GraphDB: A Data Model and Query Language

for Graphs in Databases

Ralf Hartmut Güting

Praktische Informatik IV, FernUniversität Hagen
D-58084 Hagen, Germany
gueting@fernuni-hagen.de

Abstract: We propose a data model and query language that integrates an explicit modeling and querying of graphs smoothly into a standard database environment. For standard applications, some key features of object-oriented modeling are offered such as object classes organized into a hierarchy, object identity, and attributes referencing objects. Querying can be done in a familiar style with a *derive statement* that can be used like a *select … from … where*. On the other hand, the model allows for an explicit representation of graphs by partitioning object classes into simple classes, link classes, and path classes whose objects can be viewed as nodes, edges, and explicitly stored paths of a graph (which is the whole database instance). For querying graphs, the derive statement has an extended meaning in that it allows one to refer to subgraphs of the database graph. A powerful *rewrite* operation is offered for the manipulation of *hetereogeneous sequences of objects* which often occur as a result of accessing the database graph. Additionally there are special graph operations like determining a shortest path or a subgraph and the model is extensible by such operations. It is possible to compute additions to the database graph as well as restrictions in a query.

Besides being attractive for standard applications, the model permits a natural representation and sophisticated querying of *networks*, in particular of *spatially embedded networks* like highways, public transport, etc. The GraphDB model is meant to be implemented; system architecture and a representation and query processing strategy are outlined in the paper.

Keywords: Graphs in databases, data model, query language, explicit paths, object-oriented modeling, relationships, sequence rewriting, system architecture, extensibility

February 1994

This work was supported by the ESPRIT Basic Research Project 6881 *AMUSING*.

1 Introduction

The work described in this paper arose from the observation that existing data models and query languages (and the database systems realizing them) do not offer adequate support for the modeling and querying of *networks*. In particular, we are interested in *spatially embedded networks* which are an important part of geographic information, for example, highways, rivers, public transport systems, power and phone lines etc. Current spatial database models and systems (e.g. [SvH91, RoFS88, OrM88, Gü89]) can well enough represent the geometry of such networks but have no concept of their connectivity. That is, one can express in a query which highways pass through a given region, but cannot find a path from a given position to a destination on the highway network.

We feel that the most natural representation of a highway network (taking it as a prototype for many kinds of spatial networks) is to view it as a graph whose nodes are highway junctions, whose edges are highway sections, and where highways are just certain paths over this graph. Therefore we would like to offer a data model capable to express this directly so that one can define a graph structure with corresponding node, edge, and path objects. For querying, special graph operations should be provided such as finding a shortest path from a start node to a target node, determining a subgraph within a given radius from a start node, etc.

On the other hand, modeling and querying networks is certainly not the only thing a user wants to do; hence, all of the more traditional applications should be supported as well in model and query language, and preferably in a style that is not too different from what one knew before. The challenge is therefore to achieve a smooth integration of the desired graph modeling into a more classical environment. Ideally, if one is not interested in networks, this model should be usable like any of the well-known models, e.g. a relational, functional, or object-oriented one.

The purpose of this paper is to present a data model and query language that achieves such a smooth integration. On the one hand, we show that traditional applications can be modeled and queried in a familiar style, and indeed, better than before, because this model offers very attractive features to represent relationships between objects and to use them in queries. So we claim that even without considering networks, this model is suitable and quite interesting as a general purpose data model. On the other hand, sophisticated modeling and querying of networks is possible, as we demonstrate by a number of examples. Our approach can be summarized as follows:

- The data model contains a few salient features of object-oriented models: A database is a collection of object classes. Objects have identity and a tuple structure; attributes may be data or object-valued. Classes are organized in an inheritance hierarchy. Central tool for querying is a *derive statement* which so far offers similar capabilities (in an equivalent syntactical structure) as the traditional *select … from … where*.
- The data model offers graphs: There are three different *kinds* of object classes called *simple classes*, *link classes*, and *path classes*. Simple objects are just objects, but also play the role of nodes in the database graph. Link objects are objects with additional distinguished references to source and target simple objects. Path objects are objects with an additional list of references to simple and link objects that form a path over the database graph.
- For querying the graph structure, (1) the derive statement has an extended meaning: In the *on-clause* (the counterpart to the *from …*) one can refer to connected subgraphs of the database graph and so specify relationships between simple objects, link objects, and path objects. (2) There is a special tool for sophisticated manipulation of paths returned from

queries (or, more generally, heterogeneous sequences of objects). (3) There is a collection of graph operations; they can specify argument graphs (subgraphs of the database) in the form of regular expressions over link class names (edge types). (4) The database graph can be extended or restricted dynamically within a query.

The paper offers a complete formal definition of the data model. Additionally it also has a quite practical flavor and intent. For example, it deals with language aspects, defining precisely the syntax of data definition commands and central parts of the query language. A system architecture and implementation strategy is also outlined.

In the literature, the manipulation of graphs in databases has received quite a bit of attention, a survey can be found in [MaS90]. However, to our knowledge nowhere has the focus been on an *explicit representation of graphs* together with a *smooth integration* into standard modeling and querying. Most authors assume that graphs can be modeled implicitly in terms of the usual features of a given data model. Often, this has been the relational model (e.g. [Kung86, StR86, Ag87, BiRS90]). Here a graph is usually represented through a domain D of values, which are interpreted as nodes, and a relation G with two attributes of type D, whose tuples are interpreted as edges (the other attributes can be viewed as edge labels, weights, etc.). A functional data model has also been used, two ways for representing graphs in this model are mentioned in [Rose86]. In most proposals the authors do not really care how graphs are represented but just focus on the abstract graph structure and the study of interesting operations or traversal techniques [CrMW87a, CrMW87b, CrN89, Rose86]. In all of these approaches there is no explicit modeling of graphs within a general database environment and therefore no problem of integration with the data model.

As far as queries are concerned, two main strategies are to offer general purpose facilities that allow to express graph traversal problems (like recursion, iteration) [Rose86, StR86], or to offer special operators [Ag87, Rose86, CrN89]. [BiRS90] propose an SQL extension based on the idea to generate a set of paths in the *from-clause* from which interesting paths are selected (with the help of special operators) and then projected into a relation again. An entirely different style of querying is suggested in [CrMW87a, CrMW87b, CoM90, CoM93]). The idea is to formulate a query as a set of graphs which are viewed as *patterns* (edges in patterns may be regular expressions over edge labels); all subgraphs of the database instance are returned matching these patterns. Since in all of this work the assumption is that graphs are represented by standard facilities of the data model, there is also a standard data manipulation language available. A smooth integration with this language has not been the focus of interest (except perhaps in [BiRS90]); in some cases, graph querying has a very different style from the rest of data manipulation.

In the work of [GyPV90a, GyPV90b, Andr92, GePTV93] the approach is different: Here the idea is to model the database directly and entirely as a graph and to express *all* queries in terms of a few powerful graph manipulation primitives. Graphical user interfaces are intended for direct input of queries and visualization of results in terms of graph structures.

We feel that an explicit modeling of graphs is very desirable for several reasons:

- It leads to a more natural modeling; graph structures are visible for the user,
- queries can refer directly to this graph structure,
- path objects can be defined (not present in any of the other models) and they are the interesting entities in most networks,
- the system can offer special data structures for graphs, to make traversal efficient,

- the system can contain efficient graph algorithms designed to utilize the special graph data structures.

Indeed, practically all of the research work mentioned above *does* represent graphs explicitly in the papers, but fails to integrate it with the rest of data modeling and querying.

The general approach described in this paper has been pursued in our own previous work [Gü91, ErG91] and that of collaborators [AmS92]. In [Gü91] relations and graphs were assumed to coexist as modeling facilities. However, there are some difficulties with that: A graph consisting only of nodes is practically the same as a relation. Also, if possibly several graphs occur in a database, it is hard to separate them in the design (which object class belongs into which graph). In [ErG91] graphs occur in an environment with object classes but are still separate entities. There is the same problem of partitioning a database into graphs. Also, in both approaches it becomes a nuisance to mention the graph arguments in many places in queries. Apart from modeling, the querying facilities offered in this paper go far beyond those of [Gü91, ErG91]. Amann and Scholl [AmS92] offer a few selected features of our model (node and edge objects, but no paths, and an idea similar to our derive statement) in the context of hypertext applications.

The paper consists of three major sections, describing the data model, querying facilities, and an implementation strategy, respectively.

2 The Data Model

This section introduces the data model of GraphDB. We start with an overview and show the modeling of some example applications. In the following subsections, the model is developed more formally and systematically, defining bottom up the notions of data types, object types, and tuple types, three kinds of object classes, a database and the database graph. The main purpose in the design of the data model is to achieve a "seamless" integration of graph structures and graph operations into the usual facilities for data modeling and querying.

2.1 Overview

A database is a collection of object classes which are partitioned into three kinds of classes, called *simple classes*, *link classes*, and *path classes*. Objects of a *simple class* are on the one hand just like objects in other models: They have an object type and an object identity and can have attributes whose values are either of a *data type* (e.g. integer, string) or of an *object type* (that is, an attribute may contain a reference to another object). So the structure of an object is basically that of a tuple or record. On the other hand, objects of a simple class are *nodes* of the *database graph* – the whole database can also be viewed as a single graph. Objects of a *link class* are like objects of a simple class but additionally contain two distinguished references to source and target objects (belonging to simple classes), which makes them *edges* of the database graph. Finally, an object of a *path class* is like an object of a simple class, but contains additionally a list of references to node and edge objects which form a path over the database graph.

Besides the graph structure, object classes are organized into a class hierarchy and there are related notions of subtyping among tuple types, object types, and data types. Let us now consider some examples of data modeling with these facilities.

2.1.1 Standard Applications

As a simple standard application, consider the representation of books and their authors. We describe the database schema by showing corresponding data definition commands.

```
create class book = title: STRING, publisher: STRING, year: INTEGER;
create class person = name: STRING, address: STRING;
create link class wrote from person to book;
```

Here we have two simple classes *book* and *person* and a link class *wrote*. Observe how a link class can directly represent a many-many relationship. Attributes may be defined for link classes in the same way as for simple classes. Attributes may also contain object references. For example, we might define persons to contain a reference to their home country:

```
create class state = name: STRING, region: REGIONS;
create class person = name: STRING, address: STRING, country: state;
```

Here REGIONS is a geometric data type describing the area covered by the state. Note that the model contains two different features for describing "connections" ("associations", "relationships") between objects, namely link classes and object-valued attributes. A discussion why both are needed and which should be used when is given below.

2.1.2 Highway Network

The highway network is a relatively simple example of a spatially embedded network. It is a graph whose nodes are highway junctions and exits; each of those has an associated point in the geometric (or geographic) plane. We assume junctions are characterized by a name and exits by a number. Edges of this graph are highway sections: pieces of road between junctions and/or exits with an associated geometry which is a polyline in the plane. The most interesting objects of this network are highways: they correspond to paths over the graph given by junctions, exits, and highway sections.

```
create class vertex = pos: POINT;[1]
create vertex class junction = name: STRING;
create vertex class exit = nr: INTEGER;
create link class section = route: LINE, no_lanes: INTEGER, top_speed:
     INTEGER from vertex to vertex;
create path class highway = name: STRING as section+;
```

Here junctions and exits are introduced as subclasses of a simple class *vertex*, which means they inherit the *pos* attribute. It also means that objects of both classes can be used as sources and targets of *section* edges of the graph. Indeed, there can also be "pure" *vertex* objects as nodes in the graph; they are useful to separate highway sections with different values of attributes such as *no_lanes*. The

---

[1] In this paper, in various places examples of spatial data types and their operations are used. These are all taken from the ROSE algebra [GüS93]. This algebra has types POINTS, LINES, and REGIONS, as shown in Figure 1. A value of type POINTS is generally a set of points which can of course consist of just a single element. Similarly a value of type LINES is a set of polygonal lines. For clarity, in the data definitons we write POINT and LINE to show the intent of having only a single element in each case.

highways themselves are defined to be paths over *section* edges. Essentially the expression behind the keyword *as* is a regular expression defining a *path type* which in turn describes a set of paths of the database graph. Path types are more interesting when different kinds of edges occur in a graph. We will see examples and a more precise definition below.

2.2 Data Types, Object Types, and Tuple Types

2.2.1 Data Types

Let $(D, \leq)$ be a finite set whose elements are called *data types*, with a partial order "≤" ("subtype") which is restricted to organize data types into trees (that is, $\forall\ a, b, c \in D$: $a \leq b \wedge a \leq c \Rightarrow b \leq c \vee c \leq b$). If two data types belong to the same tree, we call them *related*. If two data types are related, then a *smallest common supertype*, denoted $lub(a, b)$ exists and is uniquely defined. Figure 1 shows a collection of data types organized into several trees.

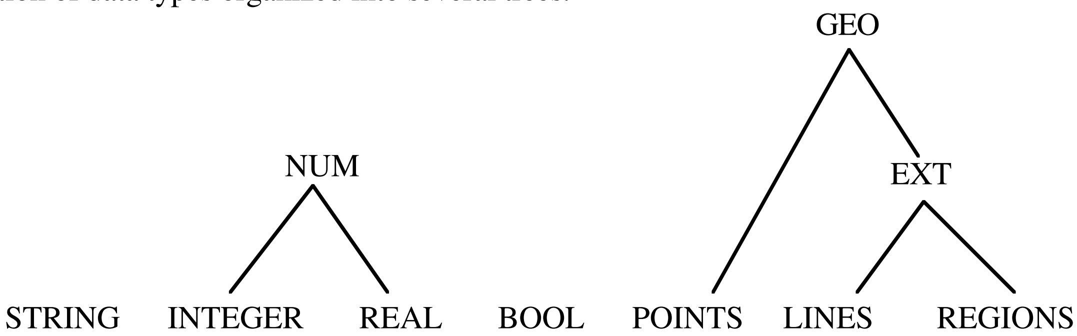

Figure 1

Here, for example, INTEGER is a subtype of NUM (INTEGER ≤ NUM) and *lub*(POINTS, LINES) = GEO. Each data type has an associated domain of values given by a function *dom* (e.g. *dom*(BOOL) = {true, false}). If $a \leq b$ then $dom(a) \subseteq dom(b)$. The purpose of the data type hierarchy is to allow polymorphic functions to be defined, for example, an intersection test can be applied to any two geometric values in EXT.

2.2.2 Object Types

There is a finite set $(OT, \leq)$ whose elements are called *object types* with a (tree) partial order "≤". We will see below that there is a one-to-one correspondence between object types and classes in the database; in fact, an object type is nothing else than the name of a class. The partial order on object types corresponds to the class hierarchy. Similarly as for data types, two object types c, d may be related (belong to the same tree) in which case the smallest common supertype, $lub(c, d)$, is well-defined.

Each object type has an associated set of object identifiers which is a subset of a set of object identifiers *OID* (which contains the identifiers of all objects created so far). This set is given by a function $oids$: $OT \rightarrow \mathbf{P}(OID)$, where $\mathbf{P}(X)$ denotes the power set of X. If $c \leq d$ $(c, d \in OT)$, then $oids(c) \subseteq oids(d)$. On the other hand, if c and d are not related, then $oids(c) \cap oids(d) = \emptyset$. For an object type c, $oids(c)$ contains precisely the identifiers of objects created so far in the corresponding class c.

Given an object identifier, we can determine its *immediate type* (the smallest object type in the hierarchy that it belongs to) by a function *itype*: $OID \to OT$ defined by:

$$itype(o) = c \;\Leftrightarrow\; (o \in oids(c) \;\wedge\; \forall\, d \in OT\colon\; o \in oids(d) \;\Rightarrow\; c \leq d)$$

2.2.3 Tuple Types

Let A be a set whose elements are called *attributes* – a domain of attribute names that can be used in forming tuple types. The set of *tuple types*, denoted TT, is defined as follows:

$$TT \;=\; \{<(a_1, t_1), \ldots, (a_m, t_m)> \mid m \geq 0, \forall i \in \{1, \ldots, m\}: a_i \in A, t_i \in D \cup OT\}$$

That is, each tuple type in TT is a list of pairs where each pair contains an attribute name together with either a data type or an object type. The empty list <> is also a tuple type.

For each tuple type, there is a corresponding domain of *tuple values* defined as follows. Let $T \in TT$, $T = <(a_1, t_1), \ldots, (a_m, t_m)>$.

$$values(T) = \bigcup_{i=m}^{\infty} V_i$$

where $V_i = \{<(v_1, u_1),\; (v_2, u_2), \ldots, (v_i, u_i)> \mid \forall j \in \{1, \ldots, i\}\colon\; u_j \in D \cup OT$

$$\wedge \quad u_j \in D \;\Rightarrow\; v_j \in dom(u_j)$$

$$\wedge \quad u_j \in OT \;\Rightarrow\; v_j \in oids(u_j)$$

$$\wedge \quad j \leq m \;\Rightarrow\; u_j \leq t_j \;\; \}$$

In other words, a tuple value is also a list of pairs of some length i which must be at least m. Each pair is a “typed value”, a value annotated with its type, and the value must belong to the corresponding data or object identifier domain. Furthermore, within the first m components the type in the tuple value must be a subtype of the corresponding data or object type in the tuple type T.

To summarize, a tuple value matches a tuple type as long as its values in the first m components belong to subtypes of those of the tuple type. Nothing is said about the remaining tuple components.

The purpose of this definition is to allow for a “dynamic generalization” of collections of tuples. We will be able to form in queries any union of sets of tuples; for the resulting set, a new tuple type will be automatically derived such that all tuples in the union match this new type. The basis for this is the following definition. Let $T = <(a_1, t_1), \ldots, (a_m, t_m)>$, $U = <(b_1, u_1), \ldots, (b_n, u_n)>$ be two arbitrary tuple types. Their *smallest common supertype* is

$$lub(T, U) := <(a_1, lub(t_1, u_1)), \ldots, (a_k, lub(t_k, u_k))>$$

where $k \in \{0, \ldots, \min(m, n)\}$ such that for $1 \leq i \leq k$, t_i and u_i are related, and either $k \in \{m, n\}$, or t_{k+1} and u_{k+1} are not related. In other words, we take the longest common prefix of related types and within it for each pair of types their smallest common super data or object type. Of course, the result may be the empty tuple type. Note that attribute names are taken from the first operand, so the operation is not commutative.

2.3 Classes and Database

A *database* is a pair $(C, \leq)$ where C is a finite set of *classes*, and "$\leq$" ("subclass") a tree partial order on C . A *class* $c \in C$ is a pair (*ctype*(c), *extension*(c)).

The set of classes C is partitioned into three subsets: $C = SC \cup LC \cup PC$. Classes in SC, LC, and PC are called *simple classes*, *link classes*, and *path classes*, respectively. The subclass partial order respects this partition, that is,

$$a \leq b \;\Rightarrow\; \{a, b\} \subseteq SC \;\vee\; \{a, b\} \subseteq LC \;\vee\; \{a, b\} \subseteq PC$$

The two components of a class, its *type* and its *extension*, are different for simple classes, link classes, and path classes. Informally, the type defines the structure of objects in the class, and the extension the collection of objects currently contained in it. In the following subsections we describe type and extension for the three kinds of classes, relating them to corresponding data definition commands. Creation of subclasses is treated in Section 2.4.

2.3.1 Simple Classes

A simple class is created by a command of the form

```
<class creation> ::= create class <class-name> [ = <attribute-list>] ;
<attribute-list> ::= <attr-name> : <type>
                   | <attr-name> : <type> , <attribute-list>
```

The *type* of a simple class is a pair (c, T), if it was created by a command

```
create class c = T;
```

where c is the class name used in the definition and T the tuple type corresponding to the attribute list. If the optional clause is omitted, then the tuple type is the empty type <>. For brevity, we will speak in definitions simply of a class (c, T) instead of "a class with type (c, T)". As mentioned in Section 2.2.2, there is a one-to-one correspondence between classes and object types. Hence, the class creation command creates at the same time a new object type $c \in OT$.

The *extension* of a simple class (c, T) is a subset of $oids(c) \times values(T)$, that is, a set of pairs consisting of an object identifier and a tuple value. Object identifiers are all distinct:

$$\forall\ (o_1, t_1), (o_2, t_2) \in extension(c) :\ \ o_1 = o_2 \;\Rightarrow\; t_1 = t_2$$

2.3.2 Link Classes

A *link class* is created by a command of the form

```
<class creation> ::= create link class <class-name> [ = <attribute-list>]
     from <class-name> to <class-name> ;
```

The *type* of a link class is a quadruple (c, T, d, e) if it was created by a command:

```
create link class c = T from d to e;
```

Here d and e must be the names of simple classes. The *extension* of a link class (c, T, d, e) is a set of quadruples which is a subset of $oids(c) \times values(T) \times oids(d) \times oids(e)$. Again, all object identifiers in the first components of the quadruples must be distinct, and class creation includes creation of a corresponding object type.

2.3.3 The Database Schema and Instance Graphs

Before we can define path classes, we need to understand the graph structure created by a collection of simple classes and link classes, which consists of a *database schema graph* and a *database instance graph*. We generally describe graphs as two sets (nodes and edges) together with two mappings *source* and *target* from the edges into the nodes. This is because multiple edges between the same two nodes are allowed in our data model.

The *database schema graph* is $SG = (S, L, source, target)$, where

(i) $S = \{\ c \mid ((c, T), ext) \in SC\}$

(ii) $L = \{\ c \mid ((c, T, d, e), ext) \in LC\}$

(iii) $source$: $L \rightarrow S$ is defined by

$source(c) = d \quad \Leftrightarrow \quad ((c, T, d, e), ext) \in LC$

(iv) $target$: $L \rightarrow S$ is defined by

$target(c) = e \quad \Leftrightarrow \quad ((c, T, d, e), ext) \in LC$

So for each simple class and each link class there is one node and one edge in the schema graph, respectively. These nodes and edges are also the object types corresponding to the respective classes.

A *path type* is a quadruple (G, μ, s, F) where

(i) $G = (V, E, source, target)$ is a connected graph.

(ii) μ: $V \cup E \rightarrow OT$ is a function labeling nodes and edges of G with object types such that

(a) $v \in V \ \Rightarrow \ \exists\, ((c, T), ext) \in SC: \ \ \mu(v) = c$

(b) $e \in E \ \Rightarrow \ \exists\, ((c, T, a, b), ext) \in LC$:

$\mu(e) = c \ \wedge \ \mu(source(e)) = a \ \wedge \ \mu(target(e)) = b$

(iii) $s \in V$ (the *start node*)

(iv) $F \subseteq V$ (the *final nodes*)

Basically, a path type is nothing else than a finite automaton belonging to a regular expression over link class names (used, for example, in the data definition command for a path class), s is the start state, F the set of final states. The labeling function μ ensures consistency with the database schema graph. Each path in G from start node s to some node in F describes a corresponding set of paths in the database instance graph, defined below. In Figure 2 the path type corresponding to the regular expression “section+” from path class *highway* (Section 2.1.2) is shown.

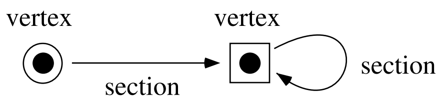

Figure 2

Path types are used in the definition of path classes, but also in queries, where graph traversal can be restricted to graphs of a desired form.

The *database instance graph* is $IG = (S, L, source, target)$, where

(i) $S = \{ o \mid \exists\, c \in SC\colon (o, t) \in extension(c)\}$

(ii) $L = \{ o \mid \exists\, c \in LC\colon (o, t, p, q) \in extension(c)\}$

(iii) $source\colon L \rightarrow S$ is defined by

$source(o) = p \;\Leftrightarrow\; \exists\, c \in LC\colon (o, t, p, q) \in extension(c)$

(iv) $target\colon L \rightarrow S$ is defined by

$target(o) = q \;\Leftrightarrow\; \exists\, c \in LC\colon (o, t, p, q) \in extension(c)$

So the nodes and edges of this graph are object identifiers of objects in simple and link classes, respectively.

A *path over IG* (also called a *database path*) is a sequence of object identifiers

$$p = \langle v_0, e_1, v_1, \ldots, v_{n\text{-}1}, e_n, v_n \rangle$$

where for $0 \le j \le n$: $v_j \in S$ and for $1 \le j \le n$: $e_j \in L$, $source(e_j) = v_{j\text{-}1}$, and $target(e_j) = v_j$. This path *matches* a path type (G, μ, s, F) iff there exists a path $P = \langle V_0, E_1, V_1, \ldots, V_{n\text{-}1}, E_{n\text{-}1}, V_n \rangle$ in G such that:

(i) $V_0 = s$

(ii) $V_n \in F$

(iii) For $0 \le j \le n$: $itype(v_j) \le \mu(V_j)$ (*itype* yields the immediate type of object identifier v_j)

(iv) For $1 \le j \le n$: $itype(e_j) \le \mu(E_j)$

In other words, the database path p must have a corresponding path P in G such that each object in the path p is of a subtype of the one required in P. We denote by $paths(G, \mu, s, F)$ the set of all database paths matching path type (G, μ, s, F).

2.3.4 Path Classes

A *path class* is created by a command of the form:

```
<class creation> ::= create path class <class-name> [ = <attribute-list>]
     as <link-expression>
<link-expression> ::=  <class-name>+ | <class-name>
     | <link-expression> <link-expression>
```

```
| (<link-expression> or <link-expression>) | (<link-expression>)*
```

Essentially a link expression *LE* is a regular expression over class names which must belong to link classes. The regular expression must be chosen in such a way that it defines a connected graph, that is, a path type. The correspondence between regular expressions and path types is straightforward. If a path class was created by a command

```
create path class c = T as LE;
```

then its *type* is the triple $(c, T, (G, \mu, s, F))$ where (G, μ, s, F) is the path type corresponding to *LE*. The *extension* of class $(c, T, (G, \mu, s, F))$ is a set of triples subset of $oids(c) \times values(T) \times paths(G, \mu, s, F)$.

2.4 Subclasses

One can create subclasses of simple classes, link classes, and path classes. In all three cases the tuple type of the subclass will be a subtype of the tuple type of the superclass. In link classes it is additionally possible to refine the source and target object types. In this section we briefly describe data definition commands for the creation of subclasses and discuss the relationship of subclasses with the database graph.

2.4.1 Subclasses of Simple Classes

A subclass of a simple class may be created as follows:

```
<class creation> ::= create <superclass-name> class <class-name>
     [ <attribute-clause> ] ;
<attribute-clause> ::=   = <attribute-list> | with <attribute-list>
```

Let *s* be a simple class of type (s, U). As a result of the command

```
create s class c;
```

there is a new class (c, U) and for the two object types *s* and *c* holds $c \leq s$. If the command is

```
create s class c = T;
```

then *T* must be a subtype of *U*, the type of *s*. This form allows one in particular to refine in *T* components that are already present in *U* (for example, specializing an EXT type in *U* into a REGIONS type in *T*). If the command is

```
create s class c with T;
```

then the resulting class has type $(c, U \bullet T)$ where the symbol "•" denotes concatenation of tuple types. By this concept, we have introduced only simple inheritance; multiple inheritance is excluded for simplicity. The general meaning of the class hierarchy is that we can substitute everywhere instances of subclasses for instances of superclasses. For example, if a tuple type has a component (a, d) where *d* is an object type (a class name), and $c \leq d$, then an attribute value in a corresponding tuple may be an instance of *c*. Similarly, if a link class has type (c, T, d, e), then an edge object may also connect two (node) objects of classes d' and e' if $d' \leq d$ and $e' \leq e$.

2.4.2 Subclasses of Link Classes

A specialization of a link class may be introduced as follows:

```
<class creation> ::= create <superclass-name> link class <class-name>
    [ <attribute-clause> ] [ from <class-name> to <class-name> ];
```

The meaning of the three variants of attribute specification (including omitting it) is the same as for subclasses of simple classes. The source and target specification (from … to …) is also optional. If it is omitted, the source and target classes of the superclass are used. For example, let (s, U, d, e) be the type of the superclass in the command:

```
create s link class c = T;
```

Then it is required that $T \leq U$ (attributes in U are refined) and the type of the resulting link class is (c, T, d, e). Hence an edge object of class c can be substituted for an edge object of class s. In the command

```
create s link class c with T from d' to e';
```

it is required that $d' \leq d$ and $e' \leq e$, and the resulting link class has type (c, $U \bullet T$, d', e'). Again an edge object of class c can be substituted for an edge object of class s.

2.4.3 Subclasses of Path Classes

For a path class, a specialization can only change the tuple type from the superclass to the subclass.

```
<class creation> ::= create <superclass-name> path class <class-name>
    [ <attribute-clause> ] ;
```

The three variants of attribute specification are the same as before. The path type of the superclass is inherited. So far, there is no specialization of path types.

The formal definition of the data model and the description of the data definition language are now complete. In the remaining two subsections of Section 2 we discuss modeling options and introduce a larger example.

2.5 Modeling with Link Classes and Object-Valued Attributes

The data model offers two different features for modeling “connections” between objects, namely *link classes* and *object-valued attributes*. Why do we need both? Are they different in “expressive power”? Which should be used when?

In terms of the entity-relationship model, an object-valued attribute can express directly only a *binary many-one relationship without attributes* (of the relationship). A binary many-one relationship *with attributes* can (although a bit awkwardly) be expressed by putting the attributes into the “source object” of the connection, the one which has the object-valued attribute.

In contrast, a link class can directly express a binary many-many relationship with attributes. For the representation of spatially embedded networks we need many-many relationships, and it is crucial to have “edge attributes”, in particular, to associate LINE values describing the geometry of the edge, as in the highway network (Section 2.1.2). Furthermore, it is possible to define path classes over link classes. The latter facility is mainly useful in those cases when link classes define cycles in the database schema graph so that paths of varying length exist. For example, consider the ER-diagram in Figure 3 (where *cb_length* stands for “common boundary length”):

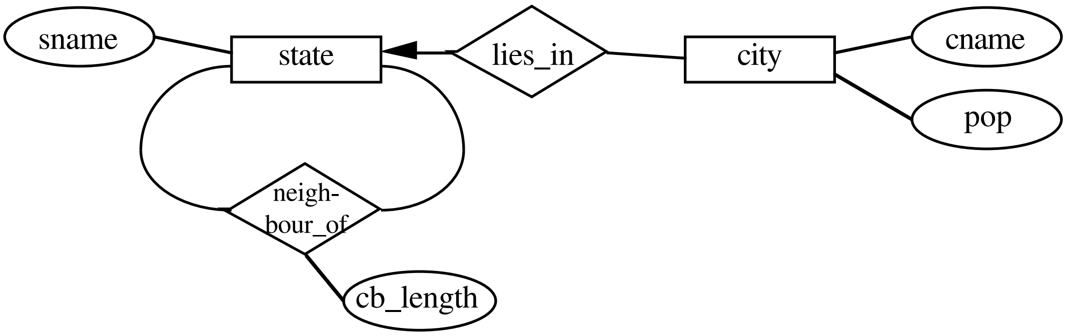

Figure 3

This could be represented in our model by class definitions (from now on we leave out the “create” when describing a database):

```
class state = sname: STRING;
link class neighbour_of = cb_length: REAL from state to state;
class city = cname: STRING, pop: INT, lies_in: state;
```

Clearly, link classes offer the most powerful and for our applications crucial modeling facility. On the other hand, the use of object-valued attributes, whenever they can be used, simplifies a database schema (less classes are needed); additionally, it is possible to achieve some substructuring of a complex graph if we perceive link classes as “stronger” connections than object-valued attributes (OVAs). A good example for this is the model of the public transport network in the next subsection where the three levels of networks can be somewhat separated by using OVAs for the inter-level-connections. Hence the recommendation for modeling is:

- Use a link class if the relationship is involved in a cycle.
- Use a link class if it is desired to create path classes involving it.
- Use a link class if the relationship is many-many.
- Otherwise, an object-valued attribute may be used.

The possible need of an explicit modeling of relationships in OO models has also been observed in the object-oriented literature (e.g. [Di90]). For a deeper discussion and a specific proposal see [AlGO91].

2.6 The Public Transport Network

In this subsection we introduce a larger example which may give a better impression of the kind of applications this data model (and database system) is intended for. We shall also show some queries for this example in the next section. The application domain to be represented is public transport, e.g. bus, tram, or train lines and schedules.

On closer inspection, this application is more complex than one might have expected. One can distinguish three levels of network, describing *the physical network*, *lines*, and *time schedules*. The lowest level represents the geometry of the network used for traveling. For a railway network, at this level we find rails and switches, switches being the nodes and rail sections the edges of the graph. For a bus network, this physical part would consists of streets and street crossings and junctions. We call paths over this level *physical routes*. This level is modeled as follows:

```
class vertex = pos: POINT;
link class arc = route: LINE from vertex to vertex;
path class phys_route as arc+;
```

The next level introduces regular connections over the physical network usually called *lines*, for example, bus or underground lines traversing a certain path of the physical network. A line may be identified with a number or by giving the names of final destinations at both ends (e.g. the Metro in Paris), and it contains a list of stops that we call *stations*. (A line is what is usually depicted on the wall within a bus or underground carriage.) Note that this level does not yet contain the time schedule for trips over lines.

```
class station = name: STRING, loc: vertex;
link class connection = travel_minutes: INT, way: phys_route from station
     to station;
path class line = line_type: STRING, line_no: INT as connection+;
```

Observe that this second level contains references to the first level, the physical network, associating *stations* by attribute *loc* with their physical positions (assuming that for each station a vertex has been established) and *connections* as a piece of the line between two stations by attribute *way* with a corresponding path over the physical network. *Lines* are paths of this level; the *line_type* attribute may be used to distinguish types of connections, e.g. fast long distance trains from slow local trains in a railway network.

The third level contains the actual time schedules. We model this as a collection of *departure* and *arrival events*, which will be the nodes of the third level graph. A departure event, for example, says that at a certain time a carrier (e.g. a train) of a specified line departs from a given station. A specific trip of a train over a line then corresponds to an alternating sequence of departure and arrival events (Figure 4).

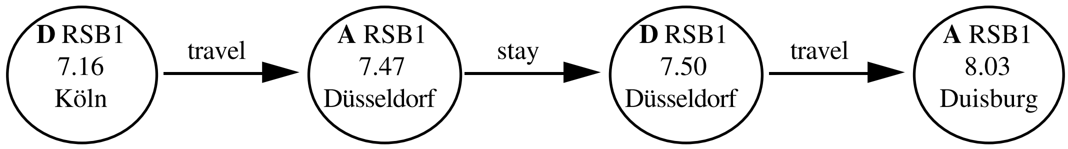

Figure 4

In Figure 4 **D** and **A** stands for departure and arrival, respectively; “RSB1” is the name of a particular line. The event nodes of a specific trip are connected by *travel* and *stay* edges. On the other hand, we can change at a station from one line to another (more precisely, from a trip of one line to a trip of another line). To model this, the arrival and departure events at one particular station are connected by *change* and *wait* edges, as shown in Figure 5.

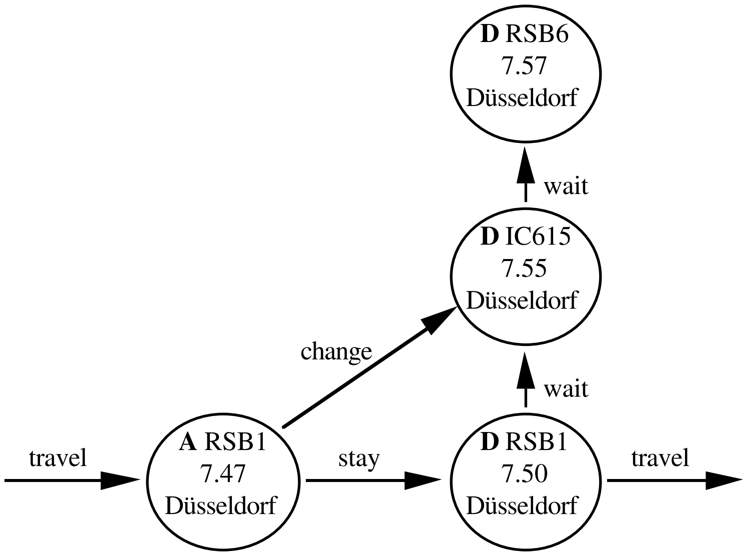

Figure 5

The idea is that a *change* edge connects an arrival event with the next departure one can reach in time at this station, and that all departure events are linked in the order of departure time. Hence changing at a station can be described by a sequence of change and wait edges of the form “change wait*” (which is a path type). A complete trip of a traveler with possibly several changes of trains has a path type

travel (stay travel)* (change wait* travel (stay travel)*)*

whose graph representation is shown in Figure 6.

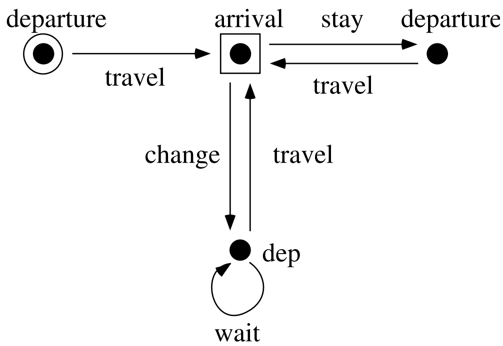

Figure 6

Hence the third level is modeled as follows:

```
class event = time: INT, at_station: station, of_line: line;
event class arrival, departure;
link class travel = through: connection from departure to arrival;
link class stay from arrival to departure;
link class change from arrival to departure;
link class wait from departure to departure;
path class trip as travel (stay travel)*;
```

A graphical representation of this rather complex database schema is given in Figure 7.

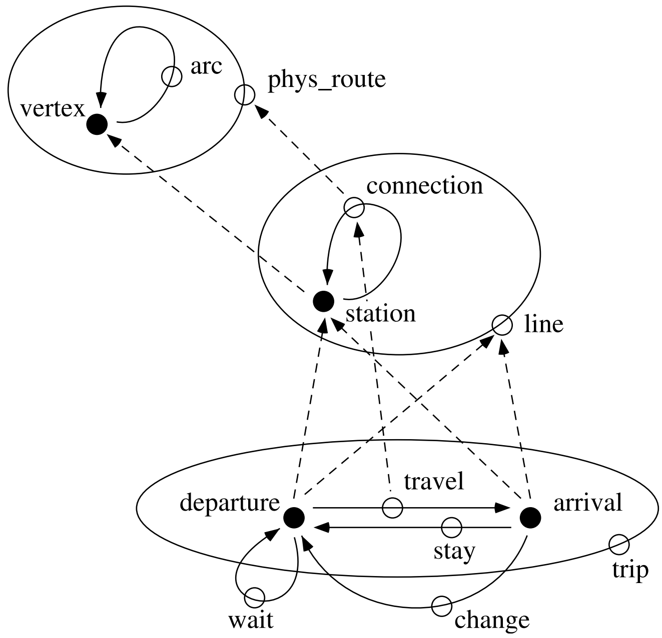

Figure 7

Here a path class is represented as a circle around its participating simple and link classes (omitting the more precise information in the path type); object-valued attributes are indicated by dashed lines.

3 Queries

In this section we discuss the concept of a query on a graph database, explain the structures the user manipulates in queries, and show some fundamental tools (statements, operations) for querying. We do not yet develop a complete query language but rather introduce some core elements. Also at this stage the semantics of operations are only described informally.

3.1 What is a Query?

Before we can look at specific operations for querying we need to understand what a query is in general. The issue arises because we would like to be able to modify in some sense the database graph in a query, for example, to add some edges computed by a query expression and then to apply a graph operation traversing old as well as new parts of the database graph. On the other hand, it should also be possible to restrict the graph for consideration in a query.

In the relational model, a query can be viewed as a function q which maps a database D (a set of relations) into a relation T. The query function q is a composition of simpler functions. There are basic functions accessing the database; writing the name of a relation maps the database into this relation. There are further functions (relational algebra) mapping relations into other relations. Query q leaves D unchanged; further operations can involve D (again, by writing the names of relations) as well as T (by applying them to expression q).

The problem is that we want to define operations that work *implicitly* on the whole database graph. If parts of the graph structure are added in a query, it must be clear for those operations whether they apply to the graph structure *before* the temporary addition or *with* the addition. Therefore we use the following concept of a query. A query Q may consist of several *steps*, $Q = q_1; \ldots; q_m$. Each step may compute one or more classes of simple, link, or path objects. After each step, these classes are added to the database (and so, implicitly, extend the database graph). Or a step may express a restriction of the database graph for the following steps. Hence a graph operation used in step q_j “sees” the graph with the changes computed in steps $q_1, \ldots, q_{j-1}$. Examples of such *multistep queries* are given below.

3.2 Structures

What kind of structures at the conceptual level does a user create and manipulate in queries? Candidates might be graphs, sets of objects, nested relations, lists of object identifiers, etc. At a more formal level this is a question about the user's type system. The design goal is to keep this collection simple but sufficiently expressive. It turns out that for our model four kinds of structures/objects suffice:

- a uniform sequence of objects
- a heterogeneous sequence of objects
- a (single) object
- a value of a data type

A *uniform sequence of objects* (USEQ) contains a set of objects, usually from a single simple class, link class, or path class (that is, in general a subset of the extension of the class) in some, not necessarily specified, order. More precisely, the objects in the sequence may come from different classes and so have several different object types, but are all viewed under only *one* common tuple type. Hence, for each object in the sequence, its tuple component must be a value of that tuple type.

We use sequences rather than sets because it is then possible to offer operations in the query language making use of the order such as sorting, or taking head or tail of a sequence (see [GüZC89, MaV93] for a more detailed discussion of this point). Such a sequence of objects is the basic structure in formulating queries; since each object contains a tuple, it is the equivalent of a relation in the relational model. The most simple way to obtain a uniform sequence is just to write the name of a class:

$$\rightarrow \text{USEQ} \quad \textit{classname}$$

This notation indicates that writing the name of a class can be viewed as a function without arguments (implicitly applied to the database) returning a value of a type “uniform sequence”. The meaning of writing the classname is that a sequence of all the objects in the extension of the class (in no particular order) is created. For example, writing “person” (database from Section 2.1.1) yields a sequence containing all person objects.

A *heterogeneous sequence of objects* (SEQ) may contain objects from several classes. These objects may have several different object as well as tuple types. For example, the node and edge objects forming a path in the database may be given as such a sequence. But more generally, heterogeneous collections of objects can be formed in queries and be manipulated in this form. The basic way to obtain a heterogeneous sequence is to write the name of several classes in angular brackets:

$$\rightarrow \text{SEQ} \quad <\textit{classname}_1, \ldots, \textit{classname}_n>$$

Again, the resulting sequence contains the objects of all mentioned classes in unspecified order. For example,

```
<book, person, wrote>
```

yields the corresponding objects from three different classes (Section 2.1.1).

What can one do with such a “mixed” collection of objects? First, there is a specialized and very powerful tool in the query language, called the *rewrite* operation, to deal with such sequences (Section 3.3.2). Second, in the same way as we have interpreted the set of simple, link, and path classes making up a database as a *database graph*, we will be able to interpret such collections as *graphs* or even as new parts of the database graph that are added in a query. Therefore, no specific types for graphs are needed in the user's conceptual model for querying. Third, one can apply a union operation in the query language to a heterogeneous sequence and so transform it into a uniform sequence (Section 3.3.3).

Single objects are also needed in some cases. For example, an operator for searching a shortest path needs a start and a target object as operands. The basic technique for determining a single object in the database is called *object identification*. The syntax for this is as follows:

```
<object identification> ::= <classname> ( <value-list> )
<value-list> ::= <attribute-value> | <attribute-value> , <value-list>
<attribute-value> ::= [<attribute-name> =] <value>
```

The basic idea is that the value-list is a prefix of the attribute values of the object to be identified, such that only a single object has this prefix. For example, to identify a state object (Section 2.1.1), one can write:

```
state("Germany")
```

It is also possible to use uniquely identifying attribute values that are not a prefix of the object's attributes; in that case one must write the attribute names before the values. Of course, there are also other ways to obtain objects, for example, evaluating an object-valued attribute of another object:

```
person("Gueting").country
```

Finally, *values of data types* may occur, for example, by writing constants (in particular, for strings and numbers), by accessing attributes, or by evaluating operations defined for these data types (e.g. intersection of two LINES values may yield a POINTS value).

3.3 Tools for Querying

The fundamental tools for querying a graph database are:

- The *derive statement*, which takes the role of the classical *select … from … where*, but has an extended meaning for graphs; it includes the functionalities of selection, join, projection, and function application;
- the *rewrite operation* as a basic tool for the manipulation of heterogeneous sequences; it allows to replace objects or subsequences by other (new) objects;
- the *union operation* for achieving “dynamic generalization”, that is, for transforming a heterogeneous collection of objects into a homogeneous one, viewed under a common super tuple type;

- a collection of graph operations, e.g. *shortest path search* and *circle* (determining a subgraph within a certain radius from a given node).

Additionally, the query language will contain further operations, e.g. for sorting, grouping, aggregate functions, data type operations, etc. In the following subsections we explain the four main tools listed above.

3.3.1 Derive

The *derive statement* is the most fundamental tool in the query language. Perhaps the best way to introduce it is to show a few examples. The first refers to the standard application from Section 2.1.1:

Q1. List the titles of all books written (coauthored) by Hopcroft in 1983!

```
on person wrote book
where person.name = "Hopcroft" and book.year = 1983
derive book.title
```

Here the on-clause says that each combination of *person*, *wrote*, and *book* objects should be considered where the *person* is connected by the *wrote* link to the *book*. From this collection of triples of objects the where-clause selects those fulfilling the two conditions. The derive-clause creates for each selected triple a new object with a single attribute called *title* whose value is taken from the attribute *title* of the *book* object in the triple. In this case, simple objects of an unnamed object type are created. It is also possible to create link objects, for example; other cases and the semantics in general are described below.

The second example query is based on the public transport database from Section 2.6.

Q2. Make a listing of all departures from Dortmund main station in the form:

Time of departure	Type and number of train	End station and arrival time
6.13	IC 615	München 14.23
6.22	D 308	Wiesbaden 12.18

…

The time of departure can be found within a departure event. Type and number of train correspond to a line type and a line number. The name of the final destination is either in the last node of a line path or can be found from the last event of a trip path. However, only the last event of a trip also contains the arrival time. The query can be formulated as follows:

```
on departure at_station station, departure of_line line, departure in trip
where station.name = "Dortmund
derive departure.time, line.line_type, line.line_no, (trip
     end).at_station.name, (trip end).time
```

Here in the on-clause all combinations of *departure* events, *stations*, *lines* and *trips* are formed where (i) the *departure* object is connected through its object-valued attribute *at_station* with the *station* object (that is, *departure.at_station* = *station*), (ii) the *departure* object is connected through attribute *of_line* with the *line* object, and (iii) the *departure* object is a node in the path of the *trip* object. Note that in queries object-valued attributes can be used quite in the same way as link objects. The only difference is that these connections cannot have their own attributes, so one cannot access attributes of *at_station* or *of_line*. There is nothing new in the where- and derive-clauses except of the use of a function *end* to get from a path object its last node object.

Let us now consider syntax and semantics of the derive-statement in general. It has the following form:

```
<derive-statement> ::= [ <range declarations> ] on <subgraph-spec> [ where
     <condition> ] derive <object-spec>
```

Range declarations are only needed to feed into a derive statement the result of an expression. They have the following form:

```
<range declarations> ::= <range>, | <range>, <range declarations>
<range> ::= range of <newname> is <expression>
```

Each range declaration introduces a variable with name *newname* which ranges over a uniform sequence computed in the *expression*. An example is given in Section 3.3.3. Usually, variables are introduced directly within the subgraph specification in the on-clause.

A *subgraph specification* has the form:

```
<subgraph-spec> ::= <pattern> | <pattern> , <subgraph-spec>
<pattern> ::= <variable-intro>
     | <simple-variable-intro> <link-variable-intro> <simple-variable-intro>
     | <simple-variable-intro> in <path-variable-intro>
     | <link-variable-intro> in <path-variable-intro>
     | <variable-intro> <attribute-name> <variable-intro>
<variable-intro> ::= <object-type> | <object-type>(<newname>)
```

A subgraph specification is a list of *patterns*. A pattern introduces either just a single variable or it connects two or three variables in various ways, requiring that

- a simple object is connected through a link object to another simple object,
- a simple object occurs as a node within a path object,
- a link object occurs as an edge within a path object, or
- a simple object has another simple object as an attribute value.

A variable is *introduced* by either writing the name of an object type (which is in the derive statement the same as a class name) – in this case the class name is used directly as a variable. The other possibility is that a new name for the variable is introduced explicitly, for example, in the form "*state*(*s*1)". This is needed when several variables range over the same class. Alternatively it is possible to use range declarations to introduce explicit variables.

As mentioned before, in the evaluation of the on-clause all possible assignments of objects to the variables will be considered and those tuples of objects be determined that are simultaneously consistent with all patterns. In general, the patterns in the on-clause describe one or more connected graphs (if we draw an edge between two variables linked in a pattern). If two or more graphs are present, it means that the cartesian product of the possible object tuples for each graph will be formed. Therefore, if each pattern is just a variable name, then we have the classical cartesian product operation or a join, if there are connecting conditions in the where-clause. If there is only one pattern which is a single variable, then we have a simple selection, for example:

Q3. Find the books published by Addison-Wesley!

```
on book
where publisher = "Addison-Wesley"
derive book
```

An example of a join expressed by the derive statement can be found below. The where-clause contains just a condition, used to filter tuples of objects coming from the on-clause. The derive-clause specifies how a resulting set of objects is to be formed, in the following form:

```
<object-spec> ::= <variable> | [ <newname> = ] <attribute-spec-list>
     | <newname> [ = <attribute-spec-list>] from <variable> to <variable>
<attribute-spec-list> ::= <attr-spec> | <attr-spec> , <attribute-spec-list>
<attr-spec> ::= <variable>.<attr-name> | <newname>: <expression>
```

The object-specification can either be one of the variables of the derive statement, which means that these objects are put into the result sequence (as in Q3); so the whole statement amounts to a more or less complex selection. The other case is that new objects are formed. For these a new object type (name) may be given by the "<newname> =" part; if it is omitted, a name will be selected internally by the system (which is obviously unknown to the user and can therefore not be used in the rest of the query). Next, attributes for the new objects are defined. The first form is "<variable>.<attr-name>" in which case the new attribute name as well as the value is taken from the object denoted by the "<variable>". The other possibility is to explicitly introduce an attribute name and to assign it by an arbitrary expression a value of a data or object type. So far, if the from-to-part is omitted, simple objects will be created. If the from-to-part is present, then the variables must refer to simple classes. In this case link objects are created connecting the corresponding pairs of objects assigned to the from- and to-variables. Creation of path objects in the derive statement has so far not been provided in the design.

The following example illustrates several points, namely the use of explicit variables, the formulation of a classical join in the derive statement, the creation of link objects, and the dynamic modification of the database graph in a multistep query, as discussed in Section 3.1. We assume the following database to be given:

```
class state = sname: STRING; region: REGIONS;
```

The query is:

Q4. How many countries must be traversed traveling (by land) from Germany to China?

```
on state(s1), state(s2)
where s1.region adjacent s2.region
derive neighbour_of = cblength: length(common_border(s1.region,
     s2.region) from s1 to s2;

state("Germany") state("China") shortest_path[neighbour_of+]
rewrite[state -> , neighbour_of -> neighbour_of] count
```

This is a multistep query; the first step is the derive statement which constructs a set of *neighbour_of* edges and adds them to the database graph; the second step uses these edges to find a shortest path from Germany to China. Here we only discuss the derive statement, the second step will be explained below when *rewrite* and *shortest-path* operations have been introduced. Range declarations are used because two variables range over the same class. The patterns in the on-clause describe two independent graphs. So a cartesian product is formed which together with a subsequent selection condition amounts to a join (*adjacent* is a geometric predicate applicable to two REGIONS values). The derive-clause creates a new link object class *neighbour_of* between any two qualifying *state* objects; as an attribute of such a link the length of the common boundary is computed (using two geometric data type operations *length* and *common_border*). The resulting *neighbour_of* class has the same definition (or type) as the one in Section 2.5.

The derive statement as a whole can be viewed as an algebraic operation with the following signature:

$\rightarrow$ USEQ   *derive*

So it is an operation without arguments (the database always being an implicit argument) which returns a uniform sequence of objects. Hence, within a query one can put a derive statement whenever a uniform sequence of objects is allowed and can further manipulate its result sequence, for example, apply a sorting operation.

3.3.2 Rewrite

The *rewrite* operation is a very powerful tool for dealing with heterogeneous sequences and in particular, to manipulate paths (which are heterogeneous sequences). One simple way to use it is similar to case-statements in programming languages, since one can specify a treatment separately for each object type that may come along in a heterogeneous sequence. But it is also possible to apply transformations to whole subsequences of a given sequence. In this case the order of elements in the sequence plays a crucial role; to understand this order, *path types* (or, more generally, *sequence types*) introduced in Section 2.3.3 are essential. Again, let us introduce the rewrite operation by a few examples. First, consider the second step of query Q4:

```
state("Germany") state("China")
shortest_path[neighbour_of+]
rewrite[state -> , neighbour_of -> neighbour_of] count
```

Here the *shortest_path* operator (whose arguments will be explained below) computes a shortest path from one state object (Germany) to another one (China). The result is a heterogeneous sequence of *state* and *neighbour_of* objects. The result has a *sequence type* (which is equivalent to a path type, but also mentions the types of node objects)

```
state neighbour_of state (neighbour_of state)*
```

We abbreviate this sequence type as SNS(NS)*. Such a sequence is input to the rewrite operation. It contains a list of *transformations*. Each transformation has a *left side* and a *right side* (separated by an arrow). The left side of a transformation is a *pattern* which is a list of one or more variables denoting object types. The right side is either an expression which must evaluate to an object, or empty.

The meaning is roughly that whenever a subsequence of objects is encountered matching one of the patterns, then the corresponding transformation is applied (this will be made more precise below). Hence in our example, the effect is that all objects of type *state* in the sequence are thrown away whereas all *neighbour_of* objects are moved unchanged into the result sequence. So rewrite can be used to realize a *type restriction* on a heterogeneous sequence.

Applying rewrite, one should keep track of the manipulation of the sequence type that it implies. In our example, the result sequence will have type NN*, that is

```
neighbour_of (neighbour_of)*
```

The second example is again based on the public transport database from Section 2.6.

> Q5. List all direct connections from Dortmund to München with the distance traveled. That means, provide a table of the form:

Departure time	Arrival time	Distance
6.13	14.23	610 kms
7.13	15.23	610 kms
7.43	16.26	578 kms

…

To answer this query, all levels of the public transport network are needed. On the level of schedules (trips) one must find trips containing departures and arrivals in Dortmund and München, respectively. These trips must be reduced to the part between Dortmund and München. To determine the distance traveled, the underlying physical network needs to be accessed and the lengths of arcs traversed be summed up. The query can be formulated as follows:

```
on departure at_station station(s1), arrival at_station station(s2),
     departure in trip, arrival in trip
where s1.name = "Dortmund" and s2.name = "Muenchen"
derive dtime: departure.time, atime: arrival.time, distance:
     trip suffix(departure) prefix(arrival)
     rewrite
          [departure -> , arrival -> , stay -> ,
          travel -> travel_dist = (dist:
               travel.through.way
               rewrite[vertex -> ,
                    arc -> arc_length = (len: length(arc.route))]
               sum[len])
          ]
     sum[dist]
order_by[dtime +]
```

This is already a fairly complex problem; it is still possible to formulate the query in a relatively concise way. The derive statement finds *trip* objects containing stops in Dortmund and München. In the derive-clause objects with three attributes are produced, called *dtime*, *atime*, and *distance*, where *distance* is computed as follows: Each trip path is reduced by operations *suffix* and *prefix* to the part between Dortmund and München. These are operations of the query language for the manipulation of sequences; the argument is besides the sequence an object, and the sequence is reduced to the part after and including the object in case of suffix, similarly the part before the object for prefix. An object of a path class can be treated directly as a sequence, hence these operations are applicable to *trip* objects. The remaining part of the trip sequence is handled by a rewrite: *departure*, *arrival*, and *stay* objects are thrown away; *travel* objects are transformed into new *travel_dist* objects with a single attribute called *dist*, whose value is computed as follows. From the *travel* object via its *through* attribute the underlying *connection* object is reached, from which via attribute *way* the corresponding *phys_route* path object is obtained. We are now at the level of the physical network. Here the path of the form

```
vertex arc vertex (arc vertex)*
```

is again treated by a rewrite; *vertex* objects are thrown away and for each *arc* object a new *arc_length* object with a single attribute *len* is created whose value is determined by applying a function *length* to the *arc* object's *route* attribute (of data type LINES). Hence the result of the inner rewrite is a uniform sequence of *arc_length* objects; *sum* is an aggregate function applicable to such a sequence. The result is a single number which is finally assigned to the *dist* attribute of the new *travel_dist* object. Again *sum* is applied to a uniform sequence of *travel_dist* objects to obtain a number which is then used as the *distance* attribute value in the objects created by the derive statement. In a final step, the sequence of (unnamed) objects returned from derive is sorted by departure time (*dtime*).

Let us now consider the syntax and semantics of the rewrite operation in general. The operand is a heterogeneous sequence; the argument in the square brackets is a *transformation-list*:

```
<transformation-list> ::= <transform> | <transform> , <transformation-list>
<transform> ::= <seq-pattern> -> <object-spec2>
<seq-pattern> ::= <variable-intro> | <variable-intro> <seq-pattern>
<object-spec2> ::= <variable> | <newname> = ( <attribute-spec-list> )
     | <newname> [ = <attribute-spec-list>] from <variable> to <variable>
```

The sequence pattern introduces a sequence of one or more variables in the same way as this is done in the derive statement – so it is possible to consider subsequences with several instances of the same object type and to refer to these instances by different variables. The object specification describes the construction of new objects in a similar way as in the derive statement; there are slight differences in that a name for the new object type is mandatory and that the list of attribute specifications is in parentheses if simple objects are constructed.

The semantics of the rewrite operation can be described as follows: The operand sequence is processed in a series of *steps*. In each step, a sequence pattern p in one of the transformations (let's say, transformation t) is matched against a prefix of the operand sequence. That prefix is then removed from the operand sequence and an object according to the object specification in t is constructed and put into the output sequence. To match the operand sequence with a pattern, the list of transformations $t_1, \ldots, t_m$ is considered sequentially. That is, first the pattern of t_1 is checked. Matching is complete when each variable in the pattern of t_1 has been assigned a corresponding element of the operand sequence. If this doesn't work, the next pattern is checked, and so forth. If none of the patterns matches, an error will be reported.

As a consequence, it is allowed that several patterns have a common prefix or that one pattern is a prefix of another; the order among them decides which is used.

The examples of rewrite that have been shown so far do not make use of the order of elements in a sequence. In closing this subsection, we consider an example of non-trivial use of this order and of the transformation of sequence types performed by rewrite. The task is to manipulate a shortest path found on the public transport network describing a traveler's trip (possibly including several changes of trains). Such a path had a path type (Section 2.6, also Figure 6)

travel (stay travel)* (change wait* travel (stay travel)*)*

which is equivalent to a sequence type (inserting the node types)

D T A (S D T A)* (C (D W)* D T A (S D T A)*)*

with D = departure, T = travel, A = arrival, S = stay, C = change, W = wait.

Q6. We want to produce a summary of such a path in the form of a short list with only one entry for each train used, for example:

Departure from station, time	Arrival at station, time
from Dortmund 11.42	to Köln 13.54
from Köln 14.04	to Bruxelles 16.51
from Bruxelles 17.05	to Paris-Nord 19.46

That means, we have to throw out from the path a lot of “uninteresting” arrivals and departures (and edges in between) to obtain a sequence which has only two elements for each line of the table above; from these two we should produce an object containing the four interesting pieces of information for the line.

There are two simple guidelines for “sequence rewrite programming”:

(1) Write the sequence type in a “normal form”: Always list first the components that are required to make up the sequence and then the repetitive structure. Apply this to the whole sequence as well as to subsequences.

(2) Choose sequence patterns for the rewrite in such a way that they do not cross parentheses. If crossing parentheses appears necessary, move them instead, applying the identity for regular expressions

$$(x\ \alpha)^*\ x = x\ (\alpha\ x)^*$$

where x is a symbol and α a string of symbols.

We have observed the first rule already in describing sequence or path types. Neglecting the second rule leads to rather error-prone rewrite programming. For our example, this means that we first transform the given sequence

D T A (S D T A)* (C (D W)* D T A (S D T A)*)*

into

D T (A S D T)* **A** (C (D W)* **D** T (A S D T)* **A**)*

Within this sequence, we are interested in the events shown in bold face. Hence we will throw out other subsequences to obtain a sequence of the form

D A (D A)*

In that sequence, we will replace each pair of departure and arrival event by new objects called *trip_segment*, so that the resulting sequence will be uniform and have type

trip_segment (trip_segment)*

Assuming that *found_path* denotes the input sequence, the query can be formulated as follows:

```
found_path
rewrite[travel -> , arrival stay departure travel -> , change -> ,
     departure wait -> , departure -> departure, arrival -> arrival]
rewrite[departure arrival ->
     trip_segment = (from: departure.at_station.name, dtime: departure.time,
          to: arrival.at_station.name, atime: arrival.time)]
```

The example illustrates that sequence rewrite programming may require a bit of training, but is also a quite powerful tool.

3.3.3 Union

The *union* operation makes it possible to transform a heterogeneous sequence of objects into a uniform one, so that all objects in the sequence are viewed under a common tuple type. It does that by computing the smallest common super (tuple) type for the tuple types of the heterogeneous sequence. Consider the following example database:

```
class city = name: STRING, region: REGIONS, pop: INTEGER;
class village = name: STRING, position: POINT, pop: INTEGER;
class river = name: STRING, way: LINE;
```

We can, for example, form the union of cities and villages:

```
<city, village> union
```

The result is a uniform sequence with a tuple type

<(name, STRING), (region, GEO), (pop, INTEGER)>

because this is the smallest common supertype of the tuple types of *city* and *village* objects (see Section 2.2). Note that *union* does not change the objects themselves; they are put, as they are, into the result sequence. Hence the objects keep their object type, object identifier, tuple value, etc., but the sequence as a whole has a new tuple type which can be used in the rest of the query, as in the following example.

Q7. List the names of all cities, villages, and rivers within Bavaria!

We assume that Bavaria has been introduced as the name of a REGIONS value.

```
range of cvr is <city, village, river> union,
on cvr
where cvr.region inside Bavaria
derive cvr.name
```

In this case the tuple type resulting from the *union* operation is <(name, STRING), (region, GEO)>. The geometric predicate *inside* has a signature GEO $\times$ REGIONS $\rightarrow$ BOOL, hence it is applicable to *region* attribute values of type GEO.

In general, the union operator produces:

- if the operands are all simple objects: simple objects
- if the operands are all link objects: link objects, if the source and target object types generalize (are related), otherwise simple objects
- if the operands are all path objects: simple objects
- a mixture of several kinds of objects (e.g. simple and link objects): simple objects.

This “dynamic generalization” feature is of particular importance for spatial databases where often collections of objects need to be formed that are just related by their spatial attributes (e.g. lie in the same area, intersect the same line, etc.). For more motivation, see [Gü91, ErG91].

3.3.4 Graph Operations

In a way, we arrive now at the main goal of the development of the GraphDB data model: to be able to formulate graph operations and to integrate them in a clean way into querying. This is possible because the database has a well-defined and explicit graph structure. In this section we do not yet describe a comprehensive collection of useful graph operations – this is a major task left to future work – as the purpose of this paper is to develop the right environment for the integration of such operations. But we show a few examples.

The first is an operation for finding shortest paths which has already been used in example query Q4. It has basically the following signature:

SIMPLEOBJ $\times$ SIMPLEOBJ $\rightarrow$ SEQ **shortest_path**

So it takes two simple objects, which are used as the start and target nodes of the search, respectively, and returns a shortest path from the start to the target node in the form of a heterogeneous sequence. The database graph as a whole is given as an implicit parameter. There are other parameters (syntactically given in square brackets behind the operator name)[2]:

- a *path type*, which on the one hand identifies those parts of the database graph that may be used in the search (including edges added in the query as in Q4) and on the other hand defines a precise structure for the resulting sequence which is then the basis for rewriting manipulations (Section 3.3.2),

---

[2] Whether these can be represented in the signature is mainly a matter of the expressiveness of the underlying type system formalism. These issues are discussed in [Gü93].

- for each class of edges (link objects) that may occur in the path according to the path type, a function assigning a cost to this edge (a mapping from the class into non-negative REAL values). The syntax for such function values is a variant of typed lambda calculus, as in [Gü93]:

  ```
  fun(n: neighbour_of) 1
  ```

  This denotes a function which assigns to a *neighbour_of* link object *n* the value of the following expression which in this case is always the constant 1. If such a function is not given as a parameter, a constant edge cost of 1 is assumed as a default.
- for each class of nodes (simple objects) that may occur in the path according to the path type, a function giving an estimated distance from this node to the target node (again, mapping into non-negative REALs). The reason this parameter is needed is that for the implementation of *shortest_path* the A* algorithm (see [Ni80]) will be used, which needs to estimate the distance from the target for nodes encountered in the search. In query Q4, there is no reasonable function to estimate the distance. In such a case a default function yielding constantly 0 is used, for Q4 this would be the function:

  ```
  fun(u: state, v: state) 0
  ```

  For A* to work correctly it is required that such a function must *underestimate* the distance to the target; with this function that is trivially the case. With such a trivial distance function A* reduces to Dijkstra's algorithm.

Hence, if there were no defaults, the second part of query Q4 would have to be written as

```
state("Germany") state("China")
shortest_path[neighbour_of+,
    fun(n: neighbour_of) 1, fun(u: state, v: state) 0]
rewrite[state -> , neighbour_of -> neighbour_of] count
```

The *subgraph* operation restricts the database graph for the following steps of a query. The argument (in square brackets) is a list of restrictions of the form

```
<restriction-list> ::= <restriction> | <restriction>, <restriction-list>
<restriction> ::= <classname> where <condition>
```

In the restriction list one can mention simple classes, link classes, or path classes. The semantics is that for the following steps of the query for each class that is mentioned only the objects qualified by the condition are part of the database graph. If, for example, a node class is restricted, then also only the edges incident with these nodes are present (or rather, visible) within the database; if edges are restricted, also paths going through “invalid” edges disappear. There is an inverse operation called *fullgraph* which restores the complete database graph for further steps of a query. Both *subgraph* and *fullgraph* form separate steps of a query.

We illustrate the use of the *subgraph* operation in connection with a more interesting example of a shortest path search. Consider the following query on the highway network (Section 2.1.2)

Q8. Find a shortest path from exit 16 to exit 252, avoiding a fog area described by a REGIONS value (a collection of polygons) *fog*!

```
subgraph[section where not (section.route intersects fog)];

exit(nr = 16) exit(nr = 252) shortest_path[section+,
     fun(s: section) length(s.route)/s.top_speed,
     fun(v: vertex, target:vertex) dist(v.pos, target.pos)/200]
```

Here we have in the first step restricted to edges free from fog and in the second step computed a shortest path over these edges with respect to traveling time (assuming lengths and distances are stored and computed in kms, and top-speed in kms/hour).

As a further graph operation, let us introduce an operation *circle* which determines a subgraph within some radius from a given start node:

SIMPLEOBJ $\times$ REAL $\rightarrow$ SEQ **circle**

Parameters are similar to those for *shortest_path*:

- a path type describing the edges that may be used
- a function assigning a cost to each kind of edge that may occur in the path.

For example :

Q9. Which exits of the highway network are within less than 50 kms from the junction “Westhofener Kreuz”?

```
junction(name = "Westhofener Kreuz") 50 circle[section+, fun(s: section)
     length(s.route)]
rewrite[exit -> exit, vertex -> , section -> ]
```

Here we have used rewrite to eliminate all *vertex* objects that are not *exits*. Observe that the semantics of rewrite (sequentially checking sequence patterns) is crucial to achieve this effect.

4 Representation and Execution Strategy

The purpose of this last major section is to show a system architecture and give a rough sketch of a planned implementation of the GraphDB data model and query language, in order to lend some credibility to the model of Sections 2 and 3. We do not claim to offer a final system design here – this is beyond the scope of this paper. After describing the overall architecture and environment we shall just cover a few selected aspects of implementation, namely, some candidates for representation structures, implementation of the derive statement, and implementation of graph operations.

4.1 System Architecture

For a GraphDB system as an implementation of the GraphDB data model and query language we assume an extensible system architecture (see Figure 8) similar to the one used in Gral [Gü89] and generalized in the SOS approach [Gü93].

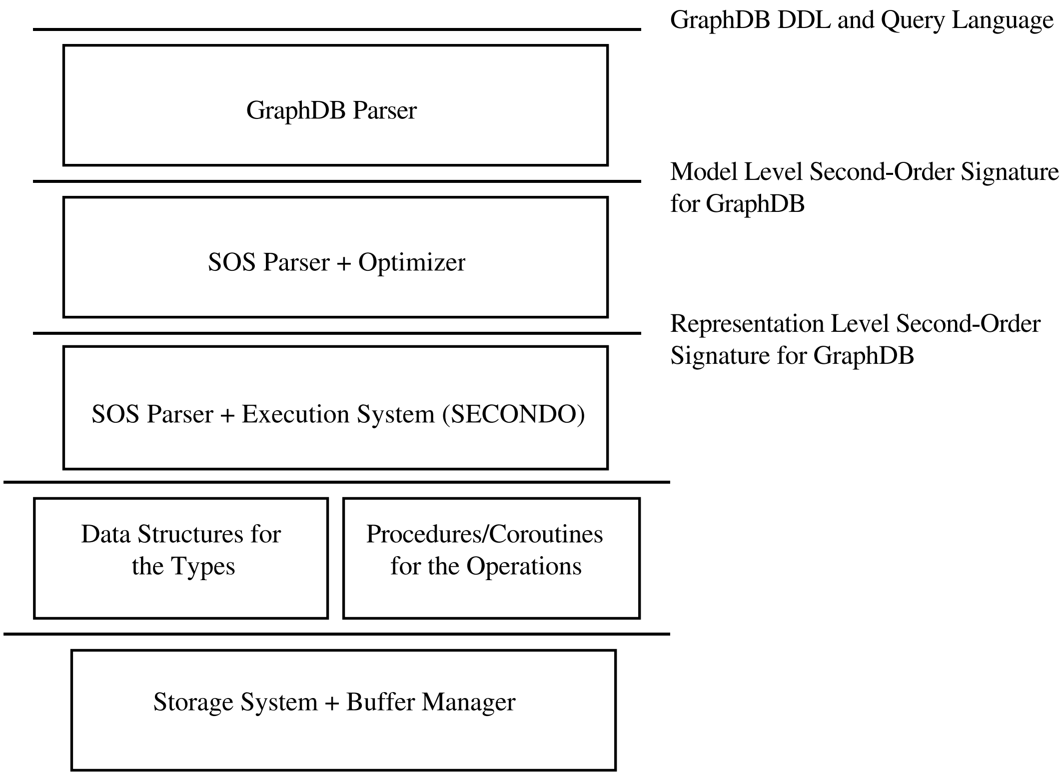

Figure 8

The basic idea of the SOS approach [Gü93] is that there is a single formal model, called *second-order signature*, that allows one to describe either a data model and query language or a representation and query processing system. A second-order signature (SOS) consists of two coupled signatures: the first signature describes a type system and the second one an algebra over the types of the first signature. Hence at the model level the types describe the conceptual structures that the user manipulates and the algebra a query language. At the representation level, the types are data structures available in the system and the operations correspond to algorithms implemented in the system, usually available as procedures. An *SOS optimizer* accepts *programs* of the model level, which consist of data definition, update, and query commands, and translates them into programs of the representation level; there is a simple generic language for programs at both levels. The SOS optimizer performs its task using a set of *optimization rules* that are part of an SOS specification. An *SOS execution system* offers mechanisms to register data structures for types and procedures (or coroutines in case of stream processing) for the operations of a representation level signature; given these, it is able to execute programs of the representation level.

Note that the components of an SOS system are completely independent of the data model and execution system they realize. We are currently implementing an SOS parser and execution system called *SECONDO* (whose parser component is capable to parse programs of the model as well as the representation level). The storage system and buffer manager for SECONDO can be taken from the Gral system. Unfortunately, it will still be a major task to implement an SOS optimizer (somewhat analogous to Gral's rule-based optimizer [BeG92]).

To implement a GraphDB system under the SOS architecture one needs to

- design an SOS signature for the model level which should closely resemble the data model and query language described in this paper,

- implement a GraphDB parser to translate the described data definition and query language into the more formal SOS programs of the model level,
- design an SOS signature for the representation level,
- write optimization rules to translate from the model level to the representation level,
- implement data structures for the types and algorithms for the operations of the representation level.

Our first goal is to design the representation level SOS signature which means to investigate which data structures are useful to represent the graph structures of the data model and which algorithms are needed to implement tools of the query language like derive or rewrite. This is the topic of the remainder of this section. When these data structures and algorithms have been implemented, we will be able to pose the "executable" equivalents of GraphDB queries to the SECONDO/GraphDB system and have them executed.

4.2 Structures

In this section we describe a number of data structures to represent a database as well as temporary structures occurring in query processing. There are permanent structures to represent simple classes, link classes, or path classes separately as well as graph structures combining them. There are also index structures and temporary structures such as streams or adjacency lists in memory.

4.2.1 Representation of a Simple Class

Let us call the following representation *SimpleClass1*. It represents the extension of a class (c, T). Note that this extension may not only contain objects of type c but also objects of other types that are subtypes of c. We assume that a class hierarchy tree can be decomposed into several subtrees and that *SimpleClass1* represents a collection of objects of *one* subtree; the tuple type under which this collection is seen will be that of the root of this subtree.

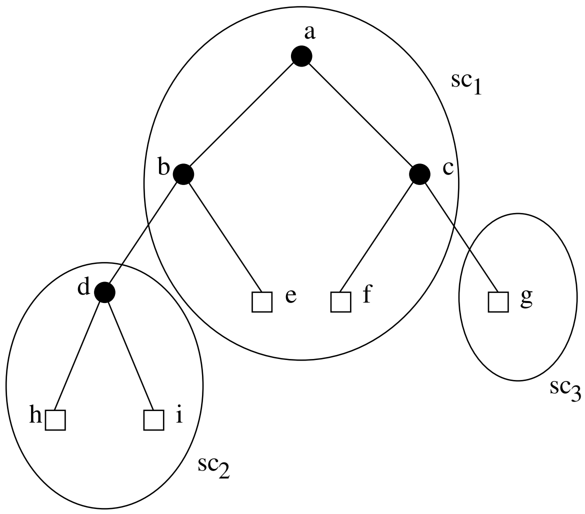

Figure 9

In Figure 9, the class hierarchy with root class a is represented in three *SimpleClass1* representations denoted as sc_1, sc_2, and sc_3, respectively. Here sc_1 represents a part of the extension of class a, sc_2

the complete class *d*, and sc_3 the class *g*. We call the extension of a class *c* without all the objects from its subclasses the *kernel class c*. Hence for example sc_1 represents the kernel classes *a*, *b*, *c*, *e*, and *f*.

We assume that an *object identifier* (e.g. in the set *oids*(*c*)) is a pair (*c*, *index*) where *c* identifies the *object type* of this object (that is, the most specific class it belongs to) and *index* is a positive integer. The numbers *index* are assigned consecutively starting from 1 on object creation.

Conceptually, a *SimpleClass1* representation *sc* consists of

- a *sequence of object representations*, one for each object in the union of the kernel classes represented by *sc*, and
- one *object index* for each kernel class in *sc*.

The object index is simply an array indexed by the *index* component of the object identifiers containing suitable physical addresses (e.g. page number + page index) for the object representations. Each object index is represented in a list of pages (so one can compute from the index the address of the entry on a page); the sequence of object representations is also represented in a list of pages. The order of the object representations can either be arbitrary or be determined by external criteria (such as B-tree clustering over an attribute).

A *SimpleClass1* representation can be built sequentially; whenever an object representation is placed on a page, its address is entered into the appropriate index. For *n* objects to be placed on *p* pages, the time required is (O(*n*), O(*p*))[3], assuming that the object index pages can be kept in a buffer.

4.2.2 Representation of a Link Class

Basically, a link class is represented (as *LinkClass1*) in the same way as a simple class. There are two additional features:

- The edge object representations are stored in sorted order of the third components, that is, the identifiers of the source nodes. Remember that the extension of a link class (*c*, *T*, *d*, *e*) is a set of quadruples subset of $oids(c) \times values(T) \times oids(d) \times oids(e)$, hence the sequence of object representations is ordered by *oids*(*d*)-values.
- There is an additional index called *source index*, which allows to find quickly the edges starting from a given node.

The source index is conceptually an array whose entries are pairs of the form (*oid*, *address*) where *oid* is an element of *oids*(*d*) and *address* the physical address of *the first edge object* in the sequence of object representations which has *oid* as a source node. The pairs are ordered within the array by *oid*-values, hence a binary search is possible. The source index is also represented in a list of pages, for binary search these should be put into consecutive buffer pages.

Building this representation is a bit more expensive, since edge objects need to be sorted (O($n \log n$), O($p \log p$)). After sorting, the source index can be built sequentially in linear time.

Optionally, the representation of a link class can be extended by a *target index*. This is, like the source index, an array with entries of the form (*oid*, *address*). The *oid* is now from *oids*(*e*). Since edge objects are not ordered by target nodes in the sequence, it is in this case necessary to have one

---

[3] In this notation, the first component denotes CPU time, the second the number of page accesses.

entry for each edge in the target index. Of course, the entries are in *oid*-order in the array so that binary search is possible. The array is represented in another list of pages.

Note that the source index and the target index are static structures. This is sufficient when a link class is only built temporarily in query processing, or if the graph to be represented does not change often. For rare insertions of edge objects, the index array can be rewritten. In highly dynamic applications one should use a B-tree index to manage the source (and target) index.

4.2.3 Representation of a Path Class

Basically, a path class is represented (as *PathClass1*) similarly as a simple class. Additionally, it has optionally a *node index* and an *edge index*.

A *node index* has the same structure as a target index of a link class. The entries are pairs (*oid*, *address*) where *oid* is an identifier of an object of a simple class occurring as a node within some path object of the path class, and *address* is the physical address of that path object in the sequence of (path) object representations. Pairs are in *oid*-order within the array.

An *edge index* is the same as a node index except that edges within a path are indexed.

4.2.4 A Clustered Graph Representation

The structures described in the previous sections represent nodes, edges, and paths of a graph separately. This is certainly needed, in particular to support the dynamic creation of a part of the graph (say, edges) in a query. An alternative is to put nodes, edges, and paths into a single structure which allows a better clustering of objects into pages with respect to graph traversal operations.

Such a structure that we call *SimpleGraph1* here has been described in [Gü91]. Essentially, a graph with possibly several node classes, edge classes, and path classes is represented as a *sequence of node blocks* which in turn is stored within a list of pages. A *node block* is a sequence of *graph representation elements (GREs)*. There are five kinds of GREs representing a node object (or simple object), an edge object (link object), a path object, a reference to an edge object, or a reference to a path object. The structure of a node block can then be described as follows:

```
<node-block> ::= <nodeobj> <edgeobj>* <edgeref>* <pathobj>* <pathref>*
```

That means, clustered behind a node object are all the outgoing edge objects from that node. Then the incoming edges are represented by references (the edge objects themselves are obviously stored with their source node). Then all path objects with this node as the start node follow; after them there may be references to other path objects in which this node participates. There are also object indices as in the three structures mentioned above.

The structure does not yet fix the relative order between different node blocks. Hence, there is a degree of freedom that can be used to cluster node blocks, for example, by geometric location (that is, by the values of a spatial attribute of the nodes). For more details see [Gü91]. The structure has already been implemented within a Gral extension.

4.2.5 Index Structures

We assume that the B-tree and the LSD-tree are available. The LSD-tree [HeSW89] is a spatial index structure that can store in particular points and rectangles and supports a number of interesting spatial search operations (e.g. range queries). We assume that each of these structures can be used in two ways:

(a) as a "clustering agent" which can be combined with any of the structures mentioned before. In this case the arrangement of objects into pages (e.g. of the sequence of object representations or the sequence of node blocks) is controlled by the index structure.

(b) as a secondary index. In this case the structure does not influence the placement of objects into pages.

Implementations of both structures are already available within the Gral system.

4.2.6 Temporary Structures

The basic temporary structures needed in query processing are *uniform streams of objects*, *heterogeneous streams of objects*, and *streams of tuples of objects*. Stream processing means that objects are moved in a pipelined fashion between query processing operators, so the complete set of objects of a stream need not be materialized or in memory at any point in time. Stream processing is the basic execution model in SECONDO and is realized by implementing operations as coroutines (it is also already available in a Gral extension). Of course, uniform and heterogeneous streams are meant to represent uniform and heterogeneous sequences of the GraphDB model. Streams of tuples of objects occur in the implementation of the derive statement (see Section 4.3).

Besides streams, object representations and data type values may be present in memory. Additionally, one might consider main memory graph representations (adjacency lists) for intermediate results of a query.

4.3 Implementation of Derive

The derive statement is at the heart of query formulation and the implementation of derive will be a centerpiece of query processing. We have seen that derive comprises the functionality of the *select … from… where* statement and even does more than that. Hence it is obvious that there is no single operator in query processing to implement derive. Rather, all the usual techniques like index access, join methods, filtering by conditions, projection, etc. will need to be used. So, in general, the implementation of derive will be a whole operator tree.

On the other hand, the special capability of derive is to follow connections in a graph. This capability will be represented in query processing by a special operator called *simple_derive*. *Simple_derive* takes as an operand a uniform stream of objects and as a further argument a *subgraph specification* in the same form as in the derive statement. However, for *simple_derive* this subgraph must be connected. The operator then processes each object in its input stream and enumerates for this object all combinations of objects connected with it according to the given subgraph specification. It does that by following pointers and using indices available in the representations of classes as described above (Sections 4.2.1 - 4.2.4). As a result, the *simple_derive* operator produces a *stream of tuples of* *objects* (*to-stream*). Such a to-stream can then be further fed into other operators such as filtering, cartesian product, join methods, and production of new objects to realize a complete derive statement.

In Section 3.3.1 we have seen that the on-clause of a derive statement specifies in general several connected graphs. One basic strategy for translating a derive statement into a query plan is the following (this needs to be done by the optimizer):

- For each connected graph in the on-clause introduce one instance of the *simple_derive* operator with this graph as an argument.
- For each *simple_derive*, select one object class in the parameter graph for its input stream. This decision must be made on the basis of which conditions are present in the where-clause of the derive statement and also, which index structures are present to support the evaluation of such conditions. If there are no conditions, select some object class and feed all its objects into the *simple_derive* (presumably the smallest class).
- Filter the output to-stream of each *simple_derive* by remaining conditions on object classes in its parameter graph.
- If there are join conditions connecting separate graphs of the on-clause, then use a join method to combine the output streams of the respective *simple_derive* occurrences. Otherwise, combine the to-streams by cartesian product operators.
- If necessary, filter by further join conditions that have not been used in the previous step.
- Finally, let an operator produce new objects as specified in the derive-clause.

Let us illustrate (part of) this process by considering possible translations for query Q1:

Q1. List the titles of all books written (coauthored) by Hopcroft in 1983!

```
on person wrote book
where person.name = "Hopcroft" and book.year = 1983
derive book.title
```

which refers to the database

```
class book = title: STRING, publisher: STRING, year: INTEGER;
class person = name: STRING, address: STRING;
link class wrote from person to book;
```

Here the on-clause contains only a single connected graph, hence there will be only one *simple_derive* operator. The graph can be represented as shown in Figure 10.

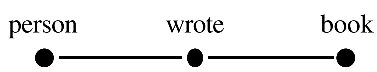

Figure 10

One obtains various possibilities to feed the *simple_derive* operator by picking one of the nodes of this graph and making it the root of a tree, the root will be the input stream. The operator will then for each object in the input stream instantiate the root variable and then descend to instantiate the subtrees.

In the example, the following translations might be possible:

```
person feed filter[name = "Hopcroft"]
sderive[person wrote book]
filter[book.year = 1983] make_object[unnamed = book.title]
```

Here *feed* is an operator enumerating all objects of a given class from some representation (e.g. SimpleClass1) into a stream. The *filter* operator filters objects from a stream (of objects or tuples of objects) by some condition. *Make_object* constructs for each tuple of objects that it receives a new object and sends it into its output stream.

Another alternative is to make *book* the root of the tree. Let us assume that books are clustered by year through a B-tree so that index access is possible. Then the following translation can be used:

```
book 1983 1983 range
sderive[person wrote book]
filter[person.name = "Hopcroft"] make_object[unnamed = book.title]
```

Here the *range* operator denotes a range query on a B-tree; the range is given by two constants. All objects with value between these two constants are fed into a stream. For a more complete example of a stream-based query processing algebra see [Gü93].

The decision between the possibilities has to be taken by the optimizer, as usual, probably on the basis of cost estimates.

Clearly, this is not the whole story of the implementation of the derive statement. For example, to traverse links given by object-valued attributes one needs additional index structures. Also, another valuable strategy for evaluation in the case of several connected graphs in the on-clause may be to first evaluate a join condition to generate an input stream for one of the *simple_derive* occurrences. But the strategy described above is certainly of major importance.

4.4 Implementation of Graph Operations

The general strategy is to implement graph operations by efficient graph algorithms. The two examples *shortest_path* and *circle* from Section 3.3.4 will both be implemented by the A* algorithm (which contains Dijkstra's algorithm as a special case). *Subgraph* has no corresponding algorithm but is implemented by moving the conditions into subsequent graph operations or other database accesses.

The *shortest_path* operator is implemented by an executable operator *sp_astar* which takes the following arguments:

- source and target objects, *source* and *target*,
- a path type,
- for each class of nodes that may lie on the path according to the path type, a function to estimate the distance from such a node to the target of the search,
- for each class of edges that may lie on the path according to the path type, a function to assign a cost to this edge,
- for each class of nodes and edges that the path may traverse according to the path type, possibly a *filter expression* that allows to exclude nodes or edges from consideration.

Most of these arguments have already been explained for the *shortest_path* operator in Section 3.3.4. The additional filter expressions can be used to evaluate during the search conditions given separately in the query (in particular, by a *subgraph* operation).

For the example query Q4, *sp_astar* would work as follows: It would start from the representation of the *source* node ("Germany") in class *state*. It would then look at its path schema to discover that the next edge to be considered must lie in link class *neighbour_of*. Using the source-index in the representation of class *neighbour_of* it would locate the group of edges starting from node "Germany". For each edge it would look at the target node in class *state* and compute the distance to the node *target* using the function supplied as a parameter. It would then select the node with minimal estimated total distance (in fact, in this case minimal distance from the source node) for expansion, expand it by going into the link class through the source index again, and so forth. The interesting thing to observe is how *sp_astar* combines old and new parts of the database graph, and also works relatively efficiently.

As an example for the treatment of the *subgraph* operation, consider the following query plan for query Q8:

```
exit feed filter[nr = 16] single_object
exit feed filter[nr = 252] single_object
sp_astar[section+,
     fun(s: section) length(s.route)/s.top_speed,
     fun(v: vertex, target:vertex) dist(v.pos, target.pos)/200,
     fun(s: section) not (s.route intersects fog)]
```

Here it is assumed that no index over exits is present. The *single_object* operation just checks that its input stream of objects contains only a single object (otherwise an error is reported) and returns this object.

5 Conclusions

We have presented a data model that integrates an explicit modeling of graph structures smoothly into a "standard" object-oriented modeling and querying environment. In particular, explicit path objects are offered, and graph operations can be defined whose argument graphs (subgraphs of the database graph) can be specified by regular expressions over link class names. The derive statement extends the familiar *select … from… where* to a convenient querying of relationships (link classes, edges of the graph). The rewrite operation is a powerful tool for the manipulation of sequences, especially paths, in queries. The model is coupled to an implementation concept which offers special data structures for the representation of graphs and efficient graph algorithms for the graph operations. System architecture and implementation strategy have been outlined in the paper. Besides being attractive for standard applications, the model is particularly suitable for a sophisticated modeling and manipulation of spatially embedded networks, as has been demonstrated by the public transport example.

We are currently developing a first partial prototype for GraphDB following the system architecture and implementation plan described in Section 4.1. The general extensible query processing environment will be offered by the SECONDO which is just about to be finished. To reduce the implementation effort (that is, to make the task manageable at all) we are trying to use as much as possible modules from the Gral system, for example, storage and buffer management, index structures, data types and administration of data types, implementations of query processing operations (e.g. join algorithms). In a first phase, we would like to arrive at a prototype version that demonstrates some interesting part of the query processing capabilities needed for GraphDB. To realize the GraphDB query language as such, it is first necessary to implement a SECONDO optimizer (perhaps along the lines of [BeG92]) which is still a major open task. The development of the GraphDB model and prototype is part of the ESPRIT project AMUSING.

Other future work includes a more complete design of the query language – in this paper we have only introduced some key elements –, the design of a corresponding SOS model level signature, and a formal definition of the semantics of query language operations. Note that the extensible system architecture makes it possible to postpone a complete query language design even until the system is running; missing operations can always be added later.

Acknowledgments

Helpful discussions with Hieu-Thien Pham and Frank Schoppmeier and their work on implementing the GraphDB prototype are gratefully acknowledged. Thanks also to Martin Erwig, Hieu-Thien Pham and Markus Schneider for reading drafts of this paper.

References

[Ag87] Agrawal, R., ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries. Proc. IEEE Data Engineering Conf. 1987, 580-590.

[AlGO91] Albano, A., G. Ghelli, and R. Orsini, A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. Proc. of the 17th Intl. Conf. on Very Large Data Bases, Barcelona, 1991, 565-575.

[AmS92] Amann, B., and M. Scholl, Gram: A Graph Data Model and Query Language. Proc. ECHT'92, Milano, December 1992.

[Andr92] Andries, M., M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche, Concepts for Graph-Oriented Object Manipulation. Proc. 3rd EDBT 1992 (LNCS 580), 21-38.

[BeG92] Becker, L., and R.H. Güting, Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. *ACM Transactions on Database Systems 17 (1992)*, 247-303.

[BiRS90] Biskup, J., U. Räsch, and H. Stiefeling, An Extension of SQL for Querying Graph Relations. *Computer Languages 15 (1990)*, 65-82.

[CoM90] Consens, M., and A. Mendelzon, GraphLog: A Visual Formalism for Real Life Recursion. Proc. ACM Conf. on Principles of Database Systems 1990, 404-416.

[CoM93] Consens, M., and A. Mendelzon, Hy^+: A Hygraph-based Query and Visualization System (Video Demonstration). Proc. ACM SIGMOD 93, 511-516.

[CrMW87a] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, A Graphical Query Language Supporting Recursion. Proc. SIGMOD Conf. 1987, 323-330.

[CrMW87b] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, G^+: Recursive Queries Without Recursion. Proc. 2nd Intl. Conf. on Expert Database Systems, 1989, 355-368.

[CrN89] Cruz, I.F., and T.S. Norvell, Aggregative Closure: An Extension of Transitive Closure. Proc. of the 5th Intl. Conf. on Data Engineering, 1989, 384-391.

[Di90] Dittrich, K., Object-Oriented Database Systems: The Next Miles of the Marathon. *Information Systems 15 (1990)*, 161-168.

[ErG91] Erwig, M., and R.H. Güting, Explicit Graphs in a Functional Model for Spatial Databases. FernUniversität Hagen, Informatik-Report 110, 1991, to appear in *IEEE Transactions on Knowledge and Data Engineering*.

[GePTV93] Gemis, M., J. Paredaens, I. Thyssens, and J. van den Bussche, GOOD: A Graph-Oriented Object Database System (Video Demonstration). Proc. ACM SIGMOD 93, 505-510.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Data Bases, 1989, 33-44.

[Gü91] Güting, R.H., Extending a Spatial Database System by Graphs and Object Class Hierarchies. In: G. Gambosi, H. Six, and M. Scholl (eds.) Proc. Int. Workshop on Database Management Systems for Geographical Applications (Capri, May 1991), Springer, 1992.

[Gü93] Güting, R.H., Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. Proc. ACM SIGMOD Conf. (Washington, 1993), 277-286.

[GüS93] Güting, R.H., and M. Schneider, Realm-Based Spatial Data Types: The ROSE Algebra. Fernuniversität Hagen, Report 141, 1993.

[GüZC89] Güting, R.H., R. Zicari, and D.M. Choy, An Algebra for Structured Office Documents. *ACM Transactions on Information Systems 7 (1989),* 123-157.

[GyPV90a] Gyssens, M., J. Paredaens, and D. van Gucht, A Graph-Oriented Object Database Model. Proc. ACM Conf. on Principles of Database Systems 1990, 417-424.

[GyPV90b] Gyssens, M., J. Paredaens, and D. van Gucht, A Graph-Oriented Object Model for Database End-User Interfaces. Proc. ACM SIGMOD Conf. 1990, 24-33.

[HeSW89] Henrich, A., H.-W. Six, and P. Widmayer, The LSD Tree: Spatial Access to Multidimensional Point- and Non-Point-Objects. Proc. of the 15th Intl. Conf. on Very Large Data Bases (Amsterdam, 1989), 45-53.

[Kung86] Kung, R., *et al.*, Heuristic Search in Database Systems. In: L. Kerschberg (ed.), Expert Database Systems. Benjamin Cummings, 1986.

[MaS90] Mannino, M., and L. Shapiro, Extensions to Query Languages for Graph Traversal Problems. *IEEE Trans. on Knowledge and Data Engineering 2 (1990)*, 353-363.

[MaV93] Maier, D., and B. Vance, A Call to Order. Proc. ACM Symposium on Principles of Database Systems (Washington, 1993), 1-16.

[Ni80] Nilsson, N.J., Principles of Artificial Intelligence. Tioga Publ. Company, Palo Alto, CA, 1980.

[OrM88] Orenstein, J., and F. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Trans. on Software Engineering 14 (1988)*, 611-629.

[RoFS88] Rossopoulos, N., C. Faloutsos, and T. Sellis, An Efficient Pictorial Database System for PSQL. *IEEE Trans. on Software Engineering 14 (1988)*, 639-650.

[Rose86] Rosenthal, A., S. Heiler, U. Dayal, and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications. Proc. SIGMOD Conf. 1986, 166-176.

[StR86] Stonebraker, M., and L.A. Rowe, The Design of POSTGRES. Proc. of the 1986 SIGMOD Conf. (Washington, DC, May 1986), 340-355.

[SvH91] Svensson, P., and Z. Huang, Geo-SAL: A Query Language for Spatial Data Analysis. Proc. SSD 91 (Zurich, Switzerland), 1991, 119-140.