This way, the known concept of *overriding* (both of attributes and components) is incorporated in the EER model.

As a final note, we would like to stress that we use the EER model (which was motivated and discussed at length in several foregoing publications, e.g., in [15]) merely as a vehicle to introduce and discuss the notion of hybrid database languages.

3 SQL for the Extended Entity-Relationship Model

Based on the data model discussed in Section 2, we now informally repeat the main concepts of the textual query language SQL/EER. A complete description of syntax and semantics of this language can be found in [22].

SQL/EER directly supports all the concepts of the EER model, and takes into account well known features that nowadays are an integral part of many query languages:

1. relationships, attributes of relationships, components and type constructions;
2. arithmetic;
3. aggregate functions;
4. nesting of the output;
5. subqueries as variable domains.

Analogous to relational SQL, SQL/EER uses the **select-from-where** clause.

As a first example, consider the SQL/EER query of Figure 2 (over the scheme of Figure 1). It retrieves the name and age of all adults, i.e., persons older than 18.

select p.Name, p.Age
from p **in** PERSON
where p.Age ≥ 18

Figure 2: Name and age of adults (SQL/EER version)

In this query, the variable p is declared. It ranges over the set of currently stored persons. The variable p is used to build terms like p.Name and p.Age, to compute the name and age of the person p, respectively. The formula "p.Age $\geq$ 18" uses the predicate "$\geq$", defined for the integer data type.

Besides entity types and relationship types, any multi-valued term can also be used as range in a declaration. For instance, in the SQL/EER query of Figure 3, the variable a1 is bound to the finite list of addresses of person p1.

select p1.Name
from a1 **in** p1.Addr, p1 **in** PERSON, a2 **in** p2.Addr, p2 **in** PERSON
where a1 = a2 **and** p2.Name = `'John'`

Figure 3: Name of persons sharing an address with John (SQL/EER version)

This query retrieves the names of all persons who share an address with a person called 'John'. Note that the result of an SQL/EER is a multiset. This means that the same name may appear

several times in the answer of this query. By placing the reserved word **distinct** in front of the term list in the **select**-clause, a set of distinct names is computed. Note also that since the domain of the Addr-attribute contains complex values (namely lists of addresses, where an address is a user-defined atomic value type), the equality predicate in the query stands for value equality.

The next example shows the use of inheritance and the use of relationship types as predicates in SQL/EER. Suppose we want to know the names of those professional surfers who surf on rivers that flow into lakes on which they also surf. Figure 4 shows the corresponding SQL/EER query.

```
select  p.Name
from    r in RIVER, l in LAKE, p in PRO
where   p surfs_on_lake l  and p surfs_on_river r  and r flows_into l
```

Figure 4: Name of pros, surfing on rivers, flowing into lakes they surf on (SQL/EER version)

Here, the variable p is declared of type `PRO`. As pros are "specialized" persons, the attribute `Name` is also defined for them. Thus, p.Name is a correct term. Furthermore, relationship types can be used as predicate names in formulas. In the case of relationships with more than two participating entity types, prefix notation is used instead of infix.

Participation in a relationship is inherited, too. Therefore, a variable of type `LAKE` (like l) is allowed as participant in relationship `flows_into` within the (sub-)formula "r `flows_into` l".

A final example illustrates the use of subqueries. Suppose we want to retrieve for each type of surf-board recorded in the database, the bag of colors in which this type of surf-board is available. This retrieval may be expressed by means of the SQL/EER-query depicted in Figure 5.

```
select  type, ( select  b.Color
                from    b in SURF–BOARD
                where   b.Type = type )
from    type  in ( select  sb.Type
                   from    sb in  SURF–BOARD )
```

Figure 5: For each type of surf-board recorded in the database, the bag of colors in which this surf-board is available (SQL/EER version)

This example illustrates how a subquery may be used in the **select**-clause of an SQL/EER-query to obtain "structured" output. The output of this query consists of a bag of pairs, each pair consisting of a string and a bag of strings. Note how the variable `type` may be used in the subquery in the **select**-clause, since this subquery lies within the scope of the declaration of `type` in the outermost query. At the same time this example illustrates how a subquery may be used in the **from**-clause, namely as variable domain.

4 Specification of Hybrid Queries

In the previous section, we discussed a fully textual query language for the EER model. In this section, we show by means of examples how this language can be extended with *graphical alternatives* for (some of) its language constructs. Since these graphical alternatives do not replace their textual counterparts, but are really offered as alternatives, we obtain a *hybrid* query language. The language resulting from this extension is therefore called the hybrid query language HQL/EER.

Briefly, a query in HQL/EER consists of an attributed labeled graph and/or a piece of text (obeying the syntax of SQL/EER). As it is the case with the textual part, the graph generally consists of declarations (as in the **from**-clause of the text), conditions (as in the **where**-clause of the text), as well as selections (as in the **select**-clause of the text). As mentioned in the Introduction, for expressing declarations and conditions graphically, we restrict ourselves to the set of graphical symbols used for representing EER-schemes. For instance, variables for a river and a water may be declared by drawing two (rectangular) nodes labeled respectively `RIVER` and `WATER`.

The structural constraints applying to the construction of EER schemes, apply to the graphical part of hybrid queries as well. For instance, the condition that we are only interested in pairs of a river and a water such that the river flows into the water, is indicated by drawing a (diamond shaped) node labeled `flows_into` with (appropriately labeled) edges to both other nodes (see Figure 6).

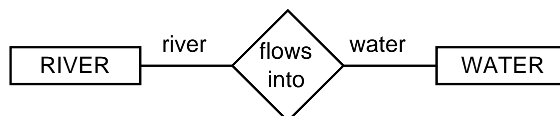


Figure 6: A sample graphical part of an HQL/EER query

The remainder of this Section consists of a number of examples of hybrid queries over the scheme of Figure 1. Figure 7 shows a possible expression of the SQL/EER query of Figure 2 in HQL/EER. It consists of both a graphical and a textual part. Intuitively, the `PERSON`-node corresponds to the declaration of the variable p in the SQL/EER version, while e.g., the `int`-node corresponds to the term p.Age in the SQL/EER version, since it is linked to the node corresponding to the variable p by means of an edge labeled `Age`.

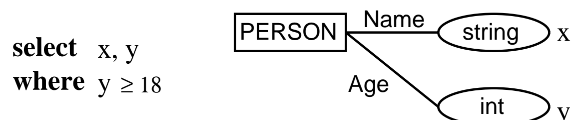


Figure 7: Name and age of adults (HQL/EER version)

Note that in the textual part of the hybrid query, the two variables x and y are used but not declared. Instead, they refer to nodes in the graph. Hence the variable x ranges over the names of all persons, while y ranges over their ages. This is the way in which the textual and graphical part of a hybrid query are interrelated. The textual part specifies that the values assigned to x and y are retrieved, if and only if the value for y, i.e., the person's age, is larger than 18.

In the following sections, we formalize this correspondence between declarations, terms and formulas in a textual expression on one hand, and subgraphs of a graphical expression on the other hand.

In the example of Figure 7, the graphical part of the HQL/EER query consists simply of a subgraph of the graphical representation of the scheme. The graphical part of the following example is a more general graph, which however still satisfies the structural constraints imposed by the scheme.

Figure 8 shows a way of expressing the SQL/EER query of Figure 3 in HQL/EER. The fact that a *single* `address`-node is linked to the `Addr`-attributes of *both* `PERSON`-nodes indicates that we are interested in people *sharing* an address.

The graphical part of this query contains two illustrations of constructs whose graphical expression may be considered more natural than their textual counterpart. The aforementioned sharing of the `address`-node as opposed to the join predicate "a1 = a2" in the SQL/EER query, is one example. Second, the graphical arrangement of the various nodes shows the interconnection of the persons and their respective address lists in a more straightforward manner than the declarations in the SQL/EER version of this query.

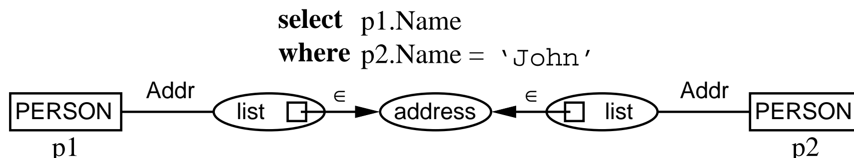


Figure 8: Name of persons sharing an address with John (HQL/EER version I)

With another version of the SQL/EER query of Figure 3, we illustrate hybrid queries consisting *merely* of a graph (see Figure 9). The fact that the `Name`-attribute of one of the persons should have the string value 'John' is indicated by adding this value under the corresponding node. The shading of the other `string`-node indicates the information to be retrieved, i.e., the names of the persons who share an address with John.

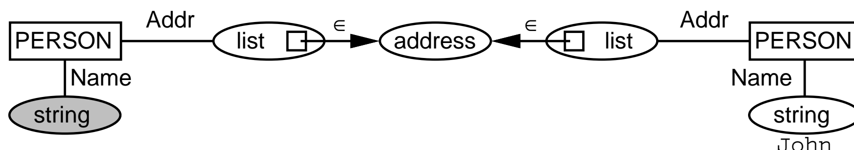


Figure 9: Name of persons sharing an address with John (HQL/EER version II)

Figure 10 shows another example of an entirely graphical specification of a query, namely that of Figure 4. Note that an entity of type `LAKE` plays the role of a water in the relationship `flows_into`, illustrating how inheritance is used in HQL/EER, in a manner similar to SQL/EER. Since `LAKE` inherits the participation in the `flows_into` relationship from `WATER`, the graphical part of this query is still considered to satisfy the structural constraints imposed by the scheme (analogously for `PRO` and `surfs_on_lake` and `SURFER`).

Note also how each of the graph increments consisting of a diamond node and the two rectangles it is connected to, corresponds to a conjunct in the **where**-clause of the textual expression.

We continue this collection of examples with a slightly more involved one, illustrating our claim that HQL/EER allows those parts of a query that are most clearly specified graphically respectively textually, to be indeed specified graphically respectively textually. The hybrid query of Figure 11 retrieves the address lists of surfers, who have a relative (i.e., a person with the same name)

- being a professional windsurfer;
- owning a board of the same type as the (single) board owned by the surfer;
- not surfing on dangerous lakes.

Since it was our design decision to restrict the set of query-elements (such as declarations, atomic formulas,...) which may be represented graphically to the set of symbols used for graphically

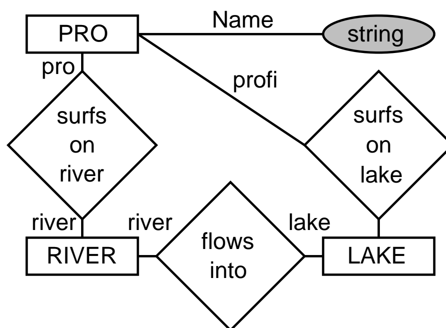
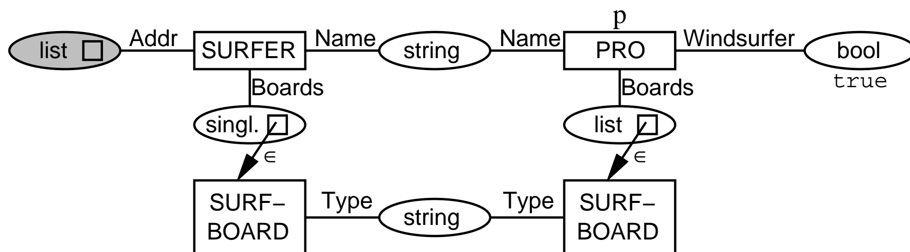


Figure 10: Names of pros, surfing on rivers, flowing into lakes they surf on (HQL/EER version)

representing schemes, we cannot express negation graphically. Hence we have to leave the part of the query involving negation in the textual part, and express everything else graphically.



where **not exists** l **in** LAKE : (l.Dangerous **and** p surfs_on_lake l)

Figure 11: "Involved" hybrid query

In contrast, Figure 12 shows an SQL/EER version of the same query. Note how in this textual query, related information is again dispersed over e.g., the declarations in the **from**-clause and the join-predicates in the **where**-clause. Note by the way that this SQL/EER query is itself an HQL/EER query with an empty graphical part!

```
select  s.Addr
from    s in SURFER, pb in p.Boards, p in PRO
where   not exists l in LAKE : ( l.Dangerous and p surfs_on_lake l )
        and p.Name = s.Name and p.Windsurfer
        and s.Boards.Type = pb.Type
```

Figure 12: "Involved" textual query

We conclude this Section with an example of a hybrid query involving subqueries. Figure 14 presents a totally graphical version of the query whose formulation in SQL/EER is depicted in Figure 5. In recapitulation, this query (shown once more in Figure 13) retrieves, for each type of surf-board recorded in the database, the bag of surf-boards that have this type.

```
select  type, ( select  b.Color
                from    b in SURF–BOARD
                where   b.Type = type )
from    type  in ( select  sb.Type
                   from    sb in  SURF–BOARD )
```

Figure 13: For each type of surf-board recorded in the database, the bag of surf-boards that have this type (SQL/EER version)

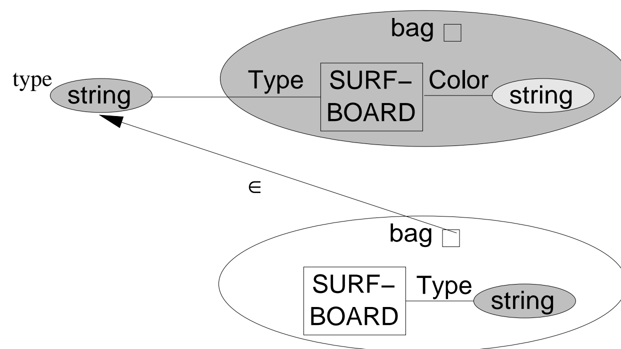


Figure 14: For each type of surf-board recorded in the database, the bag of surf-boards that have this type (HQL/EER version)

Although Figure 14 at first glance does not look much like an EER diagram, a closer look reveals that it is still composed of nothing but graphical primitives also present in EER diagrams. The basic idea behind the graphical representation of subqueries is that, since subqueries return a bag (of either entities or values), we may as well use the graphical convention used in EER-diagrams for depicting a bag, namely an oval with `bag` inscribed in it. In addition, we put the graphical equivalent of the subquery inside the oval.

Let us now have a look at how these ideas are incorporated into the example query of Figure 14. The large oval in the bottom right corner of the picture corresponds to the subquery in the **from**-clause of the SQL/EER query. Indeed, the query depicted inside this oval selects the Type-attribute of every surf-board in the database. The $\in$-labeled edge denotes the fact that the string depicted in the top left corner of the picture (which, as indicated superficially in the picture, corresponds to the variable `type` in the SQL/EER-query) ranges over this subquery.

The large (shaded) oval in the top right corner of the picture corresponds to the subquery in the **select**-clause of the SQL/EER-query. The lighter shading of the node labeled `string` indicates that this subquery selects a bag of strings. The `Type`-labeled edge connecting the node labeled `SURF-BOARD` (which corresponds to the variable b in the SQL/EER query) to the string depicted in the top left corner of the picture, expresses precisely the condition "`b.Type = type`" in the SQL/EER-query.

5 Definition of the Hybrid Query Language

In Section 4, we introduced the ideas and concepts behind HQL/EER by means of examples. In this section, we outline the formal definition of syntax and semantics of HQL/EER. A full definition of HQL/EER may be found in [2].

Formalization of HQL/EER involves the *representation* of the graphical part of hybrid queries as *labeled, attributed graphs*. Such a graph represents the (abstract) syntactical structure of a query. Node labels correspond to scheme elements, like entity type names. Node attributes[3] are used for a double purpose, namely the storage of

1. non-structural information which is part of the query, such as atomic values, and
2. (SQL-)declarations, formulas and terms corresponding to nodes.

This latter information is combined with the textual part of the hybrid query into a complete SQL/EER-query, whose semantics is defined to be the semantics of the hybrid query itself.

As an example, consider Figure 15, which shows the graph corresponding to the graphical part of the hybrid query of Figure 8. In informal terms, Figure 8 corresponds to what a user would see on the screen of a tool supporting HQL/EER, while Figure 15 corresponds to the internal representation of (the graphical part of) the query.

Within each node, its (unique) node identifier and its node label are depicted. Table 1 shows the attributes of some of the nodes in the graph depicted in Figure 15. The precise meaning of the different labels and attributes is explained in the remainder of this Section.

Our need for an expressive graph model motivates our choice of the graph rewriting formalism PROGRES [31]. PROGRES is a very high level operational specification language based on **PRO**grammed **G**raph **RE**writing **S**ystems. A PROGRES specification consists of two components:

1. a *graph scheme*, declaring the types of nodes and edges, as well as node attributes, which may occur in a graph, and

[3] Note that the word "attribute" is now used in two different meanings : one in the context of graphs, and the other in the context of the EER-model.

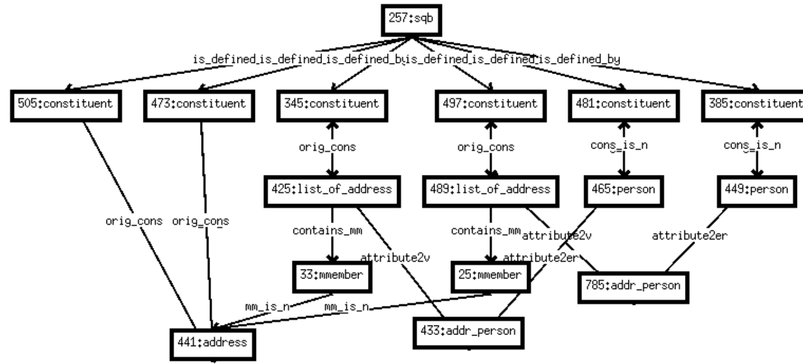


Figure 15: Graph representation of the hybrid query of Figure 8

Id.	Label	Att.Name	Attribute Value
257	sqb	SFW-Term	select from a1 in p1.addr_person, a2 in p2.addr_person, p1 in person, p2 in person where a2 = a1
465	person	Declaration Term	p2 in person p2
425	list_of_address	Term	p2.addr_person
441	address	Declaration Term Formula	a1 in p1.addr_person, a2 in p2.addr_person a1 a2 = a1
473	constituent	Declaration Term Formula	a1 in p1.addr_person a1 a2 = a1
489	list_of_address	Term	p1.addr_person
449	person	Declaration Term	p1 in person p1

Table 1: Attributes of some nodes from Figure 15

2. a set of graph rewrite rules or *productions* (obeying the type restrictions imposed by the graph scheme) which specify the connection between the components defined in the graph scheme.

Such an operational specification defines a graph language, in which a syntactically correct graph is yielded by applying a sequence of productions to an initial empty graph. A more elaborate description of PROGRES, based on a small example specification, is given in Appendix A.

A major motivation for choosing the PROGRES formalism was that the entire specification of the graphical part of HQL/EER could consequently be made using the PROGRES-*system* [27]. Concretely, the specification was entered using this system's syntax-directed editor [33], which allowed the specification to be analyzed by the system's incrementally working type-checker and executed by the system's integrated interpreter. The graph shown in Figure 15 is the result of such an interpretation (that is, the *execution* of a sequence of productions), and was generated using the PROGRES system.

Formalizing the graphical part of HQL/EER queries by means of a PROGRES specification is done by means of a two step process. In a first step (see Section 5.1), the *syntactic* structure of the graphical part of HQL/EER queries is captured in a PROGRES specification. This specification is extended in a second step (see Section 5.2) to define also the *semantics*. This is done by extending the specification resulting from the first step with additional node attributes and attribute derivation rules. These rules translate the graphical part of the HQL/EER query into a (possibly incomplete) SQL/EER query, which in turn may be combined with the textual part of the hybrid query into a full SQL/EER-query, defining the semantics of the original hybrid query.

5.1 The Syntax of HQL/EER

To start the syntax definition of HQL/EER, we formalize (part of) the graphical notation introduced in Section 4 in terms of a PROGRES graph scheme. Such a graph scheme consists of two parts:

1. an EER scheme *independent* part, which for instance expresses the fact that there are nodes for representing entities and nodes for representing values.

2. an EER scheme *dependent* part extending the EER scheme independent part, which among others expresses the fact that there are entities of type `SURFER`, and values of type `address`.

Figure 16 shows part of this scheme.

The node class `NODE` is the root-class of (most of) the node class hierarchy. The class `VALUE` is a direct subclass of `NODE`, and has (among others) the classes `ATOMIC_VALUE` and `COMPLEX_VALUE` as direct descendants. Nodes of class `ATOMIC VALUE` have an attribute `Value` in which the "actual value" is stored. For simplicity, we assume all values are stored as strings. The only subclass of `COMPLEX_VALUE` shown in Figure 16 is `MVALUE`, which stands for "Multi-VALUE". This name is in turn explained by its two subclasses, namely `LIST_VALUE` and `BAG_VALUE`: one and the same value or entity may be an element of a list or bag than once.

Figure 17 exemplifies the usage of the classes `ENTITY`, `ATTRIBUTE` and `LIST_VALUE`, as well as of the edge types `attribute2er` and `attribute2v`. This figure shows the graph-representation of a list-valued attribute of an entity.

Figure 18 exemplifies the usage of the classes `MVALUE` and `MMEMBER` (which stands for "Multi-MEMBER"), as well as of the edge types `contains_mm` and `mm_is_n` (which stands for "mmember_is_node"). This figure shows the graph representation of membership of a multi-value. Nodes of type `MMEMBER` are necessary for representing multiple membership in one and the same multi-value, since (in the PROGRES formalism) it is impossible to draw multiple edges between different nodes.

The class `SQB` stands for (Sub)QueryBag, and has a single type `sqb`. Nodes of type `sqb` represent a query or subquery. `SQB` is a subclass of `BAG_VALUE`, since (the result of) a (sub)query

```
section FixedGraphScheme
   node class NODE end;
   node class PART_OF_COMPLEX is a NODE end;
   node class ENT_REL is a NODE end;
   node class ENTITY is a ENT_REL, PART_OF_COMPLEX end;
   node class VALUE is a NODE end;
   node class VALUE_PART_OF_COMPLEX is a VALUE, PART_OF_COMPLEX end;
   node class ATOMIC_VALUE is a VALUE, VALUE_PART_OF_COMPLEX
      intrinsic
         Value : string := "";
   end;
   node class COMPLEX_VALUE is a VALUE
      derived
         Elem_Type : type in NODE;
   end;
   node class MVALUE is a COMPLEX_VALUE end;
   node class BAG_VALUE is a MVALUE end;
   node class SQB is a BAG_VALUE, VALUE_PART_OF_COMPLEX end;
   node type sqb : SQB
    redef derived
     Elem_Type =
    ((self.=oPartNode=>:CONSTITUENT[1:1]).-cons_is_n->:NODE[1:1]).type;
   end;
   node class LIST_VALUE is a MVALUE end;
   node class MMEMBER is a NODE
      intrinsic
         Index : integer := 0;
   end;
   node type mmember : MMEMBER end;
   edge type contains_mm : MVALUE -> MMEMBER;
   edge type mm_is_n : MMEMBER -> NODE [1:1];
   node class CONSTITUENT is a NODE
      intrinsic
         Output : boolean := false;
   end;
   node type constituent : CONSTITUENT end;
   edge type is_defined_by : SQB -> CONSTITUENT;
   edge type cons_is_n : CONSTITUENT -> NODE [1:1];
   edge type orig_cons : NODE -> CONSTITUENT;
   node class ATTRIBUTE end;
   edge type attribute2er : ATTRIBUTE -> ENT_REL [1:1];
   edge type attribute2v : ATTRIBUTE -> VALUE [1:1];
end;
```

Figure 16: EER scheme independent part of the PROGRES graph scheme for HQL/EER

Figure 17: Graph-representation of a list-valued attribute of an entity

Figure 18: Graph-representation of membership of a multi-value

is itself a bag. Type constituent is the single type of class CONSTITUENT. Nodes of this type (together with edges of type is_defined_by and cons_is_n) are used to link a node of type sqb to a node that constitutes part of the definition of the considered (sub)querybag. For instance, if a certain query involves a certain entity, then the graph representation contains the graph depicted in Figure 19. The Output-attribute is set to true if the node reachable by means of the outgoing cons_is_n-edge is selected in the query corresponding to the node reachable by means of the outgoing is_defined_by. Since nodes may be constituents of more than one (sub)querybag (in case of subqueries), selection for output has to be indicated on the constituent-nodes, rather than on the nodes themselves.

Figure 19: Graph-representation of constituents of a (sub)querybag

In Section 5.2, it is shown how in the attributes of nodes of type sqb, information (that is, declarations, terms and formulas) is "collected" from all over the graph, which is combined with the textual part of the hybrid query into a fully textual query.

Classes depicted in Figure 16 but not mentioned in the explanation above, are either introduced to ensure that the inheritance-hierarchy is a lattice (like the class PART_OF_COMPLEX), or as coercions of node classes which allow for a more concise expression of other parts of the specification (like the class ENT_REL, which is a common superclass of the classes ENTITY and RELSHIP, which share several common properties).

As announced previously, the second part of the formalization of HQL/EER in terms of a PROGRES specification, consists of an extension of the EER scheme independent part of the graph scheme outlined above, with an EER scheme dependent part. Figure 20 shows part of the PROGRES graph scheme corresponding to the EER scheme depicted in Figure 1.

The extra node classes are introduced to cope with inheritance relationships between entity types, present in the EER scheme. On one hand it is not possible to specify inheritance relationships between node types, so we have to use a class for each entity type in the EER scheme. On the other hand, actual nodes have to belong to a type, so for each class we have to declare a type of each of these classes. Note also how attributes and roles are uniformly modeled using node types. The meaning of the attribute Elem_Type will be explained when we discuss productions.

The collection of declarations in the graph scheme described above, is in itself insufficient to completely describe the syntax of the graphical part of hybrid queries. Indeed, the part of the specification given so far, for instance, does not enforce that any node representing an entity, a relationship or a value, should be connected to a node representing a (sub)query. Hence we still need to specify which configurations of nodes[4] and edges are allowed. As PROGRES is an *operational* specification language, this is done by means of a set of productions[5]. Correct HQL/EER graphs are those graphs that may be obtained as a result of the application of any sequence of these productions to an initial empty graph.

The complete specification includes about twenty productions, three of which are discussed below.

First reconsider Figure 15, depicting the graph representation of the hybrid query of Figure 9. Construction of this (and of any other) graph starts with the creation of an sqb-labeled node, representing the query. Next, we need two nodes representing entities (persons, in the case of the

[4] Other than those enforced by the edge type declarations

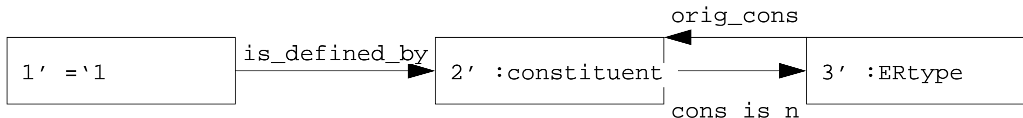
[5] Note that this does not imply that the user of HQL/EER should look upon the *execution* of a query as graph rewriting on the database (as opposed to e.g., the formalism discussed in [3]). In HQL/EER, graph rewriting is only used for the *definition* of the query language.

```
section VariableGraphScheme
   section NodeClasses
      node class PERSON is a ENTITY end;
      node class SURFER is a PERSON end;
      node class PRO is a SURFER end;
   end;
   section NodeTypes
      node type person : PERSON end;
      node type surfer : SURFER end;
      node type pro : PRO end;
      node type surfs_on_lake : RELSHIP end;
      node type surfer_surfs_on_lake : ROLE end;
      node type address : ATOMIC_VALUE end;
      node type list_of_address : LIST_VALUE
         redef derived
            Elem_Type = address;
      end;
      node type text : ATOMIC_VALUE end;
      node type int : ATOMIC_VALUE end;
      node type bool : ATOMIC_VALUE end;
      node type name_person : ATTRIBUTE end;
      node type addr_person : ATTRIBUTE end;
   end;
end;
```

Figure 20: EER scheme dependent part of the PROGRES graph scheme for HQL/EER

example), which are created using the production `Add_ER_labeled_node`. This production adds a new entity (or relationship) to a given (sub)querybag.[6]

```
production Add_ER_labeled_node
      ( s : SQB ; VarName : string ; ERtype : type in ENT_REL ;
        out E : ENT_REL ; out C : CONSTITUENT )
 =

    `1 = s

   ::=
```



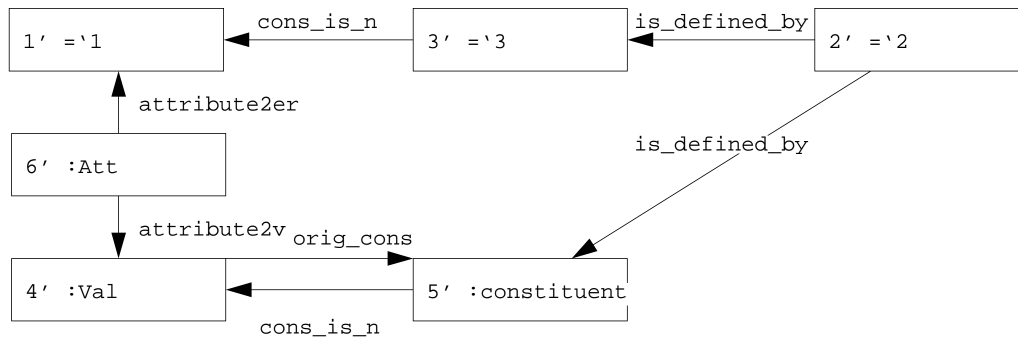

```
   return E := 3';
          C := 2';
end;
```

Construction of the graph of Figure 15 continues with the addition of certain attributes (in the EER sense of the word "attribute") like name and address to the newly created entities. This is done using the production `Add_Attribute`.

[6]Note that in the graph depicted in Figure 15, multiple edges between two nodes (like `orig_mm` and `m_is_n`) appear as one edge with an arrowhead on both sides.

```
production Add_Attribute
   ( Er : ENT_REL ; s : SQB ; Att : type in ATTRIBUTE ;
     Val : type in VALUE ; out v : VALUE ; out C : CONSTITUENT )
   =
```

```
   ::=
```

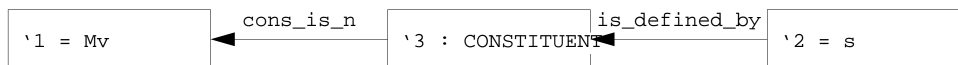



```
   return v := 4';
          C := 5';
end;
```

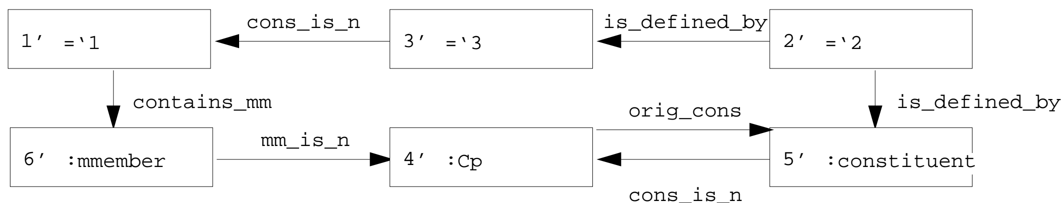
This production adds an attribute to an entity (or relationship). Note how the newly created value-node (with node identifier 4') is linked to both the entity as well as the (sub)querybag.

As a final example of a production, we discuss the production `Add_to_MValue`, which puts a value or entity into a multi-value (in the case of the example, to put an address into a list of addresses). Note how the newly created `PART_OF_COMPLEX`-node (with node identifier 4') is linked to both the complex value as well as the (sub)querybag. In the `condition`-clause, it is checked whether the given value or entity has the correct type, using the attribute `Elem_Type`.

```
production Add_to_Mvalue
   ( Mv : MVALUE ; s : SQB ; Cp : type in PART_OF_COMPLEX ; VarName : string ;
     out c : PART_OF_COMPLEX ; out C : CONSTITUENT ; out M : MMEMBER )
      =
```




```
         ::=
```




```
   condition `1.Elem_Type = Cp;
   return c := 4';
          C := 5';
          M := 6';
end;
```

To conclude this Section, note that if a given EER scheme is mapped as exemplified in Figure 20 to an extension of the PROGRES graph scheme as outlined above, it is guaranteed that any application of a PROGRES production results in a graph that obeys the structural constraints imposed by this scheme. For instance, the production for creating entities may be used to create a node of type `person`, since this type is declared (indirectly) of class ENTITY. Analogously, the production for the addition of attributes may be used to link a new node of type `text` to an existing node of type `person` by means of a node of type `name_person`. The "instantiated" production for specifying the name of a person is shown in Figure 21.[7]

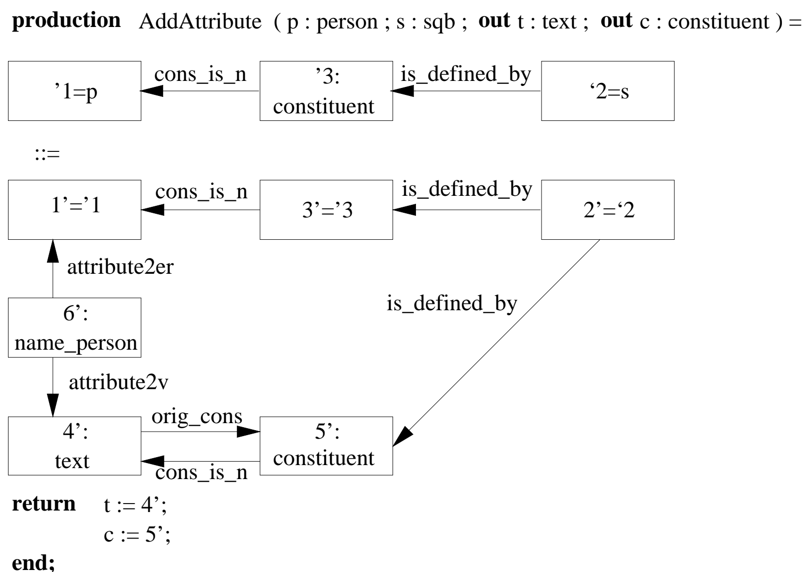


Figure 21: An instantiation of a PROGRES production

5.2 The Semantics of HQL/EER

We now define the semantics of HQL/EER queries in terms of the (formally defined) semantics of SQL/EER [22]. As announced previously, we therefore first of all extend both the declarations of the graph scheme and the productions by attribute derivation rules,[8] which translate the graphical part of the query into a (possibly incomplete) SQL/EER-query. Secondly, this SQL/EER-query is combined with the textual part of the hybrid query into a full SQL/EER-query, defining the semantics of the hybrid query.

The graph scheme depicted in Figure 22 includes the auxiliary attribute declarations and derivation rules needed for defining the semantics of hybrid queries. In the specification of the node class `NODE`, two attributes are declared which are to be used for storing the term corresponding to a node. If this term is computed in the production that creates this node, then the (intrinsic) attribute `Term` is used. If the term is computed "automatically" by means of a derivation rule, the (derived) attribute `SFW_Term` is used. Likewise, the attributes (`SQB_`)`Formula` and `Declaration` contain formulas and declarations corresponding to a certain node.

Probably the single most important sentence in the entire specification is the derivation rule for the attribute `SFW_Term` as given in the declaration of the node type `sqb`. In this rule, information gathered from all over the graph is combined into a (possibly incomplete, in case of a hybrid query

[7] Note that such an instantiated production is just an intuitive notion, and has no formal meaning.

[8] We refer once more to Appendix A for an explanation of the usage of attributes in PROGRES specifications.

```
section FixedGraphScheme
   node class NODE
      intrinsic
         Formula : string := "true";
         Declaration : string := "";
         Term : string := "";
      derived
         SFW_Term : string = "";
         SQB_Formula : string = "";
   end;
   node class PART_OF_COMPLEX is a NODE end;
   node class ENT_REL is a NODE end;
   node class ENTITY is a ENT_REL, PART_OF_COMPLEX end;
   node class VALUE is a NODE end;
   node class VALUE_PART_OF_COMPLEX is a VALUE, PART_OF_COMPLEX end;
   node class ATOMIC_VALUE is a VALUE, VALUE_PART_OF_COMPLEX
      intrinsic
         Value : string := "";
   end;
   node class COMPLEX_VALUE is a VALUE
      derived
         Elem_Type : type in NODE;
   end;
   node class MVALUE is a COMPLEX_VALUE end;
   node class BAG_VALUE is a MVALUE end;
   node class SQB is a BAG_VALUE, VALUE_PART_OF_COMPLEX end;
   node type sqb : SQB
    redef derived
     Elem_Type =
    ((self.=oPartNode=>:CONSTITUENT[1:1]).-cons_is_n->:NODE[1:1]).type;
     SFW_Term =
      "select " & concom (
                  concom ( "", all self.=oPartNode=>.Term ),
                  concom ( "", all self.=oPartSQB=>.SFW_Term ) ) &
      " from " & concom ( "", all self.-is_defined_by->.Declaration )&
      " where " & conand (
                  conand ( "true", all self.=PartNode=>.Formula ),
                  conand ( "true", all self.=PartSQB=>.SQB_Formula ));
      end;
   node class LIST_VALUE is a MVALUE end;
   node class MMEMBER is a NODE
      intrinsic
         Index : integer := 0;
   end;
   node type mmember : MMEMBER end;
   edge type contains_mm : MVALUE -> MMEMBER;
   edge type mm_is_n : MMEMBER -> NODE [1:1];
   node class CONSTITUENT is a NODE
      intrinsic
         Output : boolean := false;
   end;
   node type constituent : CONSTITUENT end;
   edge type is_defined_by : SQB -> CONSTITUENT;
   edge type cons_is_n : CONSTITUENT -> NODE [1:1];
   edge type orig_cons : NODE -> CONSTITUENT;
   node class ATTRIBUTE end;
   edge type attribute2er : ATTRIBUTE -> ENT_REL [1:1];
   edge type attribute2v : ATTRIBUTE -> VALUE [1:1];
end;
```

Figure 22: Full PROGRES graph scheme for HQL/EER, including all attribute declarations

with a non-empty textual part) SQL/EER query. The actual gathering of the information is expressed by means of *path expressions*, such as "all `self.=oPartNode=>.Term`". The declaration of the path `oPartNode` is shown below.

```
path oPartNode : SQB -> CONSTITUENT [1:n] =
    `1 => `2 in
```

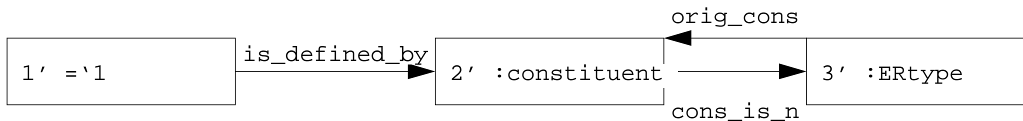
```
    condition `2.Output;
end;
```

Informally speaking, the above path-expression results in the set of all `Term`-attributes of all `constituent`-nodes reachable from the considered `sqb`-labeled node, which have been selected for output by means of their `Output`-attribute. Likewise, relevant declarations and formulas are gathered, and transformed into comma- or "and"-separated lists (by means of the functions `concom` and `conand`), and put in their proper place in the `select-from-where`-statement.

Some attributes needed in the translation of a hybrid query into its textual equivalent cannot be derived by means of such rules, since they depend on "externally provided" information (such as variable names) and hence have to be set in productions. For instance, in the complete version of production `Add_ER_labeled_node`, the `Term`-attribute of the newly created entity or relationship is assigned a given `Var`(iable)`Name`. The Declaration-attribute is set accordingly. Note that "&" stands for string-concatenation, while the function `string` converts a type-name to a string. Both the Term and Declaration attribute are copied to those of the corresponding (new) `constituent`-node. The latter copying operation is necessary in case the entity (or relationship) gets merged with another one to express an equality condition.

```
production Add_ER_labeled_node
      ( s : SQB ; VarName : string ; ERtype : type in ENT_REL ;
        out E : ENT_REL ; out C : CONSTITUENT )
 =
```

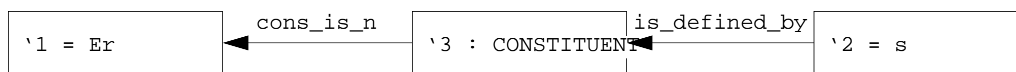
```
    ::=
```



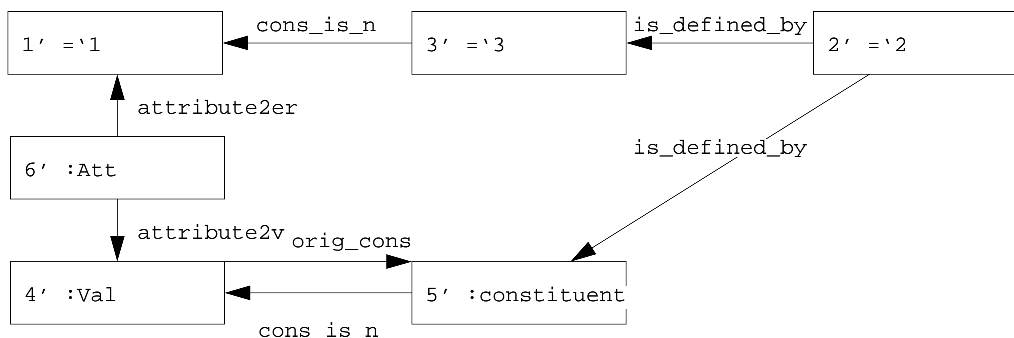

```
   transfer 3'.Term := VarName;
            3'.Declaration := VarName & " in " & string ( ERtype );
            2'.Term := VarName;
            2'.Declaration := VarName & " in " & string ( ERtype );
   return E := 3';
          C := 2';
end;
```

In the complete version of the production **Add_Attribute**, the **Term** of the newly created node is assigned the string concatenation of the term of the given entity or relationship, a dot and the string representation of the given attribute name. Once more, this information is copied to the **Term**-attribute of the corresponding **constituent**-node.

```
production Add_Attribute
   ( Er : ENT_REL ; s : SQB ; Att : type in ATTRIBUTE ;
     Val : type in VALUE ; out v : VALUE ; out C : CONSTITUENT )
   =
```




```
   ::=
```

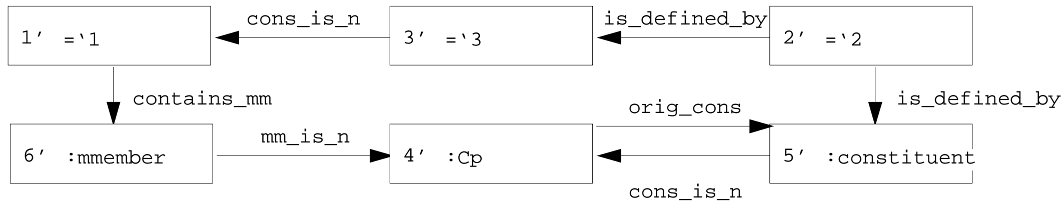



```
   transfer 4'.Term := `1.Term & "." & string ( Att );
            5'.Term := `1.Term & "." & string ( Att );
   return v := 4';
          C := 5';
end;
```

Since adding an element to a multi-value corresponds to the declaration of a variable, just as in the case of adding an entity or relationship, the `transfer`-clause of (the complete version of) the production `Add_to_Mvalue` is quite similar to that of the production `Add_ER_labeled_node`. Only, because this production may also be used to link nodes to `sqb`-nodes representing subqueries, it has to be checked if the given multi-value is a subquerybag or not.

```
production Add_to_Mvalue
   ( Mv : MVALUE ; s : SQB ; Cp : type in PART_OF_COMPLEX ; VarName : string ;
     out c : PART_OF_COMPLEX ; out C : CONSTITUENT ; out M : MMEMBER )
      =
```

```
      ::=
```




```
   condition '1.Elem_Type = Cp;
   transfer 4'.Term := VarName;
            4'.Declaration := VarName & " in " & [ '1.type = sqb :: '1.SFW_Term
                                                 | '1.Term ]                    ;
            5'.Term := VarName;
            5'.Declaration := VarName & " in " & [ '1.type = sqb :: '1.SFW_Term
                                                 | '1.Term ]                    ;
            6'.Term := VarName;
            6'.Declaration := VarName & " in " & [ '1.type = sqb :: '1.SFW_Term
                                                 | '1.Term ]                    ;
   return c := 4';
          C := 5';
          M := 6';
end;
```

As a final step in the semantics definition of HQL/EER, we provide a translation algorithm which transforms a hybrid query consisting of a textual and a graphical part into a purely textual SQL/EER query. Consider an HQL/EER query with a textual part "**select** T **from** D **where** F" (with T a list of terms, D a list of declarations, and F a formula), and graphical part G. As a result of the attribute evaluations in the PROGRES specification outlined above, we already have a (possibly incomplete) SQL/EER-query $\mathcal{Q}$, stored in the `SFW_Term`-attribute of the `sqb`-labeled node corresponding to the hybrid query. Then the following algorithm shows how to combine $\mathcal{Q}$ with the textual part, resulting in an SQL/EER query. The semantics of the HQL/EER query is then defined as the semantics of this SQL/EER query.

1. In both T, D and F, substitute any variable referring to a node in the graphical part, by the `Term`-attribute of this node;
2. Append T (with a comma) to the `select`-clause of $\mathcal{Q}$;
3. Append D (with a comma) to the `from`-clause of $\mathcal{Q}$;
4. Append F (with an "and") to the `where`-clause of $\mathcal{Q}$.

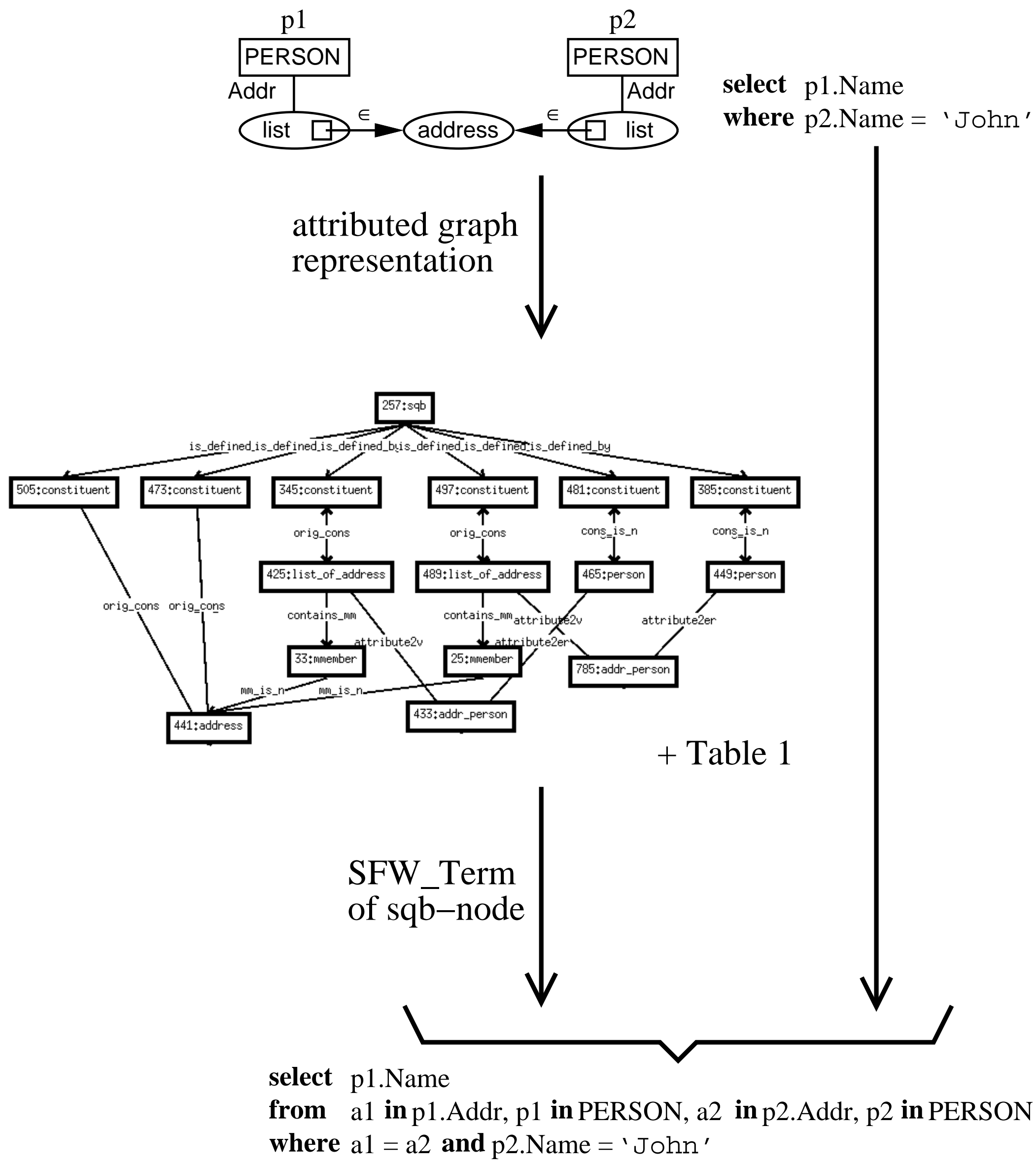


Figure 23: Translating a hybrid into a textual query

As shown in Figure 5.2, application of this algorithm to the hybrid query of Figure 8 results in the textual query of Figure 3.

Finally, note that since

- each HQL/EER query can (by definition) be translated into an equivalent SQL/EER query, and
- each SQL/EER query is (by definition) itself an HQL/EER query

HQL/EER and SQL/EER have precisely the same expressive power.

6 Implementation Efforts

Our claims about the advantages of a hybrid language over purely textual or purely graphical languages undoubtedly require validation, preferably in the form of the development and testing of a prototype implementation. Hence we are currently in the process of implementing a prototype Hybrid Query Tool that supports the specification and execution of hybrid queries [34]. In this Section we briefly discuss the major design issues and functionality of this tool.

A major design decision was to base the Hybrid Query Tool on the Object Modeling Technique [30] rather than on the EER model, mainly in order to demonstrate the applicability of the concepts of hybrid languages to other data models. From this experiment, we deduced the following two criteria a data model must satisfy, if one wants to define a hybrid query language on top of it:

1. the data model must allow for a natural and modular graphical expression of schemes; and
2. a closed textual query languages must be available.

Obviously, OMT satisfies the first criterion, but to our knowledge, no expressive closed query language has yet been developed based on OMT. Hence a first task preceding the design and implementation of a Hybrid Query Tool based on OMT, consisted in "translating" (a sublanguage of) SQL/EER to the context of OMT [34].

In its current state, the Hybrid Query Tool offers a (dummy) editor for hybrid queries. The editor was implemented in C++, using the Interviews 3.1 Toolkit. Figure 24 shows a screendump of the tools' user interface.

The interface includes the following four major components:

- a menubar (at the top), including six menus from which all operations offered by the tool may be activated;
- a palette (on the left hand side), from which the most commonly used operations offered by the tool may be activated;
- a scheme subwindow (at the top right side) into which an OMT diagram is displayed; and
- a query subwindow (at the bottom right side) in which a hybrid query can be constructed.

Specification of a hybrid query starts by loading an OMT diagram (produced by means of some other tool) into the scheme subwindow. In the scheme subwindow of the tool as depicted in Figure 24, an OMT diagram is depicted which models part of the running example scheme of this article.

Composition of the hybrid query in the query subwindow then proceeds by sequential application of some of the following editing operations:

- Copy & Paste of arbitrary subdiagrams from the scheme subwindow to the query subwindow (Buttons 1,3)

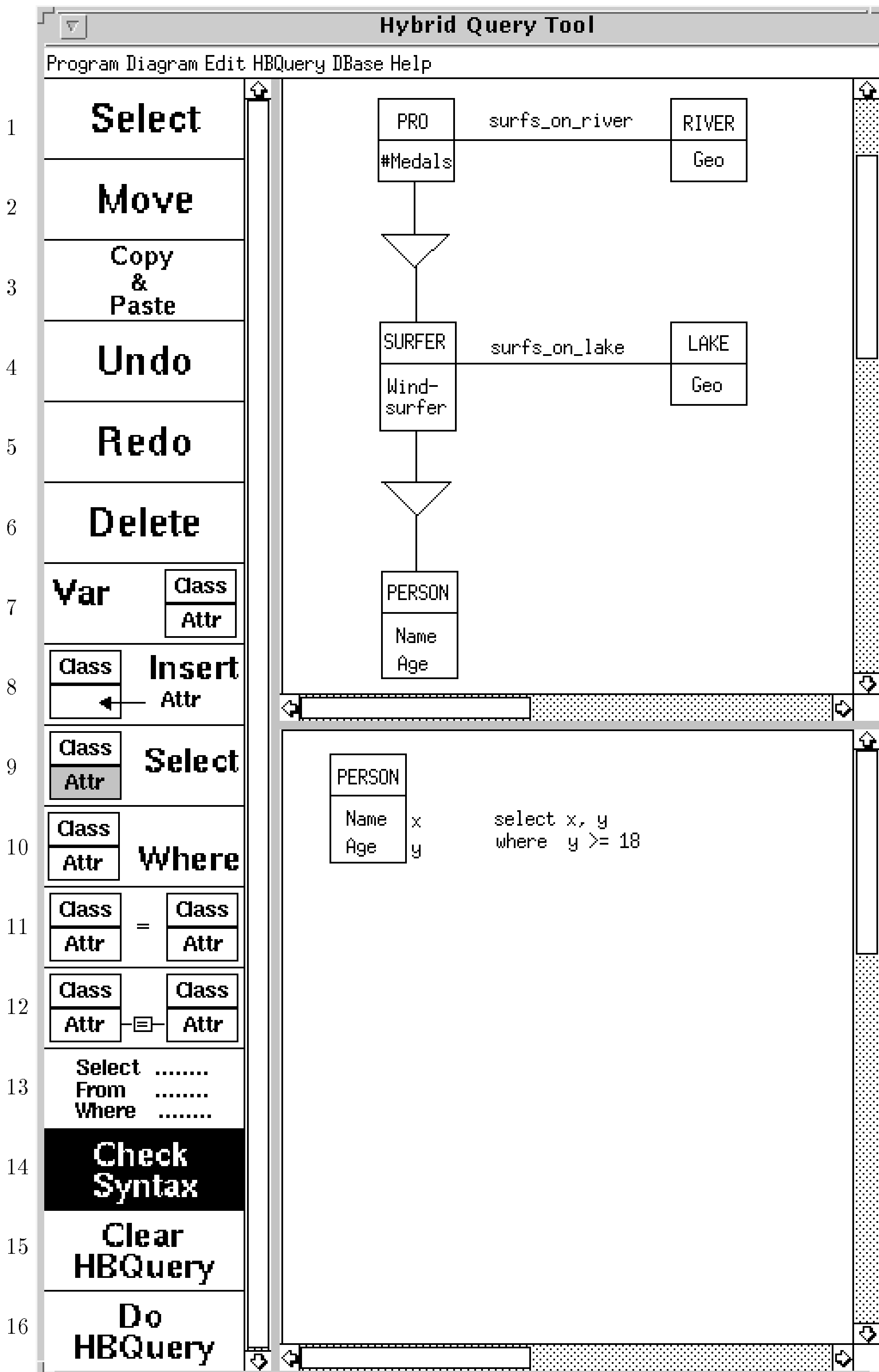


Figure 24: The user interface of the Hybrid Query Tool

- Showing/Hiding of attributes in a class-box in the query subwindow (Button 8)
- Merging of class-boxes in the query subwindow, to express equality conditions or to make inheritance of participation in associations explicit (Button 11)
- Linking attributes with edges labeled with an (in)equality predicate (Button 12)
- Textual specification of an atomic condition on a single attribute (Button 10)
- Selection for output of a single attribute (Button 9)
- Assignment of a variable to an attribute or class (Button 7)

Pressing Button 13 pops up a window in which the textual part of the hybrid query may be entered.

Most other buttons (such as the ones labeled "Move", "Undo", "Redo", "Delete" and "Clear HBQuery") are quite self-explaining.

Although the editor is syntax-directed to a certain degree (only correct "OMT-like" diagrams can be constructed in the query subwindow) there is still a need for a facility to explicitly check the syntax of a hybrid query (Button 14). A simple example of an incorrect query which could be build using the tool is that of a totally graphical one, in which nothing has been selected for output.

In its current state, the Hybrid Query Tool only allows syntax-directed *editing* of hybrid queries. A back-end for this tool, which will allow the actual *execution* of hybrid queries, is currently under development. Although the main topic of this article is the *formal definition* of HQL/EER, we nevertheless included this short description of the tool's (intended) functionality in order to give the reader an idea of how a tool can support usage of a hybrid query language.

7 Conclusions

In this article, we introduced the Hybrid Query Language for the Extended Entity Relationship Model. We formally defined both its syntax and semantics using programmed graph rewriting systems. Our future plans mainly concern the further development of the Hybrid Query Tool (briefly discussed in Section 6 of this article). Once this tool becomes available, we intend to thoroughly test both its functionality as well as the actual merits of the very notion of hybrid (query) languages by means of experiments with end-users.

Acknowledgments

We would like to extend our gratitude towards the anonymous referees for their numerous valuable suggestions for improvement of this article. Thanks go to Andy Schürr for his efforts in extending his formalism PROGRES in order to accommodate our needs in modeling HQL/EER. Thanks also to Nasser Silakhori for his work on the implementation of the Hybrid Query Tool. The outlook of the graphical part of HQL/EER queries was inspired by the view-tool of the XGOOD visual database interface described in [18], based on the Graph-Oriented Object Database model [21].

References

[1] *Proceedings of the Ninth ACM Symposium on Principles of Database Systems.* ACM Press, 1990.

[2] M. Andries. *Graph Grammars for Visual Database Languages.* PhD thesis, Leiden University, The Netherlands, 1995. In preparation.

[3] M. Andries, M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. Concepts for graph-oriented object manipulation. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *3rd International Conference on Extending Database Technology, Proceedings*, number 580 in Lecture Notes in Computer Science, pages 21–38, Berlin, 1992. Springer.

[4] M. Andries and J. Paredaens. A Language for Generic Graph-Transformations. In G. Schmidt and R. Berghammer, editors, *Proceedings of the 17th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 570 of *Lecture Notes in Computer Science*, pages 63–74, Berlin, 1992. Springer.

[5] M. Angelaccio, T. Catarci, and G. Santucci. *QBD*: A Graphical Query Language with Recursion. *IEEE Trans. Softw. Eng.*, 16(10):1150–1163, 1990.

[6] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, 1989.

[7] D. Bryce and R. Hull. SNAP: A Graphics-Based Schema Manager. In *Proceedings of the International Conference on Data Engineering*, pages 151–164, 1986.

[8] D. M. Campbell, D. W. Embley, and B. Czejdo. A relationally complete query language for an entity-relationship model. In P. P. Chen, editor, *Entity-Relationship Approach: The Use of ER Concept in Knowledge Representation*. IEEE CS Press/North-Holland, 1985.

[9] P. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[10] M. Consens and A. Mendelzon. GraphLog: a visual formalism for real life recursion. In ACM [1], pages 404–416.

[11] H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Graph-Grammars and Their Application to Computer Science, International Workshop*, volume 532 of *Lecture Notes in Computer Science*, Berlin, 1990. Springer.

[12] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1989.

[13] R. Elmasri and G. Wiederhold. GORDAS: A formal high-level query language for the entity-relationship model. In P. P. Chen, editor, *Entity-Relationship Approach to Information Modeling and Analysis*, pages 49–70. North-Holland, 1983.

[14] G. Engels. Elementary actions on an extended entity-relationship database. In Ehrig et al. [11], pages 344–362.

[15] 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*, 9(2):157–204, Dec. 1992.

[16] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments, Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, Apr. 1992.

[17] D. Fishman, J. Annevelink, E. Chow, T. Connors, J. Davis, W. Hasan, C. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M. Neimat, T. Risch, M. Shan, and W. Wilkinson. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, Frontier Series, pages 219–250. ACM Press, Addison-Wesley, New York, 1989.

[18] M. Gemis, J. Paredaens, and I. Thyssens. A visual database managment interface based on GOOD. In R. Cooper, editor, *Interfaces to Database Systems*, Workshops in Computing, pages 155–175. Springer, 1993.

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

[20] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In ACM [1], pages 417–424.

[21] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object model for end-user interfaces. In H. Garcia-Molina and H. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, volume 19:2 of *SIGMOD Record*, pages 24–33. ACM Press, 1990.

[22] U. Hohenstein and G. Engels. SQL/EER – Syntax and Semantics of an Entity-Relationship-Based Query Language. *Information Systems*, 17(3):209–242, 1992.

[23] T. Houchin. Duo: Graph-based database graphical query expression. In Q. Chen, Y. Kambayashi, and R. Sacks-Davis, editors, *Proceedings of The Second Far-East Workshop on Future Database Systems*, volume 3 of *Advanced Database Research and Development Series*, pages 286–295, Singapore, Apr. 1992. World Scientific.

[24] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.

[25] M. Kuntz. The gist of GIUKU: Graphical interactive intelligent utilities for knowledgeable users of data base systems. *SIGMOD Record*, 21(1), 1992.

[26] M. Kuntz and R. Melchert. Pasta-3's Graphical Query Language: Direct Manipulation, Cooperative Queries, Full Expressive Power. In P. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 97–105. Morgan Kaufmann, 1989.

[27] M. Nagl and A. Schürr. A Specification Environment for Graph Grammars. In Ehrig et al. [11], pages 599–609.

[28] P. Peelman, J. Paredaens, and L. Tanca. G-Log: A declarative graphical query language. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings 2nd International Conference on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 108–128, Berlin, Dec. 1991. Springer.

[29] M. Roth, H. Korth, and D. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[30] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[31] A. Schürr. Introduction to PROGRESS, an Attribute Grammar Based Specification Language. In M. Nagl, editor, *Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, pages 151–165, Berlin, 1989. Springer.

[32] A. Schürr. PROGRESS: a VHL-Language Based on Graph Grammars. In Ehrig et al. [11], pages 641–659.

[33] A. Schürr. PROGRESS-Editor: A text-oriented hybrid editor for PROgrammed Graph REwriting SyStems. In Ehrig et al. [11], page 67.

[34] N. Silakhori. Design and Implementation of a Hybrid Query Tool. Master's thesis, University of Leiden, Department of Computer Science, 1995. In Dutch.

[35] K.-Y. Whang, A. Malhotra, G. Sockut, L. Burns, and K.-S. Choi. Two-dimensional specification of universal quantification in a graphical database query language. *IEEE Trans. Softw. Eng.*, 18(3):216–224, Mar. 1992.

[36] M. M. Zloof. Query-by-example : a data base language. *IBM Syst. J.*, 16(4):324–343, 1977.

A An overview of PROGRES

PROGRES is an operational specification language based on **PRO**grammed **G**raph **RE**writing **S**ystems. We demonstrate the major characteristics of PROGRES by means of a small example specification, namely for a simple list data structure (see Figures 25, 26 and 27).

The basis of a PROGRES specification is a *graph scheme*. Such a graph scheme specifies a set of graph properties (i.e., structural integrity constraints and attribute dependencies) common to a certain collection of graphs. The following components of a graph scheme are distinguished:

- type declarations: these are used to introduce labels for nodes and edges in the considered collection of graphs, to declare and initialize node attributes;
- class declarations: these denote coercions of node types with common properties by means of multiple inheritance, hence they play the role of second order types. Class declarations may also include attribute declarations.

In the example (see Figure 25), three node classes are specified, namely `NODE`, and its two direct subclasses `LIST_ELEM` and `LIST_HEADER`. The latter two classes each have a single node type, namely `lheader` of class `LIST_HEADER` and `lelem` of class `LIST_ELEM`.

Furthermore, Figure 25 shows three edge type declarations. Like in many ER dialects, expressions of the form `[n:m]` denote cardinality constraints. For example, the constraint `[0:1]` in the declaration of the edge type `lfirst` indicates that each node of class `LIST_HEADER`, may have at most one (but possibly no) outgoing edge of type `lfirst`. Intuitively, the edge labeled `lfirst` (if existent) leaving a node labeled `lheader` (which is the only type of class `LIST_HEADER`) points to the first element in the list (which is represented by a node of type `lelem`).

In the `derived`-clause of node class declarations, attributes of nodes of this class (or of its subclasses) are declared, whose value may be derived automatically. Actual derivation rules for these attributes are provided either in the declaration of subclasses, or in productions. For example, the attribute `No_of_elements` of class `LIST_HEADER` is derived by counting the number of elements (by means of the function `card`) in the set of all incoming edges labeled `elem_of`. The latter set is denoted by means of the expression `self.<-elem_of-`.

In the `intrinsic`-clause, attributes are declared whose value depends on a user-supplied parameter of the production (see below) which creates the node. For example, each node of class `NODE` has an "externally" supplied `Name` (which is also a `key`-attribute), while every node of class `LIST_ELEM` contains some externally supplied `Information`.

The class definition also specifies a default value for all attributes.

Productions specify how graphs are constructed by substituting an isomorphic occurrence of one graph (called the *left-hand side* of the production) by an isomorphic copy of another graph (called the *right-hand side* of the production). Productions are *parametrized* by node classes (and atomic values).

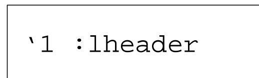
In the example specification (see Figure 25), the production `Create_New_List` has an empty left-hand side, and is therefore applicable to any graph. Hence it adds a node of type `lheader` to the graph to which it is applied. The production `Add_Last` (see Figure 26) appends a new element to an existing list. The list is passed to the production using the input parameter `l`. While matching the left-hand side to the graph, the node with identifier `'2` will match the last element of the list, as a result of the *restriction* `isLast`, which restricts the set of all nodes of class `LIST_ELEM` to those that have no outgoing edges labeled `next` (captured in the formula `not def -next->`).

Whereas restrictions determine a subset of a given set of nodes, *path expressions* determine a subset of a given set of *pairs* of nodes, hence they may be considered as "virtual edges". As an example, the path-expression `follows` (whose declaration is shown at the bottom of Figure 26) defines a "virtual" edge between two nodes n1 and n2, if there exists a non-empty sequence of `next`-edges from n2 to n1.

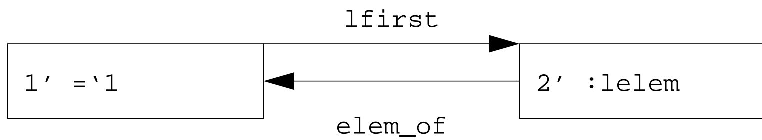
```
spec Lists

   section Graph_Scheme

      node class NODE
         intrinsic
            key Name : string := "";
      end;

      node class LIST_HEADER is a NODE
         derived
            No_of_elements : integer = 0;
         redef derived
            No_of_elements = card ( self.<-elem_of- );
      end;

      node class LIST_ELEM is a NODE
         intrinsic
            Info : string := "";
      end;

      node type lheader : LIST_HEADER end;

      node type lelem : LIST_ELEM end;

      edge type lfirst : LIST_HEADER -> LIST_ELEM [0:1];

      edge type elem_of : LIST_ELEM -> LIST_HEADER [1:1];

      edge type next : LIST_ELEM -> LIST_ELEM [0:1];
   end;

   section Productions

      production Create_New_List ( n : string ; out l : LIST_HEADER )
      =

         ::=

               +--------------+
               | 1' : lheader |
               +--------------+

         transfer 1'.Name := n;
         return l := 1';
      end;
```

Figure 25: A PROGRES specification for lists I

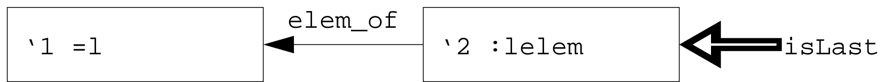
```
production Add_First ( n1 : string ; n2 : string ; i : string )
=
```



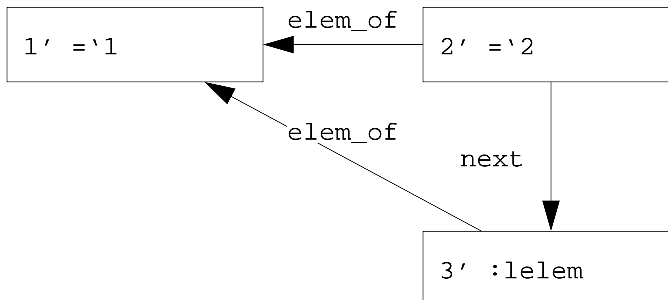

```
   ::=
```




```
   condition ‘1.Name = n1;
             ‘1.No_of_elements = 0;
   transfer 2’.Name := n2;
            2’.Info := i;
end;

restriction isLast : LIST_ELEM =
   not def -next->
end;

production Add_Last ( l : LIST_HEADER ; n : string ; i : string )
=
```




```
   ::=
```




```
   transfer 3’.Name := n;
            3’.Info := i;
end;

path follows : LIST_ELEM -> LIST_ELEM =
   <-next- +
end;
end;
```

In addition, productions may specify application conditions (in their `condition`-clause) in terms of structural and attribute properties of the isomorphic occurrence of the left-hand side. In the example specification, the production `Add_First` adds a new element to an empty list. For the sake of the example, the header of the list to which the element should be added, is not itself passed as a parameter, but by means of its key-attribute `Name` (in the input parameter `n1`). Hence, when applying the production, the corresponding node of type `lheader` has to be sought for, using the condition `'1.Name = n1`. In addition, the condition `'1.No_of_elements = 0` ensures that the list is indeed an empty one.

Attribute-computations are performed in the `transfer`-clause. For example, in the production `Add_First`, the input parameters `n2` and `i` are assigned to respectively the `Name` and `Info` attribute of the newly created node of type `lelem`.

The `embedding`-clause (not used in the example specification) states how to embed an isomorphic copy of the right hand side of the production in the considered graph, by means of additional edges.

Finally, the `return`-clause (used in the production `Create_New_List`) is used to return certain nodes to the calling *transaction*. A transaction is a collection of *calls* to productions, structured by means of a variety of programming constructs, such as loops, selections,....

Figure 27 shows the (single) `MAIN` transaction of the example specification. It consists of a single statement of the form "`use <variable-declarations> do <sequence of production calls> end`". This transaction creates a list named "list1" and adds to it two elements named "element1" and "element2", containing respectively the information "short" and "list". Figure 28 shows the graph representation of this list.

```
transaction MAIN =
   use l1 : LIST_HEADER
   do
         Create_New_List ( "list1", out l1 )
      & Add_First ( "list1", "element1", "short" )
      & Add_Last ( l1, "element2", "list" )
   end;
```

Figure 27: A PROGRES specification for lists III

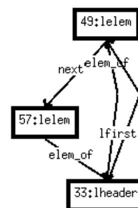


Figure 28: Graph representation of a list