A Hybrid Query Language for the Extended Entity-Relationship model

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 freely mix graphical and textual formulation of a query. We demonstrate the user-friendliness 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.

Keywords: User Interface, Query Language, Visual Interface, Object-Oriented Concepts

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 modelling 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 part of the model, but also its data *manipulation* language [5, 10, 12, 19, 25, 29, 34]. In several of these models, it is investigated to what extent arbitrary data manipulations may be expressed in a *purely* graphical way. One can conclude from this research that there is no limit to the expressive power that may be obtained with pure graph based manipulation languages [4].

However, one also gets the impression that some of this research overshoots its mark in the sense that the pure graphical formulation of a query sometimes even looks more complex than its textual equivalent. The obvious solution to this problem is to try and combine the "best of both worlds", i.e., to develop languages that allow those parts of an operation that are most clearly specified graphically resp. textually, to be indeed specified graphically resp. textually. In a sense, Zloof's Query-By-Example [35], which is commonly considered to be the first attempt at a "two-dimensional" query language, already offers facilities along this line, since complex conditions involving e.g., aggregate functions, are to be entered in plain text in what is called a *condition box*, rather then in the relation skeletons in which join conditions and selections can be entered. Similar facilities are offered in prototype interfaces for semantic database models, like SNAP [7]. In more recent proposals, like [27, 28], tools are presented which give the user a (limited) choice

*Work supported by COMPUGRAPH II, ESPRIT BRWG 7183.

between graphical and textual specification of operations. However, in the latter proposals, graphs and text may not be mixed within the same operation.

It is our aim in this paper to introduce a *hybrid query language*, in which almost any component of a given query may be expressed either textually or graphically, according to the users taste.[1] As data model, we use an extended version of the Entity Relationship model [13], the basic concepts of which we repeat in Section 2. However, we would like to stress that in our view, the concept of hybrid languages is generally applicable in the context of languages for other (e.g., object-oriented) database models. Our Hybrid Query Language for the Extended Entity Relationship model (called HQL/EER in the sequel) is an extension of SQL/EER, an SQL-like textual query language for the EER model (cf. Section 3). This language was inspired by some proposals made for query languages for the Entity Relationship model [8, 11], as well as proposals to extend SQL to cope with features of other database models than the relational one [30, 16]. Syntax and semantics of this language are defined using an Extended Backus-Naur Form grammar. After an informal introduction of HQL/EER by means of examples (cf. Section 4), we show in Sections 5 and Section 6 how the language is formally defined using the concepts of programmed and attributed graph rewriting systems [32]). Finally, in Section 7, we conclude with some practical issues and ideas for future work.

2 The Extended Entity-Relationship Model

Before we discuss textual as well as graphical query languages, we sketch briefly the main concepts of our Extended Entity-Relationship (EER) model [13, 23, 24]. It is based upon the classical Entity-Relationship model [9] and extended with the following concepts known from semantic data models [26]:

- 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 specification and generalization;
- several structural restrictions like the specification of keys, cardinality constraints,..., which are, however, of no interest to this paper.

Let us illustrate the EER model and its features with a small example. It models the world of surfing people who surf on different kinds of waters (cf. Figure 1). 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 `PROFI`, this time by `spec2`. This means that each `PROFI` 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`. We require the output types to have disjoint sets of instances. This means that in our example modelling, a water can be either a lake, or a river. In the case of specialization, we assume that all attributes are implicitly 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`.

[1] The term "hybrid" is in fact inspired by hybrid syntax directed editors, where the user can freely choose between a syntax-directed and a free style of editing [15].

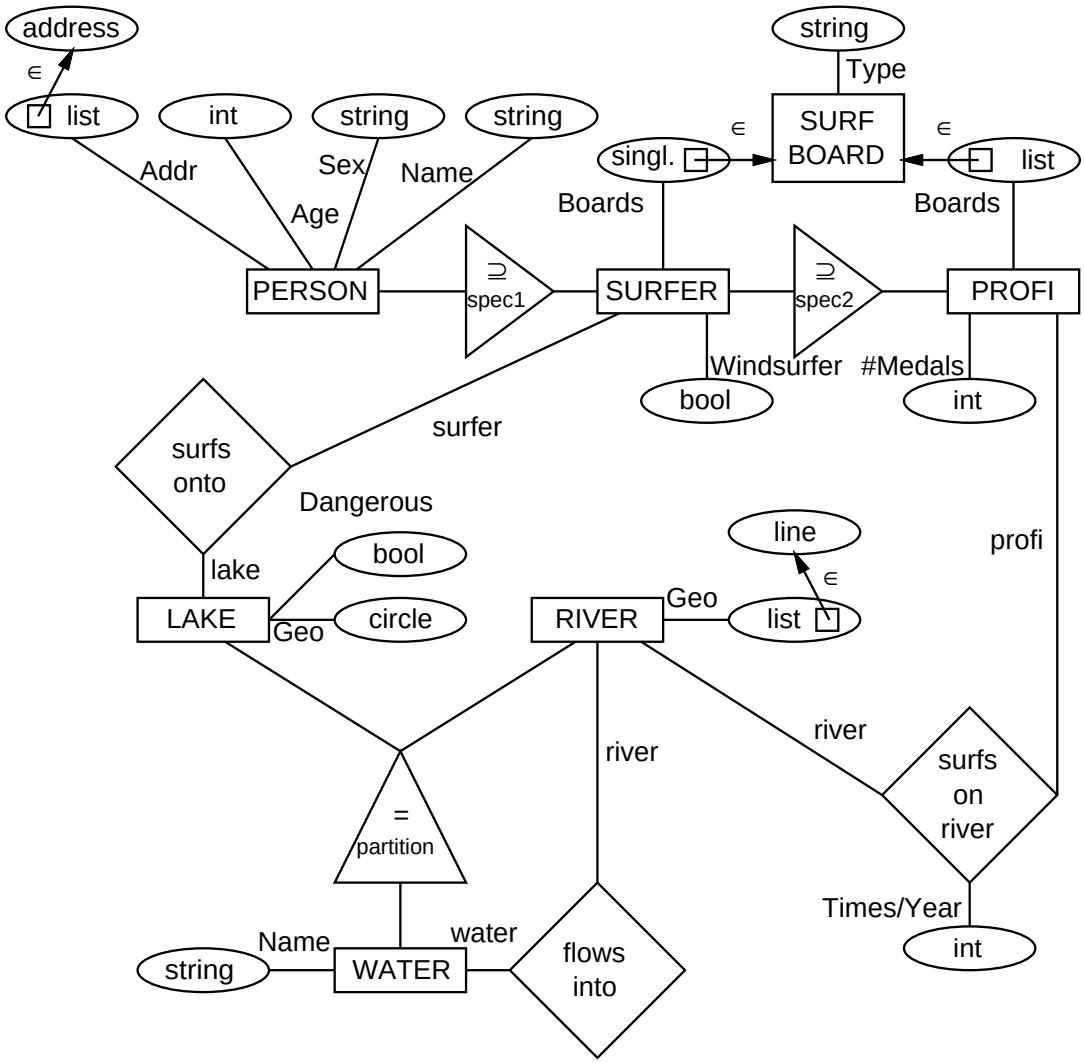


Figure 1: An EER diagram, modelling the world of surfers

We do not allow entity types to be constructed by several type constructions. Every entity type can only be constructed once. Furthermore, every constructed type must not, directly or indirectly, be 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 profis. 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, `Boards` is a component of `PROFI`, which consists of a list of references to instances of type `SURFBOARD`. Each profi possesses one or more surfboards. Components are always represented by an oval, even if they are single valued, in which case we label the oval with "singl.". Both attributes and components can be multivalued, i.e., set-, bag-, or list-valued. In this case, we write a square into the oval that is connected to the corresponding entity type or atomic value type via an arrow, which is always labeled $\in$. E.g., note that both the entity type `PROFI` and its 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 the entity type `PROFI` is a *list* of surf boards. This way, the known concept of *overriding* (both of attributes and components) is incorporated in the EER model.

3 SQL for the Extended Entity-Relationship Model

Based on the data model introduced in Section 2, we now informally repeat the main concepts of the textual query language SQL/EER. A complete description of this language can be found in [22]. Its formal semantics is based on a formally defined calculus for the EER model [18]. Both syntax and semantics of SQL/EER are defined by means of an attributed string grammar [21].

SQL/EER directly supports all the concepts of the EER model, and takes into account well known features that are an integral part of nowadays 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. This is captured in the following EBNF grammar rule.

SFW-TERM ::= **select** TERMLIST
from DECLLIST
[**where** FORMULA]

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 $\geq$ 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.

DECL ::= VARIABLE **in** ENTITYTYPE
VARIABLE := STRING
ENTITYTYPE := STRING

The variable p can now be used to build terms like p.Name and p.Age, to compute the name and age of the person p, respectively.

TERM ::= VARIABLE
| TERM '.' ATTRIBUTE

The formula "p.Age $\geq$ 18" uses the predicate "$\geq$", defined for the integer data type.

FORMULA ::= TERM DATAPRED TERM

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

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

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 a is bound to the finite list of addresses of person p1.

This query retrieves the name 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.

The last 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** PROFI
where p surfs_onto l **and** p surfs_on_river r **and** r flows_into l

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

Here, the variable p is declared of type `PROFI`. As profis 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.

```
FORMULA     ::= PARTICIPANT RELSHIPTYPE PARTICIPANT
                | RELSHIPTYPE '(' PARTLIST ')'
PARTICIPANT ::= TERM
PARTLIST    ::= PARTICIPANT [ ',' PARTLIST ]
```

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".

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 informally how this language is extended with *graphical alternatives* for some of its language constructs. The resulting language of this extension is called the *hybrid* query language HQL/EER.

Briefly, a query in HQL/EER consists of a piece of text (obeying the syntax of SQL/EER) and/or an attributed labeled graph. 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).

For expressing these declarations and conditions graphically, we restrict ourselves to the same 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 resp. `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 edges to both other nodes (cf. Figure 5).

From these restrictions, it readily follows that the graphical part of a hybrid query always consists of subgraphs of a graphical representation of the EER-scheme, in which nodes may be

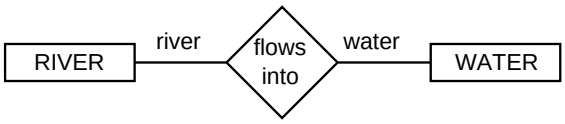


Figure 5: A sample graphical part of an HQL/EER query

"identified" to express sharing. Consequently, arithmetic and aggregate functions cannot be expressed in this language. However, it would not be difficult to think of graphical notations for these concepts also. Besides, since the graphical part of a hybrid query is only related to the "outermost" query, graphical parts cannot be associated to subqueries. Independent of these restrictions, the expressive power of HQL/EER is of course equal to that of SQL/EER.

Since (some) nodes in the graphical part of an HQL/EER query correspond to declared variables, "references" to such nodes may be used as variables in the textual part. This is the way in which the textual and graphical part of a hybrid query are linked together.

We now illustrate these concepts on some simple examples over the scheme of Figure 1. Figure 6 shows a possible expression of the query of Figure 2 in HQL/EER. Intuitively, the `PERSON`-node corresponds to the declaration of the variable p in the textual expression, while e.g., the `int`-node corresponds to the term p.Age in the textual expression, since it is linked to the node corresponding to the variable p by means of an `Age`-edge.

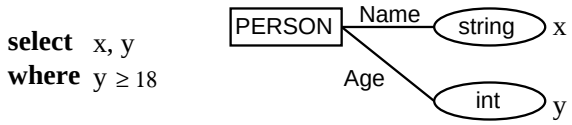


Figure 6: Name and age of adults (HQL/EER version)

Note that in the textual part of the hybrid query, two variables 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. 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 previous example, 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 "general" graph, which however still satisfies the structural constraints imposed by the scheme.

Figure 7 shows a way of expressing the 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 "a=b" in the textual 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 textual version of this query.

With another version of the SQL/EER query of Figure 3 (cf. Figure 8), we illustrate hybrid queries consisting *merely* of a graph. 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.

Since HQL/EER queries may be totally graphical, our formalism subsumes the purely graphical query languages mentioned in the Introduction. More precisely, the outlook of the graphical part

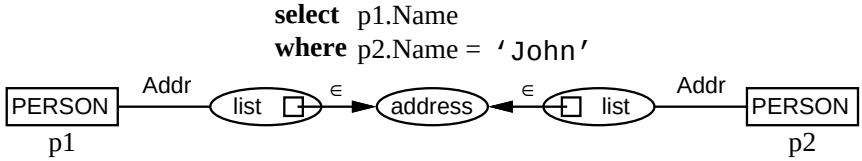


Figure 7: Name of persons sharing an address with John (HQL/EER version I)

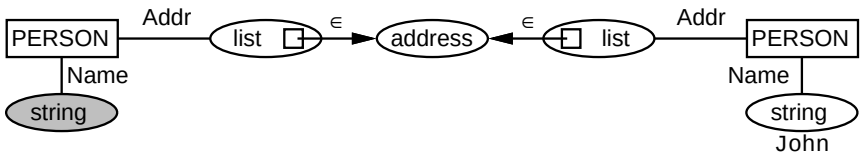


Figure 8: Name of persons sharing an address with John (HQL/EER version II)

of HQL/EER queries was inspired by the view-tool of the prototype visual database interface described in [17], based on the Graph-Oriented Object Database model [20].

Figure 9 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 a manner similar to SQL/EER. Since `PROFI` (resp. `LAKE`) inherits the participation in the `surfs_onto` (resp. `flows_into`) relationship from `SURFER` (resp. `WATER`), the graphical part of this query is still considered to satisfy the structural constraints imposed by the scheme.

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.

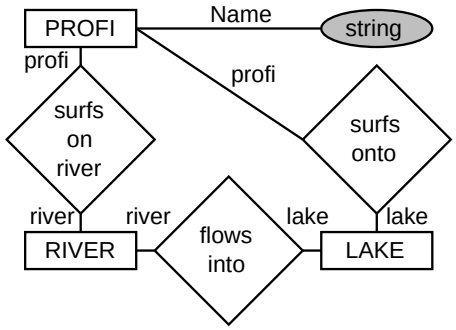


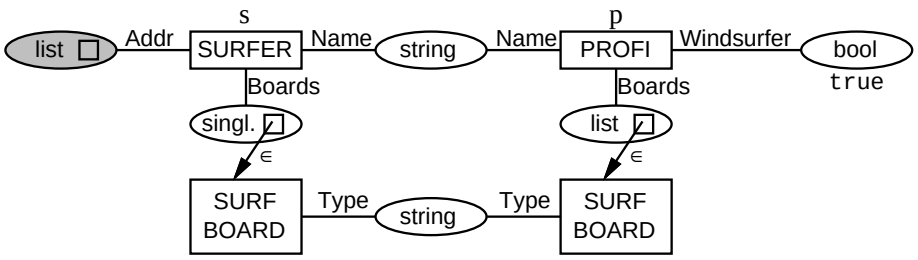
Figure 9: Names of profis, surfing on rivers, flowing into lakes they surf on (HQL/EER version)

We conclude this collection of examples with a slightly more involved one, illustrating our claim that HQL/EER allows those parts of an operation that are most clearly specified graphically resp. textually, to be indeed specified graphically resp. textually. The hybrid query of Figure 10 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.

As it is hard to express negation graphically, we leave the part of the query involving this negation in the textual part, and express everything else graphically.

In contrast, Figure 11 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.



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

Figure 10: "Involved" hybrid query

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

Figure 11: "Involved" textual 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 and the following section, we explain how syntax and semantics of HQL/EER are formally defined.

In this section, we concentrate on the syntax definition. On one hand, the formalization of HQL/EER involves the *representation* of the graphical part of hybrid queries as labeled, attributed graphs. Node and edge labels correspond to scheme elements, like entity type names. Node attributes are used a.o., for storing non-structural information which is part of the query, such as atomic values. On the other hand, we formalize the *construction* of an HQL/EER query as a sequence of applications of graph rewriting rules.[2]

As an example, consider Figure 12, which shows the attributed graph corresponding to the graphical part of the hybrid query of Figure 8. In the upper part of the rectangles, resp. near the arrows, we indicate the label of nodes resp. edges. In the bottom part of the rectangles, we specify the name and value of node attributes. `Formula` and `Decl` will be referred to as attributes of the graph itself (formally these might be looked upon as attributes of some uniquely labeled node, present in any graph). The precise meaning of the different node attributes and variables is explained in the remainder of this Section.

The need for both an expressive graph model and a powerful graph rewriting formalism, motivates our choice of the graph rewriting formalism called PROGRES [32, 33, 36].

Before showing how HQL/EER is formalized using PROGRES, we first give a short summary of its major characteristics. PROGRES is a very high level language based on the concepts of **PRO**grammed **G**raph **RE**writing **S**ystems, and was originally developed in the context of modelling software development environments [14].

Graph rewriting rules (or *productions*) in PROGRES are specified over a *graph scheme*. Such a graph scheme is a set of graph properties (i.e., structural integrity constraints and attribute dependencies) common to a certain category 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 category of graphs, to declare and initialize node attributes;

[2] Note that this does not imply that the user of HQL/EER should look upon the execution of his queries as graph rewritings 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 syntax and semantics of the query language.

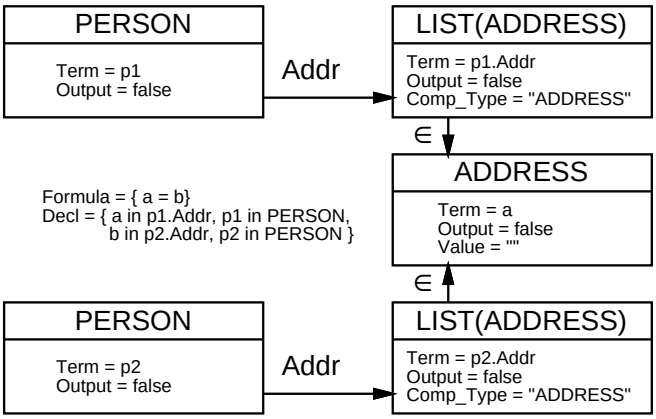


Figure 12: Graph representation of the hybrid query of Figure 8

- 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.

Productions specify when and 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 and edge types. Furthermore, we call rules *generic*, if they are specified using (among others) classes. Generic graph rewriting rules represent the *set* of parametrized productions, obtained by instantiating these classes with any of their types.

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. The **embedding**-clause states how to embed an isomorphic copy of the right hand side of the production rule in the considered graph. Attribute-computations are performed in the **transfer**-clause.

The remainder of this section is organized as follows: in a first step (cf. Section 5.1), we show how to capture the structure of the graphical part of HQL/EER queries in a PROGRES graph scheme and a set of generic productions. For instance, the graph scheme shows that there will be nodes for representing entities and nodes for representing values, while some (generic) production shows that given an entity, a value may be linked to it by means of an edge, which then represents one of the entities attributes. All this is still independent of any particular EER diagram.

In a second step (cf. Section 5.2), we show how this graph scheme is to be extended by further class as well as type definitions corresponding to a concrete EER diagram. Instantiation of the generic productions (resulting from the first step) with appropriate types then yields a set of productions which completely describe the attributed graph representation of allowed graphical parts of HQL/EER queries.

5.1 HQL/EER in terms of PROGRES classes

The starting point for the definition of HQL/EER is a formalization of the graphical notation introduced in Section 4 in terms of a PROGRES graph scheme. Figure 13 shows part of this scheme.

The node class NODE is the root of our node class hierarchy. In the **external**-section, attributes of nodes of types of this class (or of its subclasses) are declared, whose value depends on a user-supplied parameter of the production which creates the node. If the value of an attribute (like Comp_Type in the class COMPLEX_VALUE) is derived from that of other attributes, we use the keyword **derived**. The class definition also specifies a default value for all attributes.

```
section NODE_CLASSES;
    node class NODE
        external Term: String := "";
                 RefVar: String := "";
                 Output: Boolean:=false;
    end;
    node class ENT_REL is a NODE end;
    node class ENTITYTYPE is a ENT_REL end;
    node class VALUE is a NODE end;
    node class ATOMIC_VALUE is a VALUE
        external Value: String := "";
    end;
    node class COMPLEX_VALUE is a VALUE
        derived  Comp_Type: String := "";
    end;...
end;
```

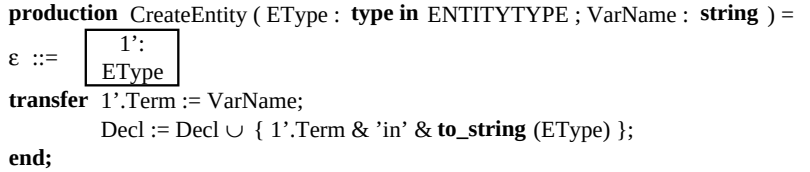
Figure 13: Definition of a PROGRES graph scheme, EER diagram independent part

The classes ENT_REL and VALUE are direct descendants of the NODE-class. ENT_REL is an extra common ancestor for classes ENTITYTYPE and RELSHIPTYPE, which is useful for the definition of edges ending in attribute nodes. The class VALUE in turn has two direct descendants, namely ATOMIC_VALUE and COMPLEX_VALUE. Nodes of types of the former class have an attribute in which the actual value is stored. For simplicity, we assume all values are stored as strings.

We now come to the graph rewriting rules specifying the syntax of the graphical part of HQL/EER. Since building this graph always starts with the creation of one or more isolated nodes, similar to the declaration of variables in the textual part, we must first specify a graphical alternative for some of the EBNF-rules for variable declarations.

Consider e.g., the graphical counterpart for the rule for declaring variables of an entity type.

DECL ::= VARIABLE in ENTITYTYPE



The left hand side of this PROGRES production is the empty graph (represented as ε), while the right hand side consists of a single new node of a type of class ENTITYTYPE. The quoted number is used as a node identifier in e.g., the **transfer**-clause. When creating a new node of some entity type, the value of the Term-attribute (i.e., the name of a variable) must be provided by the user. The corresponding declaration (built using the new nodes Term-attribute as well as its type, obtained by applying the built-in function **to_string**) is inserted in the graph-attribute `Decl`.

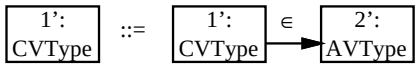
Note that there is no need to specify a graphical alternative for the rule which states that a variable is a term. The act of creating an isolated node in a graph corresponds to the declaration of a variable, while the node itself may readily be used as a term.

In some cases, a graphical alternative can also be used for declaring a variable ranging over

something else than an entity or relationship type. Concretely, if a variable ranges over a list-, bag- or set-valued attribute, the production shown below may be used.

DECL ::= VARIABLE in TERM

```
production VarInTerm ( CVType : type in COMPLEX_VALUE ;
                       AVType : type in ATOMIC_VALUE ;
                       ∈ : CVType -> AVType ; VarName : string ) =
```




```
condition  1'.Comp_Type = to_string (AVType);
transfer   2'.Term := VarName;
           Decl := Decl ∪ { 2'.Term in 1'.Term };
end;
```

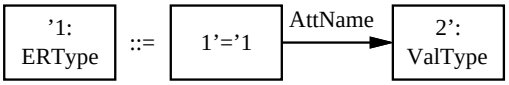
This production also illustrates a convention in the usage of quoted numbers. The same number is used for corresponding nodes in the left and right hand side of a production. A number preceded by a quote then refers to the node in the left hand side, while a number followed by a quote refers to the node in the right hand side.

The PROGRES part of the rule which states that a constant string is a term, is quite straightforward. It allows the addition of an atomic value to the graphical part of an HQL/EER query. The actual value must of course be provided as a parameter. This value is then assigned to both the Value- and the Term-attribute of the newly created node.

The graphical alternative for the rule which states that a term followed by an attribute is also a term, also has an obvious semantics: to a node of a type of class ENT_REL, a newly created node of a type of class VALUE is linked by means of an edge of a type with the proper source and target type. The term of this new node is assigned the string concatenation (expressed using the operator "&") of the term of the existing node, a dot and the string representation of the type of the edge.

TERM ::= TERM '.' ATTRIBUTE

```
production AddAttribute ( ERType : type in ENT_REL ; ValType : type in VALUE ;
                          AttName : ERType -> ValType ) =
```




```
transfer  2'.Term := 1'.Term & "." & to_string (AttName);
end;
```

A complete description of the (hybrid) grammar of HQL/EER (including e.g., productions for roles, formulas, components, constructions,...) can be found in [2].

5.2 HQL/EER in terms of PROGRES types

In the foregoing subsection, we have shown how to define the syntax of the graphical part of HQL/EER queries in terms of a PROGRES graph scheme and associated PROGRES productions. As classes play the role of second order types, productions specified using classes cannot be applied to concrete graphs without the declaration of a set of related first order types. Given such a set of type definitions, a rule specified in terms of classes actually represents the set of rules, obtained by substituting any class by a type of this class. If a given EER scheme is mapped to an extension of the graph scheme outlined in the foregoing Section, it is guaranteed that any application of a PROGRES rule from the syntax definition of HQL/EER results in a graph that obeys the

```
section NODE_CLASSES;
    node class PERSON_C is a ENTITYTYPE end;
    node class SURFER_C is a PERSON_C end;
    node class WATER_C is a ENTITYTYPE end;
    node class RIVER_C is a WATER_C end;...
end;
section NODE_TYPES;
    node type SURFER : SURFER_C end;
    node type PERSON : PERSON_C end;
    node type FLOWS_INTO : RELSHIPTYPE end;
    node type STRING : ATOMIC_VALUE end;
    node type ADDRESS : ATOMIC_VALUE end;
    node type LIST_OF_ADDRESS : ORDERED_VALUE
        derived Comp_Type:String := "ADDRESS";
    end;...
end;
section EDGE_TYPES;
    edge type Name : PERSON→STRING;
    edge type Addr : PERSON→LIST_OF_ADDRESS;
    edge type Water : FLOWS_INTO→WATER;
    edge type River : FLOWS_INTO→RIVER;...
end;
```

Figure 14: Definition of a PROGRES graph scheme, EER diagram dependent part

structural constraints imposed by this scheme. Figure 14 shows part of the graph scheme for the translation of the scheme of Figure 1.

The node classes whose name is suffixed with a "C" are introduced to cope with inheritance. 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. 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 edge types.

Given these extra definitions, the production for declaring variables of an entity type may for instance 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 NUMBER to an existing node of type PERSON by means of an edge of type Age or Length. The "instantiated" production for specifying the age of a person is shown in Figure 15.

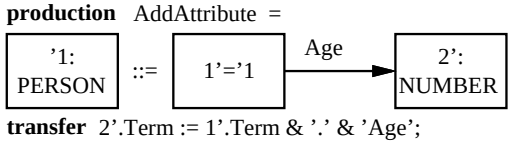


Figure 15: An instantiation of a PROGRES rule

6 Semantics of HQL/EER

In this section, we define the semantics of HQL/EER queries in terms of the formally defined semantics of SQL/EER [21]. We do this by providing a translation algorithm which transforms a hybrid HQL/EER 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 term list, D a list of declarations, and F a formula), and graphical part G. Then the following algorithm shows how to augment the textual part with information from the attributes of G's nodes and G's global variables, 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. Add to T (using commas) the Term-attribute of any node in G whose Output-attribute is true;
3. Add to D (using commas) all declarations in Decl;
4. Add to F (using conjunctions) all formulas in Formula.

The reader may verify that an application of this algorithm to the hybrid query of Figure 8 (whose graph-representation is given in Figure 12) results in the textual query of Figure 3 (cf. Figure 16).

7 Future Work

In this paper, we formally defined a hybrid query language, allowing the intertwined usage of text and graphs for the specification of queries. Releasing some of the restrictions we initially imposed, provides a number of ideas for further research:

- As mentioned previously, we restricted the graphical part of HQL/EER queries to using only those graphical elements also present in the graphical notation for EER schemes. This condition may be relaxed to allow e.g., graphical representation of functions, predicates and quantification (cf [34]).
- The graphical part of an HQL/EER query only applies to the "outermost" level of the textual part of the query. We are currently investigating how graphs may be associated to subqueries.
- The concept of hybrid languages may well be applied to languages for the specification of other things than queries, e.g., for updates.

On the more practical side, our claims about the advantages of a hybrid language over purely textual or purely graphical languages require validation, preferably in the form of the development and testing of a prototype implementation

In view of this, we conclude this article by shortly elaborating on how we imagine an interface should allow users to specify the mixture of graphs and text of which an HQL/EER query consists. Our proposal is to let this process be guided (as much as possible and feasible) by the interface. In other words, hybrid queries should be specified in a *syntax directed* manner. A primary observation to be made however, is that syntax directed editing of graphs and text are two basically different things. In a sense, they are even each others opposite.

On one hand, when a user starts an editing session with a syntax directed textual editor for some given formal language, he is normally presented a single non-terminal symbol of the language's grammar. For instance, a session with a syntax directed editor for SQL/EER would start with the non-terminal QUERY. This non-terminal may then be expanded by recursively applying rules from the grammar. As a result of such an application, an occurrence of the left hand side in the current expression is replaced by the right hand side. For instance, the non-terminal FORMULA may be replaced by the sequence of non-terminals "TERM DATAPRED TERM". Briefly, syntax directed editing of text is essentially a *top down* process.

On the other hand, composing graphs is essentially a *bottom up* process. Imagine making the drawing of Figure 9 with some mouse driven drawing tool. Typically, you first make some loose labeled nodes, and later on connect these nodes into a semantically meaningful graph. As observed in [3], "syntax direction" may be added to such a process. The paper outlines how purely graphical queries (as well as many other types of database operations) may be constructed by among others, copying and pasting graph-increments from a graphical representation of the considered scheme. In this paper, we formalized this bottom up specification of the graphical part of a hybrid query by means of graph rewriting rules, associated to rules from the (textual) grammar of SQL/EER. Building the graphical part of an HQL/EER query using these rules, always starts with an empty graph. Recursive application of graph rewriting rules results in the creation of nodes or edges or the assignment of values to node and edge attributes. Created nodes are always "terminal", i.e., they never play the role of "placeholders" like non-terminals in the specification of the textual part do. This rule driven specification process corresponds closely to the drawing tool approach.

We are currently preparing the implementation of a hybrid syntax directed editor.

Acknowledgments

Thanks go to Andy Schürr for our discussions concerning the extensions to be made to his formalism PROGRES, in order to accomodate our needs in modeling HQL/EER.

References

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

[2] M. Andries and G. Engels. A Hybrid Query Language for the Extended Entity Relationship Model. Technical Report 93-15, Leiden University, Dept. of Comp. Science, 1993.

[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 Schmidt and Berghammer [31], pages 63–74.

[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] 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.

[12] G. Engels. Elementary actions on an extended entity-relationship database. In 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*, pages 344–362, Berlin, 1990. Springer.

[13] 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.

[14] G. Engels, C. Lewerentz, and W. Schäfer. Graph Grammar Engineering – A Software Specification Method. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, International Workshop*, volume 291 of *Lecture Notes in Computer Science*, pages 186–201, Berlin, 1987. Springer.

[15] G. Engels and W. Schäfer. *Programmentwicklungsumgebungen, Konzepte und Realisierung*. Leitfäden der Angewandten Informatik. B.G.Teubner, Stuttgart, 1989.

[16] 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.

[17] M. Gemis, J. Paredaens, and I. Thyssens. A visual database managment interface based on GOOD. In *Proceedings of the International Workshop on Interfaces to Database Systems*, 1992. To appear.

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

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

[20] 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.

[21] U. Hohenstein and G. Engels. Formal Semantics of an Extended Entity-Relationship Query Language. In S. Spaccapietra, editor, *Proceedings of the 9th International Conference on Entity-Relationship Approach*, 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] U. Hohenstein and M. Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. In C. Batini, editor, *Proceedings of the 7th International Conference on Entity-Relationship Approach*, pages 129–148, 1988.

[24] 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, *Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen*, volume 143 of *Informatik-Fachberichte*, pages 58–88. Springer, 1987.

[25] 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.

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

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

[28] 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.

[29] 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*, number 566 in Lecture Notes in Computer Science, pages 108–128, Berlin, Dec. 1991. Springer.

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

[31] 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*, Berlin, 1992. Springer.

[32] 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.

[33] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen.* PhD thesis, RWTH Aachen, 1991. Deutsche Universitäts Verlag, Wiesbaden. (in German).

[34] 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.

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

[36] A. Zündorf and A. Schürr. Nondeterministic Control Structures for Graph Rewriting Systems. In Schmidt and Berghammer [31], pages 48–62.

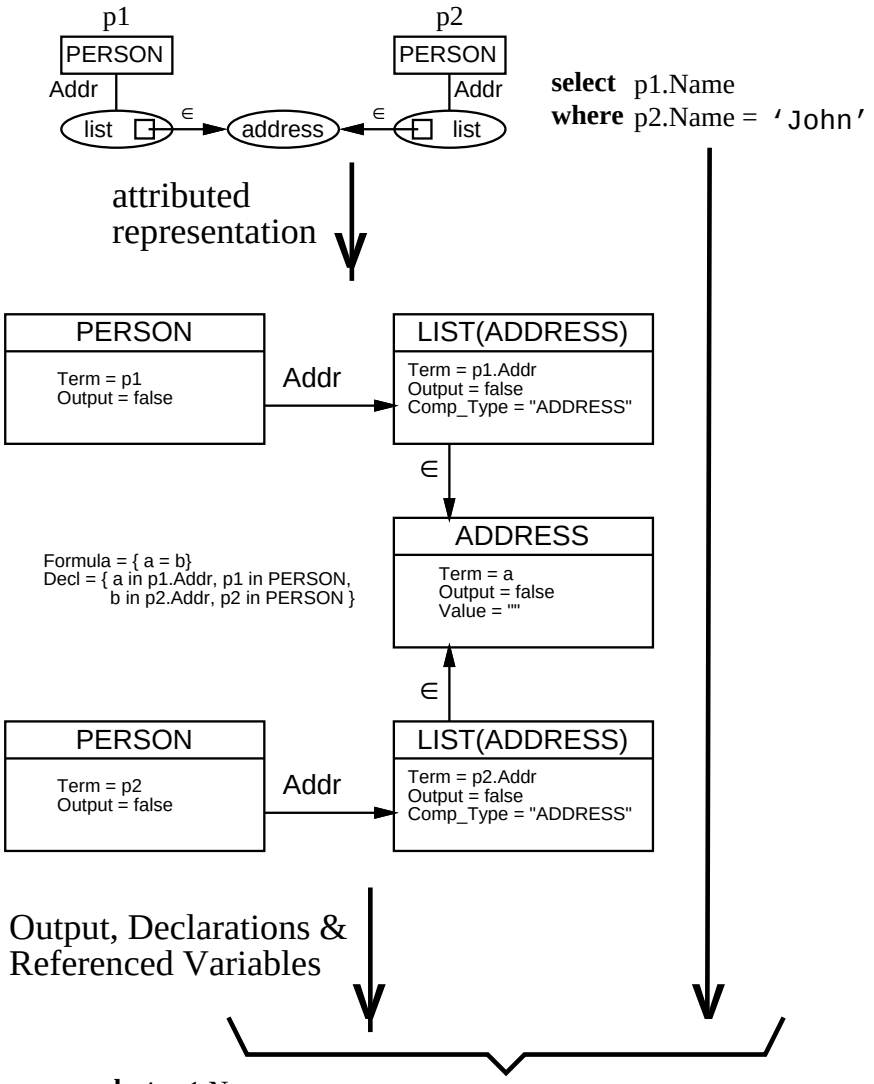


Figure 16: Translating a hybrid into a textual query