

# Realm-Based Spatial Data Types: The ROSE Algebra

Ralf Hartmut Güting and Markus Schneider

*Received April 15, 1993; revised version received, February 14, 1994; accepted March 18, 1994.*

**Abstract.** Spatial data types or algebras for database systems should (1) be fully general, that is, closed under set operations, (2) have formally defined semantics, (3) be defined in terms of finite representations available in computers, (4) offer facilities to enforce geometric consistency of related spatial objects, and (5) be independent of a particular DBMS data model, but cooperate with any. We present an algebra that uses *realms* as geometric domains underlying spatial data types. A realm, as a general database concept, is a finite, dynamic, user-defined structure underlying one or more system data types. Problems of numerical robustness and topological correctness are solved within and below the realm layer so that spatial algebras defined above a realm have very nice algebraic properties. Realms also interact with a DMBS to enforce geometric consistency on object creation or update. The ROSE algebra is defined on top of realms and offers general types to represent point, line, and region features, together with a comprehensive set of operations. It is described within a polymorphic type system and interacts with a DMBS data model and query language through an abstract *object model interface*. An example integration of ROSE into the object-oriented data model  $O_2$  and its query language is presented.

**Key Words.** Realm, finite resolution, numerical robustness, topological correctness, geometric consistency, object model interface.

## 1. Introduction

We consider a *spatial database system* to be a full-fledged DBMS with additional capabilities for the representation and manipulation of geometric data. As such, it provides the database technology to support applications such as *geographic information systems*. The standard DBMS view for the organization of spatial information is the following: A database consists of several classes of objects. A

---

Ralf Hartmut Güting, Prof. Dr. rer. nat., is Professor, and Markus Schneider, Dipl.-Inform., is Research Assistant, Praktische Informatik IV, FernUniversität Hagen, D-58084 Hagen, Germany, [gueting@fernuni-hagen.de](mailto:gueting@fernuni-hagen.de), [schneide@fernuni-hagen.de](mailto:schneide@fernuni-hagen.de).

*spatial object* is just an object with an associated value (“attribute”) of a *spatial data type* such as *point*, *line*, or *region*. This is true regardless of whether the DBMS uses a relational, complex object, object-oriented, or other data model. Hence, the definition and implementation of spatial data types is probably the most fundamental issue in the development of spatial database systems.

Although spatial data types (SDTs) are used routinely in the description of spatial query languages (e.g., Lipeck and Neumann, 1986; Joseph and Cardenas, 1988; Svensson and Huang, 1991, Tomlin, 1990), have been implemented in some prototype systems (e.g., Rossopoulos et al., 1988; Orenstein and Manola, 1988; Güting, 1989), and some formal definitions have been given (Güting, 1988a, Scholl and Voisard, 1989; Gargano et al., 1991), there is still no completely satisfactory solution available according to the following criteria:

- *Generality.* The geometric objects used as SDT values should be as general as possible. For example, a region value should be able to represent a collection of disjoint areas, each of which may have holes. More precisely, the domains of data types *point*, *line*, and *region* must be closed under union, intersection, and difference of their underlying point sets. This allows for the definition of powerful data type operations with nice closure properties.
- *Rigorous definition.* The semantics of SDTs, that is, the possible values for the types and the functions associated with the operations, must be defined formally to avoid ambiguities for the user and the implementor.
- *Finite resolution.* The formal definitions must take into account the finite representations available in computers. This has so far been neglected in definitions of SDTs. It is left to the programmer to close this gap between theory and practice, which predictably leads to not only numerical but also topological errors.
- *Treatment of geometric consistency.* Distinct spatial objects may be related through geometric consistency constraints (e.g., adjacent regions have a common boundary). The definition of SDTs must offer facilities to enforce such consistency.
- *General object model interface.* Spatial data types as such are of little use unless they are integrated into a DBMS data model and query language. However, a definition of SDTs should be valid regardless of a particular DBMS data model and, therefore, not dependent on it.<sup>1</sup> Instead, the SDT definition should be based on an abstract interface to the DBMS data model, which we call the *object model interface*.

The purpose of this article (together with a companion work, Güting and Schneider, 1993) is to develop a formal definition of spatial data types fulfilling

---

1. This also holds for the implementation level: A spatial type extension package (STEP) should be able to cooperate with any extensible DBMS that offers a suitable interface regardless of its data model.

these criteria. A central idea is to introduce into the DBMS the concept of a *realm*, a finite, user-defined structure that is used as a basis for one or more system data types. Realms are somewhat similar to enumeration types in programming languages. A realm used as a basis for spatial data types is essentially a finite set of points and *non-intersecting* line segments. Intuitively, it describes the complete underlying geometry of an application. All points, lines, and regions associated with objects (*spatial attribute values*) can be defined in terms of points and line segments present in the realm. In fact, spatial attribute values are created only by selecting some realm objects. They are never created or updated directly. Instead, updates are performed on the realm and from there propagated to the dependent attribute values.

Hence, all attribute values occurring in a database are *realm-based*. Furthermore, the algebraic operations for the spatial data types are defined to construct only geometric objects that are also realm-based. So the spatial algebra is closed with respect to a given realm. This means that no two values of spatial data types occurring in geometric computation have “proper” intersections of line segments. Instead, two initially intersecting segments already have been split at the intersection point when they were entered into the realm. One could say that any two intersecting SDT values (say, lines or regions) “have become acquainted” already when they were entered into the realm. This is a crucial property for the correct and efficient implementation of geometric operations.

Realm objects (points and segments) are defined not in abstract Euclidean space but in terms of finite representations. All geometric primitives and realm operations (e.g., updates) are defined in error-free integer arithmetic. To map an application’s set of intersecting line segments into a realm’s set of non-intersecting segments, redrawing and finite resolution geometry is used (Greene and Yao, 1986). Although intersection points computed with finite resolution in general move away from their exact Euclidean position, this concept ensures that the unavoidable distortion of geometry (i.e., the numerical error) remains bounded and very small, and that essentially no topological errors occur (see Section 2; Güting and Schneider, 1993). This means that a programmer has a precise specification that directly lends itself to a correct implementation. It also means that the spatial algebra obeys algebraic laws precisely in theory as well as in practice. Furthermore, realms also solve the geometric consistency problem.

Most closely related to this work are the formal definitions of spatial data types (or algebras) (Güting, 1988a, 1988b; Scholl and Voisard, 1989; Gargano et al., 1991; Voisard, 1992). None of these proposals fulfills all of the criteria given above. Güting (1988a, 1988b) provided data types for points, lines, and regions, but they are too restricted (e.g., a region is a single simple polygon without holes). Scholl and Voisard (1989) defined general regions, and Voisard (1992) extended this to general types for points and lines. However, the definitions are unnecessarily complex. Gargano et al. (1991) gave only a single type for all kinds of geometric objects; a value is essentially a set of sets of pixels. We feel this is not sufficient,

since many interesting spatial operations cannot be expressed. Although all of these proposals give formal definitions, those of Güting and of Scholl and Voisard were not based on finite resolution; hence the numeric correctness problems are not addressed. Gargano et al. based their definitions on a finite underlying set of pixels, but this is not practical since these finite representations are far too large to be efficiently manageable. The geometric consistency problem is not solved in any of these proposals; there is some weak support in Güting (1988a) through an *area* data type, but it is not sufficient. Finally, all three proposals connected their spatial types to a fixed data model: Güting and Gargano et al. to the relational model, and Scholl and Voisard to a complex object algebra. Only Scholl and Voisard emphasize a clean interface between the spatial algebra and the general object model. We extend their work by offering an abstract interface not dependent on any particular data model.

Separating geometric primitives from the remainder of geometric modeling already was proposed by Frank and Kuhn (1986). Because of the conflict between the infinite precision real numbers of Euclidean geometry and the finite precision number systems of computers, they suggest abandoning coordinate-based geometry to consider only the topological structures of point sets underlying spatial values. Their topological data model (continued by Egenhofer et al., 1989) is based on simplicial complexes and has a similar purpose as our concept of realms. Essentially they offer an irregular triangular network partition of the plane as a geometric domain over which spatial objects could be defined. However, the connections are missing to the underlying finite arithmetic, as well as to spatial data types based on this model. Also, in our view, a triangular partition contains too much information; it is sufficient to keep those points and segments needed for spatial attribute values in a geometric domain. Finally, their model is an abstract one, whereas we show realms within a database context.

Our description and formal development of realm-based spatial data types was begun in a previous article (Güting and Schneider, 1993), in which the lower layers were defined, namely numerically robust geometric primitives, realms and their update operations, and a number of realm-based structures (e.g., cycles and faces), and primitives. In the present article, the spatial data types *points*, *lines*, and *regions* and their operations, that is, the ROBust Spatial Extension (ROSE) algebra, are described and defined formally. Related issues, such as modeling partitions of the plane within the type system and an abstract object model interface, are addressed. We also show how the ROSE algebra can be integrated with a DBMS data model and query language, using  $O_2$  as an example. In the following section, we provide an informal overview of the complete concept.

## 2. Overview: Realm-Based Spatial Data Types

A realm is a set of points and non-intersecting line segments over a discrete domain, that is, a grid, as shown in Figure 1. Values of spatial data types can be composed

Figure 1. Example of a realm

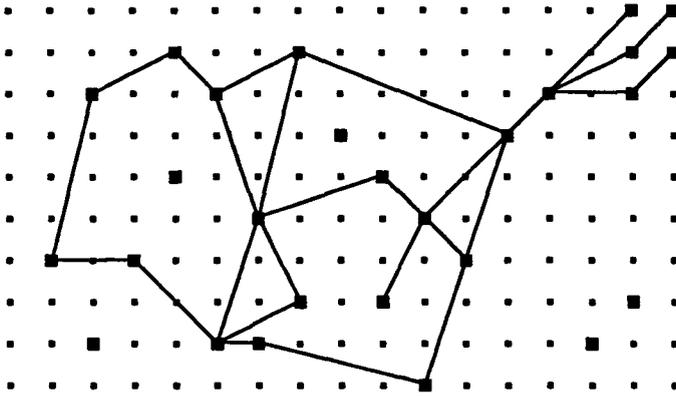
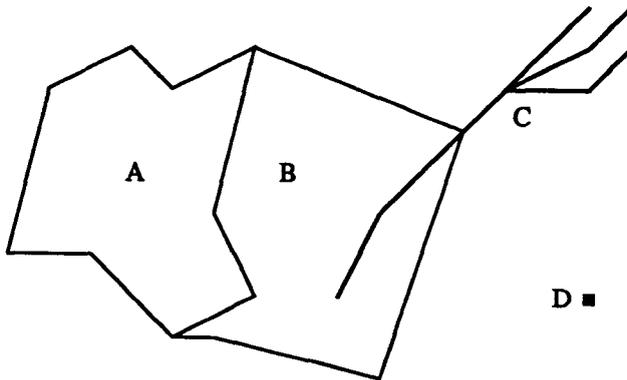


Figure 2. Realm objects defined over the realm of Figure 1



from the objects present in a realm. Figure 2 shows some values definable over the realm of Figure 1. Our realm-based spatial data types are called points, lines, and regions, hence *A* and *B* represent region values, *C* is a lines value, and *D* a points value. The precise structure of these values is not yet relevant here. One can imagine that *A* and *B* belong to two adjacent countries, *C* represents a river, and *D* a city.

The underlying grid of a realm arises simply from the fact that numbers have a finite representation in computer memory. In practice, these representations will be of fixed length and correspond to INTEGER or REAL data types available in programming languages (or to special, higher precision implementations of number systems). Of course, the resolution is much finer than can be shown in Figure 1.

The concept of a realm as a basis of spatial data types serves the following purposes:

- It enforces *geometric consistency* of related spatial objects. For example, the common part of the borders of countries  $A$  and  $B$  is exactly the same for both objects.
- It guarantees nice *closure properties* for the computation with spatial data types above the realm. For example, the intersection of region  $B$  with line  $C$  (the part of river  $C$  lying within country  $B$ ) is also a realm-based lines value.
- It shields geometric computation in query processing from numeric correctness and robustness problems. This is because such problems arise essentially from the computation of intersection points of line segments which normally do not lie on the grid. With realm-based SDTs, there are *never any new intersection points computed in query processing*. Instead, the numeric problems are treated *below* the realm level, namely, whenever updates are made to a realm.
- Additionally, a data structure representing a realm can be used as an index to the database. Our implementation concept assumes that each point and segment in a realm has an associated list of logical pointers to the spatial attribute values defined over it in the database.

Let us now focus on the treatment of numerical correctness problems below and within the realm level. This is necessary because geometric data coming from the application are not intersection-free, as required for a realm. Application data can at the lowest level of abstraction be viewed as a set of points and *intersecting* line segments. These need to be transformed into a realm. As mentioned before, the fundamental problem is that intersection points usually do not lie on the grid.

In Figure 3, the intersection point  $D'$  of line segments  $A$  and  $B$  will be moved to the closest grid point  $D$ . This leads to the following topological errors: (1) A test whether  $D$  lies on  $A$  or  $B$  fails. (2) A test whether  $D$  lies properly within some area defined below  $A$  and  $B$  will incorrectly yield true. (3) If there is another segment  $C$  between the true intersection point and  $D$ ,  $D$  will be reported to lie on the wrong side of  $C$ . The basic idea to avoid these errors, is to slightly change segments  $A$  and  $B$  by transforming them into chains of segments going through  $D$ , as shown in Figure 4. However, this allows a segment to drift (through a series of intersections) by an arbitrary distance from its original position. For example, a further intersection of  $A$  with some segment  $C$  (Figure 5) is resolved as shown in Figure 6, where intersection point  $E$  is already a considerable distance from the true intersection point of  $A$  and  $C$ . Note in particular that, in Figure 6, segment  $A$  has been moved to the other side of a grid point (indicated by the arrow) which may later be reported to lie on the wrong side of  $A$ .

A refined solution was proposed by Greene and Yao (1986). They defined for a segment  $s$  an *envelope*  $E(s)$  as the collection of grid points that are immediately above, below, or on  $s$ . An intersection of  $s$  with some other segment may lead

Figure 3.

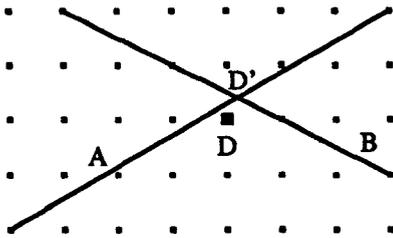


Figure 4.

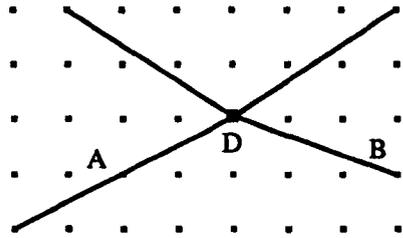


Figure 5.

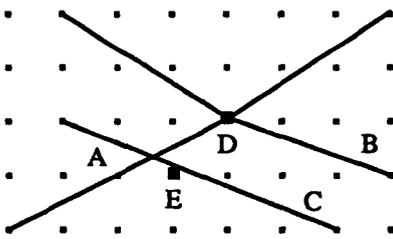
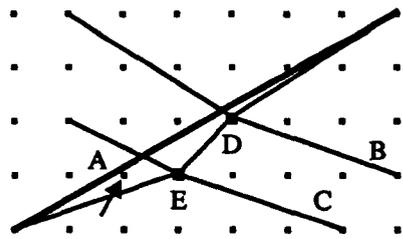


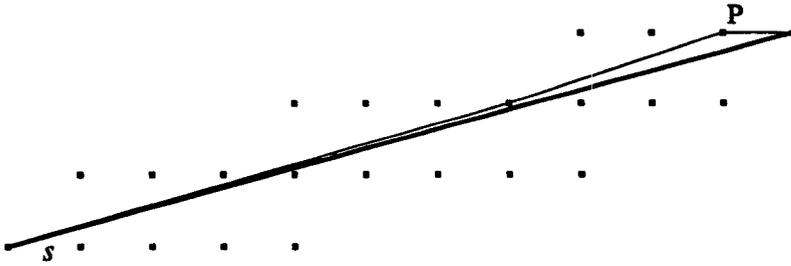
Figure 6.



to a requirement that  $s$  should pass through some point  $P$  on its envelope (the grid point closest to the true intersection point). This requirement is then fulfilled by redrawing  $s$  by some polygonal line *within the envelope* rather than by simply connecting  $P$  with the start and end points of  $s$ . Figure 7 shows a segment  $s$  (drawn fat) together with the grid points of its envelope. Slightly above  $s$ , a redrawing of  $s$  through  $P$  is shown.

Intuitively, the process of redrawing can be understood as follows: Think of segment  $s$  as a rubber band and the points of the envelope as nails on a board. Now grip  $s$  at the true intersection point and pull it around  $P$ . The resulting polygonal path is the redrawing. The number of segments of this path is, in the worst case, logarithmic in the size of the grid, but it seems that in most cases only very few segments are created. This approach guarantees that the polygonal line describing a segment always remains within the envelope of the original segment. We adopt the technique for realms. It then means that, by redrawing, a segment never can drift to the other side of a realm point. However, it still might happen that, after

**Figure 7. Redrawing of segment  $s$  through envelope point  $p$**



a redrawing, a realm point is found to lie on a segment on which it did not lie originally.

The formal definition of realm-based SDTs is organized as a series of layers. Each layer defines its own structures and primitives, using the notions of the layers below. These layers are described bottom-up in Güting and Schneider (1993) and in the rest of this article. Let us briefly provide an overview of this development.

The lowest layer introduces *robust geometric primitives*. It defines a discrete space  $N \times N$  where  $N = \{0, \dots, n - 1\}$  is a subset of the natural numbers. The objects in this space are points and line segments with coordinates in  $N$ , called *N-points* and *N-segments*. A number of operations (predicates) such as whether an *N-point* lies *on* an *N-segment* or whether two *N-segments* *intersect*, and which *N-point* is the result of intersecting two *N-segments*, are defined. The crucial point is that these definitions are given in terms of error-free integer arithmetic; hence they are directly implementable.

Next, geometric *realms* are defined as described above; elements are called *R-points* and *R-segments*. Basic operations on realms are insertion and deletion of *N-points* and *N-segments*, which may trigger the redrawing of segments as described above. Realms offer an interface to cooperate with a database system. For example, the operation of inserting an *N-segment* returns a modified realm, a redrawing of the inserted segment, and a set of redrawings of segments in the database that need to be modified together with logical pointers to database representations of these segments.

The second layer defines certain structures present in a realm that serve as a basis for the definition of SDTs. A realm can be viewed as a planar graph; an *R-cycle* is a cycle of this graph. An *R-face* is an *R-cycle* possibly enclosing some other disjoint *R-cycles* corresponding to a region with holes. An *R-unit* is a minimal *R-face*. These three notions support the definition of a regions data type. An *R-block* is a connected component of the realm graph; it supports the definition of a *lines* data type. For all of these structures, there are also predicates defined to describe their possible relationships.

This completes the scope of Güting and Schneider (1993). The remainder of this article is organized as follows: The definitions of the first two layers are reviewed in Section 3. Section 4 introduces the third layer, *spatial data types* points, lines, and regions, and defines the structure of corresponding values. The following two sections prepare the definition of the fourth and final layer. In Section 5, a flexible type system is introduced that allows one to precisely describe polymorphic operations. In Section 6, the *object model interface (OMI)* is defined. As a top layer, the ROSE algebra is described in Section 7. In Section 8, we show how the ROSE algebra can be integrated with a given DBMS data model and query language, choosing  $O_2$  as an example. Section 9 concludes the work and offers suggestions for future research.

### 3. Robust Geometric Primitives, Realms, Realm-Based Structures

In this section, we review the concepts and formal definitions from Güting and Schneider (1993) as a basis for defining the ROSE algebra. We have already mentioned that there are several layers of definitions, each of which introduces its own structures and operations and uses the notions of the layers below. To distinguish operations of the various layers we use the following typographical convention:

- Layer 1–robust geometric primitives: underscore (e.g., intersect)
- Layer 2–realms, realm-based primitives: underscore italic (e.g., *area-disjoint*)
- Layer 3–spatial algebra primitives: bold italic (e.g., ***area-disjoint***)
- Layer 4–ROSE operations: bold (e.g., **inside**)

A summary of the various layers with their objects and operations is given in the Appendix.

#### 3.1 Robust Geometric Primitives

The lowest layer introduces a finite discrete space  $N \times N$  with  $N = \{0, \dots, n-1\} \subseteq \mathbf{N}$ , points and line segments over this space, and some simple predicates and operations on them. All definitions are based on error-free integer arithmetic, which enables direct and robust implementation. An *N-point* is a pair  $(x, y) \in N \times N$ . An *N-segment* is a pair of distinct *N-points*  $(p, q)$ ; the segments  $(p, q)$  and  $(q, p)$  are defined to be equal.  $P_N$  denotes the set of all *N-points*, and  $S_N$  the set of all *N-segments*. Formal definitions of *robust geometric primitives* defined on *N-points* and *N-segments* were given by Güting and Schneider (1993). We explain the primitives informally here: Two *N-segments* meet if they have exactly one end point in common. They overlap if they are collinear and share a (partial) *N-segment*. If they have exactly one common point but do not meet, they intersect. They are disjoint if they are neither equal nor meet nor overlap nor intersect. The on primitive tests whether an *N-point* lies on an *N-segment*; the in primitive does nearly the same, but the *N-point* must not coincide with one of the end points of the *N-segment*. The intersection

primitive calculates the intersection point of two  $N$ -segments and rounds it to the nearest  $N$ -point.

### 3.2 Realms

Realms serve as a basis for SDTs and essentially represent a finite, user-defined set of points and *non-intersecting* line segments over a discrete domain. Given  $N$ , a realm over  $N$  ( $N$ -realm for short) is a set  $R = P \cup S$  such that

- (1)  $P \subseteq P_N, S \subseteq S_N$
- (2)  $\forall s \in S : s = (p, q) \Rightarrow p \in P \wedge q \in P$
- (3)  $\forall p \in P \forall s \in S : \neg (p \text{ in } s)$
- (4)  $\forall s, t \in S, s \neq t : \neg (s \text{ and } t \text{ intersect}) \wedge \neg (s \text{ and } t \text{ overlap})$

The elements of  $P$  and  $S$  are called  $R$ -points and  $R$ -segments. There is an obvious interpretation of a realm as a spatially embedded planar graph with set of nodes  $P$  and set of edges  $S$ .

### 3.3 Realm-Based Structures and Primitives

This layer defines structures and relationships between these structures that can be discovered within a realm, and that are useful for the definition of SDTs. A realm can be viewed as a planar graph; informally, an  $R$ -cycle is a cycle of this graph. An  $R$ -face is an  $R$ -cycle that possibly encloses some other disjoint  $R$ -cycles that correspond to a region with holes. An  $R$ -unit is a minimal  $R$ -face. These three notions support the definition of a *regions* data type. An  $R$ -block is a connected component of the realm graph; it supports the definition of a *lines* data type. For all of these *realm-based structures*, predicates (primitives) are defined to describe their possible relationships. We now review the most important formal definitions.

An  $R$ -cycle  $c$  is a cycle in the graph interpretation of a realm, defined by a set of  $R$ -segments  $S(c) = \{s_0, \dots, s_{m-1}\}$ , such that

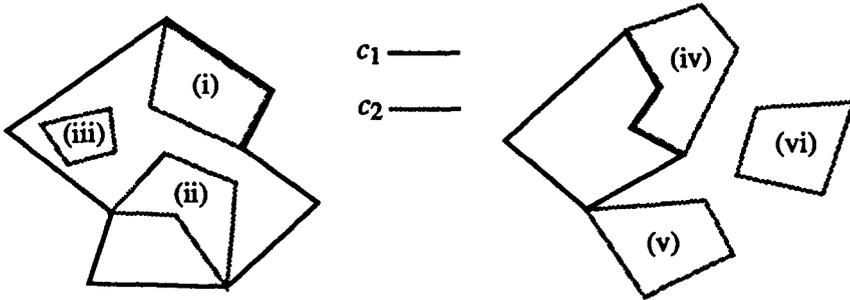
- (1)  $\forall i \in \{0, \dots, m-1\} : s_i \text{ meets } s_{(i+1) \bmod m}$
- (2) No more than two segments from  $S(c)$  meet in any point  $p$ .

Obviously the following relationships may exist between an  $N$ -point  $p$  and an  $R$ -cycle  $c$ :

- (1)  $p \text{ on } c : \Leftrightarrow \exists s \in S(c) : p \text{ on } s$

For  $p = (x, y)$  let  $s_p = ((x, y), (x, n-1))$  (i.e., a vertical segment extending from  $p$  upwards to the edge of the grid). Let  $S_r(c)$  be the set of segments in  $S(c)$  whose right end point, but not the left one, is on  $s_p$  (the left end point is the smaller one of the two end points in the  $(x, y)$ -lexicographical order). Let  $S_i(c)$  be the segments in  $S(c)$  that intersect  $s_p$ . Then

- (2)  $p \text{ in } c : \Leftrightarrow \neg p \text{ on } c \wedge |S_r(c)| + |S_i(c)| \text{ is odd}$
- (3)  $p \text{ out } c : \Leftrightarrow \neg (p \text{ on } c \vee p \text{ in } c)$

**Figure 8. Possible relationships between two R-cycles**

Hence,  $c$  partitions the set  $P_N$  into three subsets  $P_{in}(c)$ ,  $P_{on}(c)$ , and  $P_{out}(c)$ . Let  $P(c) := P_{on}(c) \cup P_{in}(c)$ .

Cycles are interesting because they are the basic entities for the definition of regions over realms. The relationships shown in Figure 8 may be distinguished between two  $R$ -cycles  $c_1$  and  $c_2$ . The following terminology is introduced for these configurations:

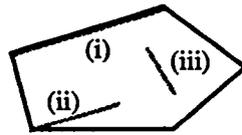
$c_2$ is	$c_1$ and $c_2$ are
• <i>(area-)inside</i> (i,ii,iii)	• <i>area-disjoint</i> (iv,v,vi)
• <i>edge-inside</i> (ii,iii)	• <i>edge-disjoint</i> (v,vi)
• <i>vertex-inside</i> (iii)	• <i>(vertex-)disjoint</i> (vi)

$c_1$ .

This means that (i)  $c_2$  is (w.r.t. *area*) *inside*  $c_1$ ; (ii)  $c_2$  additionally has no common edges with  $c_1$ ; and (iii)  $c_2$  has not even common vertices with  $c_1$ . Similarly, (iv)  $c_2$  is *disjoint* (w.r.t. *area*) with  $c_1$ ; (v) additionally has no common edges with  $c_1$ ; and (vi) has not even common vertices with  $c_1$ . The standard interpretation of the term *inside* is *area-inside*, and the standard interpretation of the term *disjoint* is *vertex-disjoint*. Furthermore, there are two positive notions:  $c_1$  and  $c_2$  are *adjacent* if they are *area-disjoint* and have common edges, and they *meet* if they are *edge-disjoint* and have common vertices. The predicates are formally defined as follows:

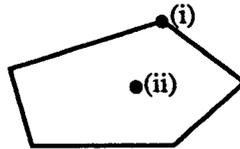
$c_1$ <i>(area-)inside</i> $c_2$	: $\Leftrightarrow P(c_1) \subseteq P(c_2)$
$c_1$ <i>edge-inside</i> $c_2$	: $\Leftrightarrow c_1$ <i>area-inside</i> $c_2 \wedge S(c_1) \cap S(c_2) = \emptyset$
$c_1$ <i>vertex-inside</i> $c_2$	: $\Leftrightarrow c_1$ <i>edge-inside</i> $c_2 \wedge P_{on}(c_1) \cap P_{on}(c_2) = \emptyset$
$c_1, c_2$ are <i>area-disjoint</i>	: $\Leftrightarrow P_{in}(c_1) \cap P(c_2) = \emptyset \wedge P_{in}(c_2) \cap P(c_1) = \emptyset$
$c_1, c_2$ are <i>edge-disjoint</i>	: $\Leftrightarrow c_1, c_2$ are <i>area-disjoint</i> $\wedge S(c_1) \cap S(c_2) = \emptyset$
$c_1, c_2$ are <i>(vertex-)disjoint</i>	: $\Leftrightarrow c_1, c_2$ are <i>edge-disjoint</i> $\wedge P_{on}(c_1) \cap P_{on}(c_2) = \emptyset$
$c_1, c_2$ are <i>adjacent</i>	: $\Leftrightarrow c_1, c_2$ are <i>area-disjoint</i> $\wedge S(c_1) \cap S(c_2) \neq \emptyset$
$c_1, c_2$ <i>meet</i>	: $\Leftrightarrow c_1, c_2$ are <i>edge-disjoint</i> $\wedge P_{on}(c_1) \cap P_{on}(c_2) \neq \emptyset$

**Figure 9. Possible relationships of  $R$ -segment lying within  $R$ -cycle**



- $s$  (area-inside  $c$  (i, ii, iii))
- $s$  (edge-inside  $c$  (ii, iii))
- $s$  (vertex-inside  $c$  (iii))

**Figure 10. Possible relationships of  $R$ -point lying within  $R$ -cycle**



- $p$  (area-inside  $c$  (i, ii))
- $p$  (vertex-inside  $c$  (ii))

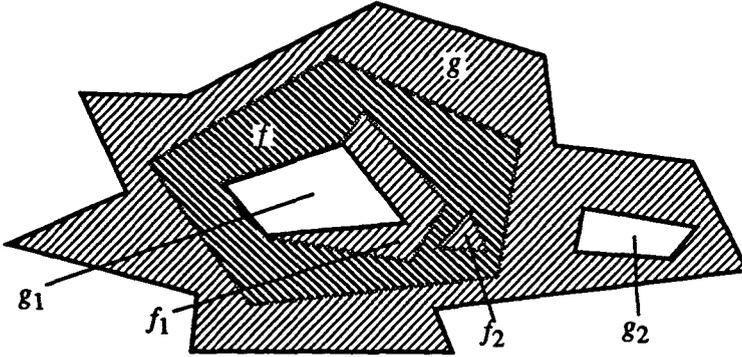
One can observe similar ways that an  $R$ -segment  $s$  can lie within an  $R$ -cycle  $c$  (Figure 9). For an  $R$ -point  $p$  and an  $R$ -cycle  $c$ , we have two possibilities (Figure 10). Formal definitions are left to the reader.

Based on the concept of  $R$ -cycles, for the definition of a SDT for regions the notions  $R$ -face and  $R$ -unit are introduced, which describe regions from two different perspectives, and which are used equivalently. Both of them essentially define polygonal regions with holes. An  $R$ -unit is a “minimal”  $R$ -face in the sense that any  $R$ -face within the  $R$ -unit is equal to the  $R$ -unit. Hence,  $R$ -units are the smallest region entities that exist over a realm. In the next section, a region (data type) will be defined that can be viewed either as a set of  $R$ -faces or, equivalently, as a set of  $R$ -units. The first view emphasizes a minimal representation of the boundary of a region, whereas the latter view supports the definition of set operations for regions.

An  $R$ -face  $f$  is a pair  $(c, H)$  where  $c$  is an  $R$ -cycle and  $H = \{h_1, \dots, h_m\}$  is a (possibly empty) set of  $R$ -cycles such that the following conditions hold (let  $S(f)$  denote the set of segments of all cycles of  $f$ ):

- (1)  $\forall i \in \{1, \dots, m\} : h_i$  edge-inside  $c$
- (2)  $\forall i, j \in \{1, \dots, m\}, i \neq j : h_i$  and  $h_j$  are edge-disjoint
- (3) Each cycle in  $S(f)$  is either equal to  $c$  or to one of the cycles in  $H$  (no other cycle can be formed from the segments of  $f$ )

The first two conditions allow a hole within a face to touch in a vertex the boundary cycle  $c$  or another hole. This is necessary to achieve closure under operations (e.g., subtracting face  $g$  from face  $f$  may lead to a hole in  $f$ ). On the other hand, to allow two holes to be area-disjoint makes no sense, because adjacent holes could be merged by eliminating common boundary segments (similarly for adjacency of a hole with the boundary). The last condition ensures uniqueness of representation, that is, there are no two different interpretations of a set of segments as sets of faces. Note that, in a given set of faces, it is entirely possible for a hole of one face to contain some other faces (“islands”).

**Figure 11. Example of relationship  $f$  area-inside  $g$** 

The grid points belonging to an  $R$ -face  $f$  are defined as  $P(f) := P(c) \setminus \bigcup_{i=1}^m$

$P_{in}(h_i)$ . The possible relationships between an  $R$ -point  $p$  or an  $R$ -segment  $s$  and an  $R$ -face  $f = (c, H)$  are:

1.  $p(\text{area-})\text{inside } f : \Leftrightarrow p \text{ area-inside } c \wedge \forall h \in H : \neg p \text{ vertex-inside } h$
2.  $s(\text{area-})\text{inside } f : \Leftrightarrow s \text{ area-inside } c \wedge \forall h \in H : \neg s \text{ edge-inside } h$

The various notions of inside and disjoint can be extended for the comparison of two  $R$ -faces  $f = (f_0, \bar{F})$  and  $g = (g_0, \bar{G})$ , for example:

$f(\text{area-})\text{inside } g : \Leftrightarrow f_0 \text{ area-inside } g_0 \wedge \forall \bar{g} \in \bar{G} : \bar{g} \text{ area-disjoint } f_0 \vee \exists \bar{f} \in \bar{F} : \bar{g} \text{ area-inside } \bar{f}$

This definition is illustrated in Figure 11.

$f \text{ area-disjoint } g$

$: \Leftrightarrow f_0 \text{ area-disjoint } g_0 \vee \exists \bar{g} \in \bar{G} : f_0 \text{ area-inside } \bar{g} \vee \exists \bar{f} \in \bar{F} : g_0 \text{ area-inside } \bar{f}$

$f \text{ edge-disjoint } g$

$: \Leftrightarrow f_0 \text{ edge-disjoint } g_0 \vee \exists \bar{g} \in \bar{G} : f_0 \text{ edge-inside } \bar{g} \vee \exists \bar{f} \in \bar{F} : g_0 \text{ edge-inside } \bar{f}$

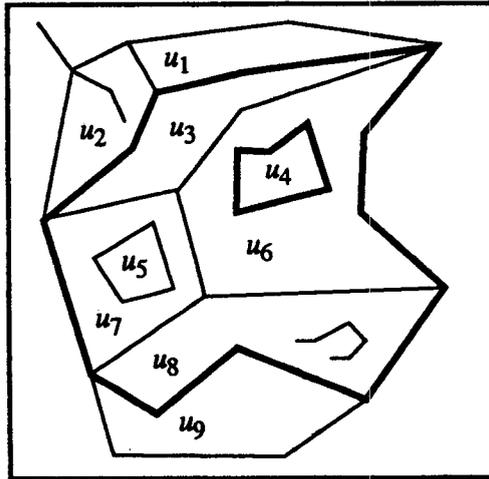
The meaning of the remaining predicates edge-inside, vertex-inside, vertex-disjoint, adjacent, and meet should be clear; definitions are omitted for brevity. We add a primitive encloses:

$f \text{ encloses } g : \Leftrightarrow \exists \bar{f} \in \bar{F} : g_0 \text{ area-inside } \bar{f}$

An  $R$ -unit as a minimal  $R$ -face is defined as follows. Let  $F(R)$  denote the set of all possible  $R$ -faces. Let  $f$  be an  $R$ -face.

$f$  is an  $R$ -unit :  $\Leftrightarrow \forall g \in F(R) : g \text{ area-inside } f \Rightarrow g = f$

**Figure 12. Example of an R-face which is not an R-unit**



We denote the set of all  $R$ -units by  $U(R)$ . Figure 12 shows an example of a realm with all its  $R$ -units  $u_i$  and an emphasized  $R$ -face which is not an  $R$ -unit.

In Güting and Schneider (1993), the equivalence of two representations of a region over a realm is formally established, namely, as a set of (pairwise) edge-disjoint  $R$ -faces, and as a set of area-disjoint  $R$ -units. Operations called *faces* and *units* are defined to convert between the two formal representations. Hence, the equivalence can be expressed as:  $\forall F \subseteq F(R): \text{faces}(\text{units}(F)) = F$ . The *units* operation is defined as  $\text{units}(F) := \{u \in U(R) \mid \exists f \in F : u \text{ area-inside } f\}$ . The *faces* operation works as follows: From a given set of area-disjoint  $R$ -units, its multiset of boundary segments is formed. Then, all segments occurring twice are removed. The remaining set of segments defines uniquely a set of edge-disjoint  $R$ -faces. As a result, we can now convert freely between the two formal representations, and use the more convenient one in the definition of operations.

Let  $T$  be a set of  $R$ -segments, that is,  $T \subseteq S$ . Then,  $\text{cycles}(T)$  denotes the set of all cycles (in the graph interpretation of realm  $R$ ) that can be formed from segments in  $T$ . Furthermore, we say that a set  $T$  of  $R$ -segments *describes a set of pairwise edge-disjoint R-faces* :  $\Leftrightarrow$  there is a set of edge-disjoint  $R$ -faces  $F$  such that  $T = S(F)$ . If  $T$  describes a set of edge-disjoint  $R$ -faces, then a function  $\text{regions}(T)$  is defined to return this set of faces.

For the definition of an SDT for lines, the notion of an  $R$ -block is introduced. A set  $T$  of  $R$ -segments is called *connected* :  $\Leftrightarrow \forall r, t \in T \exists s_1, \dots, s_m \in T : r = s_1, t = s_m, \text{ and } \forall i \in \{1, \dots, m - 1\} : s_i \text{ and } s_{i+1} \text{ meet}$ . An  $R$ -block  $b$  is a connected subgraph in the graph interpretation of a realm, defined by its set of  $R$ -segments

**Figure 13.  $p$  is meeting point,  $p'$  is not meeting point**

$S(b)$ . Two  $R$ -blocks  $b_1$  and  $b_2$  are disjoint :  $\Leftrightarrow \forall s_1 \in S(b_1) \forall s_2 \in S(b_2) : s_1$  and  $s_2$  are disjoint. For an  $R$ -point  $p$ , we consider the angularly sorted cyclic list  $L_p$  of  $R$ -segments  $s \in S(b_1) \cup S(b_2)$  that meet in  $p$ .  $p$  is called a meeting point if  $L_p$  is the concatenation of two sublists  $L_{p,1}$  and  $L_{p,2}$  so that all  $R$ -segments of  $L_{p,1}$  are elements of  $S(b_1)$  and all  $R$ -segments of  $L_{p,2}$  are elements of  $S(b_2)$ , or vice versa (see Figure 13).

Let  $b_1$  and  $b_2$  be two  $R$ -blocks.

$b_1$  and  $b_2$  meet :  $\Leftrightarrow \exists s \in S(b_1) \exists t \in S(b_2) : s$  and  $t$  meet in a meeting point  $\wedge$   
 $\forall s \in S(b_1) \forall t \in S(b_2) : s \neq t \wedge$

( $s$  and  $t$  meet in  $p \Rightarrow p$  is a meeting point)

$b_1$  and  $b_2$  intersect :  $\Leftrightarrow \forall s \in S(b_1) \forall t \in S(b_2) : s \neq t \wedge \exists s \in S(b_1) \exists t \in S(b_2) :$   
 $s$  and  $t$  meet in  $p \wedge p$  is not a meeting point

Again, we have two equivalent representations of a lines value, namely, as a set of segments, or as a set of disjoint  $R$ -blocks. For a set of segments  $T \subseteq S$ ,  $blocks(T)$  denotes its partition into maximal connected components. Then,  $S(blocks(T)) = T$ .

Some primitives relate an  $R$ -block  $b$  and an  $R$ -face  $f$ .

$b$  (area-)inside  $f$  :  $\Leftrightarrow \forall s \in S(b) : s$  area-inside  $f$

$b$  and  $f$  meet :  $\Leftrightarrow \forall s \in S(b) : \neg s$  area-inside  $f \wedge \exists s \in S(b) \exists t \in S(f) :$   
 $s$  and  $t$  meet

$b$  and  $f$  intersect :  $\Leftrightarrow \exists s \in S(b) : s$  area-inside  $f$

Embedding  $N$ -points in the Euclidean plane, we can define the distance dist ( $p, q$ ) between two  $N$ -points, the length length ( $s$ ) of an  $N$ -segment, and the area area ( $c$ ) inside an  $R$ -cycle in the usual way. The area inside an  $R$ -face  $f = (c, H)$  is defined as area ( $f$ ) := area ( $c$ ) -  $\sum_{h \in H}$  area ( $h$ ).

#### 4. Realm-Based Spatial Data Types

The realm-based structures reviewed in the previous section form the basis for a definition of spatial data types. The basic types introduced are called points, lines,

and regions,<sup>2</sup> and will be part of a spatial algebra defined in Section 7. There is a “flat” and a “structured” view of values of these types. The flat view is the following:

*For a given realm  $R$ , a value of type points is a set of  $R$ -points,  
a value of type lines is a set of  $R$ -segments, and  
a value of type regions is a set of  $R$ -units.*

The structured view that we assume as the formal definition is as follows:

*For a given realm  $R$ , a value of type points is a set of  $R$ -points,  
a value of type lines is a set of pairwise disjoint  $R$ -blocks, and  
a regions value is a set of pairwise edge-disjoint  $R$ -faces.*

In Güting and Schneider (1993), we showed that the two views are equivalent. The first view is conceptually very simple and supports a direct understanding of set operations. The second view is “semantically richer” and shows lines and regions values as consisting of a number of *components* (blocks or faces). Moreover, it allows one to express relationships between these components and also emphasizes the representation of the boundary in case of regions. Note that a regions value may have holes. Holes are important because (1) they allow for an adequate modeling of area features, and (2) they make it possible to obtain closure under point set operations. Figure 14 illustrates the data types.

It should be obvious that these data types have very nice closure properties. They are closed under the geometric operations *union*, *intersection*, and *difference* with regard to the same realm. That is, the result of such an operation is a realm-based value as well, and corresponds to the definitions of the spatial data types given above. The geometric operations can be reduced to the corresponding set-theoretic ones, and are defined as follows. Let  $P_1, P_2$  be two points values,  $L_1, L_2$  two lines values, and  $R_1, R_2$  two regions values. Then

*union* ( $P_1, P_2$ ) :=  $P_1 \cup P_2$

*union* ( $L_1, L_2$ ) := *blocks* ( $S(L_1) \cup S(L_2)$ )

*union* ( $R_1, R_2$ ) := *faces* (*units* ( $R_1$ )  $\cup$  *units* ( $R_2$ ))

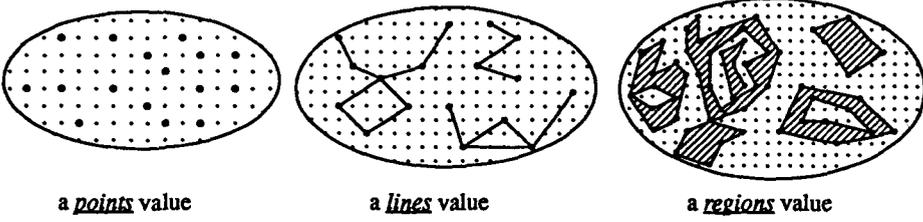
For *intersection* and *difference*, the definitions are analogous. Due to the underlying realms, these operations both in theory and in practice obey the usual algebraic laws (e.g., commutativity, associativity, distributivity).

The realm-based primitives reviewed in the previous section offer a formal basis for the definition of *spatial algebra primitives* of which *union*, *intersection*, and *difference* have just been introduced. The following further primitives are needed. Let  $F$  and  $G$  be two regions values.

---

2. Unfortunately, there is a collision between the typographical conventions for realm-based primitives and for data types (both underscore italic). It cannot be avoided to remain consistent with Güting and Schneider (1993) and Güting (1993). The latter will be used below as a framework for defining signatures.

Figure 14. Examples of spatial values



$F$  and  $G$  are *area-disjoint* :  $\Leftrightarrow \forall f \in F \forall g \in G : f$  and  $g$  are area-disjoint  
 $F$  and  $G$  are *adjacent* :  $\Leftrightarrow F$  and  $G$  are area-disjoint  $\wedge \exists f \in F \exists g \in G$   
:  $f$  and  $g$  are adjacent

The meaning of the remaining predicates (*area-*)*inside*, *edge-inside*, *vertex-inside*, *edge-disjoint*, (*vertex-*)*disjoint*, and *meet* should be clear; definitions are omitted for brevity. We define two further predicates *intersect* and *encloses*:

$F$  and  $G$  *intersect* :  $\Leftrightarrow (\text{units}(F) \cap \text{units}(G) \neq \emptyset)$   
 $F$  *encloses*  $G$  :  $\Leftrightarrow \forall g \in G \exists f \in F : f$  encloses  $g$

Let  $P$  and  $Q$  be two points values.

$P$  and  $Q$  are *disjoint* :  $\Leftrightarrow P \cap Q = \emptyset$

Let  $K$  and  $L$  be two lines values.

$K$  and  $L$  are *disjoint* :  $\Leftrightarrow \forall k \in K \forall l \in L : k$  and  $l$  are disjoint  
 $K$  and  $L$  *meet* :  $\Leftrightarrow (\forall k \in K \forall l \in L : k$  and  $l$  are disjoint  $\vee$   
 $k$  and  $l$  meet)  $\wedge (\exists k \in K \exists l \in L : k$  and  $l$  meet)  
 $K$  and  $L$  *intersect* :  $\Leftrightarrow (\forall k \in K \forall l \in L : k$  and  $l$  are disjoint  $\vee$   
 $k$  and  $l$  intersect)  $\wedge (\exists k \in K \exists l \in L : k$  and  $l$  intersect)

Let  $P$  be a points value,  $L$  a lines value,  $F$  a regions value, and  $v, w$  lines or regions values.

$P$  (*area-*)*inside*  $F$  :  $\Leftrightarrow \forall p \in P \exists f \in F : p$  area-inside  $f$   
 $L$  (*area-*)*inside*  $F$  :  $\Leftrightarrow \forall l \in L \exists f \in F : l$  area-inside  $f$   
 $L$  and  $F$  *meet* :  $\Leftrightarrow \forall l \in L \forall f \in F : \neg l$  area-inside  $f$   $\wedge$   
 $\exists l \in L \exists f \in F : l$  and  $f$  meet  
 $L$  and  $F$  *intersect* :  $\Leftrightarrow \exists l \in L \exists f \in F : l$  and  $f$  intersect  
 $P$  *on\_border\_of*  $v$  :  $\Leftrightarrow \forall p \in P \exists s = (q_1, q_2) \in S(v) : p = q_1 \vee p = q_2$   
 $v$  *border\_in\_common*  $w$  :  $\Leftrightarrow \exists s \in S(v) \exists t \in S(w) : s = t$

## 5. The Type System

The ROSE algebra is a system of spatial data types together with operations between those types. Many of the operations are applicable to several types. Hence, we need a framework and notations to describe polymorphic operations. We also need to express certain constraints for the applicability of some operations. For example, an adjacency test operation for regions should be allowed only if the two operands are known to come from a set of disjoint regions (i.e., a partition of the plane). Similarly, an overlay operation should be constrained to two partition operands and not be applicable to arbitrary collections of objects with region attributes. In this section, we briefly review a type system powerful enough to express polymorphic operations and the mentioned constraints in a precise manner.

### 5.1 Second-Order Signature

A system of several sets and functions between these sets is called a *many-sorted algebra*. A *many-sorted signature* describes the syntactic aspect of a many-sorted algebra. It consists of two sets of symbols called *sorts* and *operators*; operators are annotated with strings of sorts. Each sort is the *name of a set* of the algebra and each operator the *name of a function*. For example, the symbols lines, regions, and bool may be sorts, and *intersects*lines regions bool an operator. The annotation with sorts defines the functionality of the operator. A signature defines a set of terms.

Second-order signature, introduced in Güting (1993), is a system of two, coupled, many-sorted signatures where the top-level signature offers *kinds* (sets of types) as sorts, and *type constructors* as operators. The terms of this signature define a collection of types, that is, a type system. A simple example is shown below. Each line describes a group of operators (type constructors in this case) with the same functionality.

**kinds** DATA, GEO, SET

**type constructors**

	→ DATA	<u>int</u> , <u>real</u> , <u>bool</u>
	→ GEO	<u>points</u> , <u>lines</u> , <u>regions</u>
GEO	→ SET	<u>set</u>

Here int, set, etc. are type constructors that generally have one or more argument kinds and one result kind. A type constructor with zero argument kinds is called a *constant type*. In the example above, all constructors except for set are constant types. The terms of this signature, and therefore the available types of this type system, can be classified by result kinds. For example, there are exactly three types of kind GEO. The types of kind SET are set (points), set (lines), and set (regions). In the example, the set of types is finite, but this is generally not the case.

A second, bottom-level, signature uses the types defined by the top-level signature as sorts. Usually one does not write the bottom-level signature directly; a *signature*

*specification* allows one to quantify over kinds, and so define polymorphic operations. For example, we can define:

$$\begin{array}{llll} \forall \textit{data} \text{ in } \textit{DATA}. & \textit{data} \times \textit{data} & \rightarrow \textit{bool} & =, <, \leq, \geq, > \\ \forall \textit{geo} \text{ in } \textit{GEO}. & \textit{geo} \times \underline{\textit{regions}} & \rightarrow \textit{bool} & \textit{inside} \end{array}$$

Here, *data* and *geo* are *type variables* ranging over the kinds *DATA* and *GEO*, respectively. The semantics of such a signature specification is a many-sorted signature which is obtained by substituting for each type variable all types in the respective kind. Hence, the first specification says that the comparison operators are defined for two integers, two reals, or two boolean values. The second specification defines an **inside** operator with functionalities  $\underline{\textit{points}} \times \underline{\textit{regions}} \rightarrow \textit{bool}$ ,  $\underline{\textit{lines}} \times \underline{\textit{regions}} \rightarrow \textit{bool}$ , and  $\underline{\textit{regions}} \times \underline{\textit{regions}} \rightarrow \textit{bool}$ .

This completes the description of the basic scheme of second-order signature. Of course, there are also other ways of specifying polymorphic operations; for a discussion and references, see Gütting (1993). The basic scheme was extended in Gütting (1993) to support the definition of flexible database query languages. Some of these techniques used in this article are:

*Extensions of the concept of signature* to include “automatically” for a given collection of types (sorts, to be precise) product types, union types, list types, and function types. If  $s, s_1, \dots, s_n$  and  $t$  are sorts, then

- $(s_1 \times \dots \times s_n)$  is a sort (product sort, denoting tuples of instances of the  $s_i$ )
- $(s_1 \cup \dots \cup s_n)$  is a sort (union sort, denoting instances in any of the  $s_i$ )
- $s^+$  is a sort (the sort denoting non-empty lists of instances of  $s$ )
- $(s_1 \times \dots \times s_n \rightarrow t)$  is a sort (denoting functions from  $s_1 \times \dots \times s_n$  into  $t$ ).

With these extensions one can, for example, define the following operations:

$$\begin{array}{llll} \forall \textit{geo} \text{ in } \textit{GEO}. & & & \\ (\underline{\textit{set}}(\textit{geo}))^+ & \rightarrow \underline{\textit{set}}(\textit{geo}) & \textbf{union} & \\ \underline{\textit{set}}(\textit{geo}) \times (\textit{geo} \rightarrow \textit{bool}) & \rightarrow \underline{\textit{set}}(\textit{geo}) & \textbf{select} & \end{array}$$

Here the **union** operator takes one or more operands that are all sets of geometric values of the same type and returns a set (the union) of this type. The **select** operator takes an operand of type  $\underline{\textit{set}}(\textit{geo})$  and a predicate on type *geo* and returns a subset of the operand set fulfilling the predicate.

*Specification techniques.* Two additional specification techniques are illustrated by the following example:

$$\forall \textit{geo}_i \text{ in } \textit{GEO}. (\underline{\textit{set}}(\textit{geo}_i))^+ \rightarrow \textit{data}: \textit{DATA} \quad \textbf{weight}$$

The notation  $\textit{geo}_i$  is related to operators with a variable number of operands, and means that, for each substitution of the variable  $\textit{geo}_i$ , an instance of the kind *GEO* is selected independently. Hence, one possible operand combination for **weight** is  $\underline{\textit{set}}(\underline{\textit{points}}) \times \underline{\textit{set}}(\underline{\textit{lines}}) \times \underline{\textit{set}}(\underline{\textit{lines}})$ . With the quantification “ $\forall \textit{geo}$  in *GEO*,” all operands would have to be of the same type (e.g.,  $\underline{\textit{set}}(\underline{\textit{points}})$ ).

The notation “*data*: DATA” is to be read as “some type data in DATA,” and means that there is a type mapping associated with the **weight** operator. Intuitively, the idea is that the operator itself determines the result type within the kind DATA, depending on the given operand types. This is sometimes useful when it is not possible or desirable to describe the result type precisely in the signature. To define the semantics of such an operator, one needs to supply a type mapping function (as a part of a second-order algebra; Güting, 1993). In this example, the **weight** operator might return a value of type *int* if all operands are sets of points (and return the total number of points), and a value of type *real* otherwise (say, the total area or length). Some examples of meaningful operators with type mappings occur in the ROSE algebra defined below.

*Dynamic kinds.* (This extension was not covered in Güting, 1993). Sometimes it is necessary to modify dynamically the set of instances of a kind, that is, to create new types. For a kind  $K$ , the notation *new*( $K$ ) creates a new (anonymous) type in  $K$ ; the value of *new*( $K$ ) is a type that can be used in type expressions.

## 5.2 Type of Partition

The term *partition* refers to a disjoint subdivision of the plane into regions with associated (non-spatial) attributes. For partitions, one would like to define special operations like testing for adjacency (of two regions of a partition) or overlay (of two partitions, resulting in a new partition). The question is how partitions can be described in a type system so that the operations can be constrained to partition operands.

We feel that a partition should be modeled as a set of objects with associated *regions* attribute values and an additional constraint that for any pair of objects in one particular partition, their *regions* values are disjoint (i.e., we would like to model and manipulate sets of values such that for any two distinct values in such a set a certain condition holds). To consider an example different from partitions, let us assume we would like to model sets of integers with the property that there are no two consecutive integers in the set.

The idea to make this possible in the type system is to introduce *restriction types* and to collect them within a *special kind*. Let  $d$  be a data type and  $p$  be a binary predicate on  $d$ . Then  $d^p$  denotes a kind; each type  $d'$  in  $d^p$  describes a set of values of type  $d$  such that for any two distinct elements of  $d'$  the predicate  $p$  holds. Furthermore, any such type  $d'$  is defined to be a *subtype* of  $d$  which means that all operations defined for type  $d$  are also applicable to instances of type  $d'$ .

For the “non-consecutive integer” example, we could introduce a predicate “two-apart” on integers, being true if the difference of the two operands is at least two. Then  $\text{int}^{\text{two-apart}}$  denotes a kind whose element types have carrier sets<sup>3</sup> with

---

3. For a type, its set of instances is called the *carrier*.

the desired property. Hence, the set  $\{3, 5, 10\}$  would have a type within this kind whereas for the set  $\{1, 2, 3\}$  there would not exist a type within kind *int*<sup>two-apart</sup>. The types themselves are anonymous (i.e., no explicit names for them need to be introduced).

We use this as follows: The kind *regions*<sup>area-disjoint</sup> contains all types whose carriers are sets of *regions* values such that any two distinct values of the type are *area-disjoint*. A quantification “ $\forall$  *area* in *regions*<sup>area-disjoint</sup>” binds the *area* type variable to any such type. Hence, an adjacency test can be defined as:

$$\forall \textit{area} \textit{ in } \textit{regions}^{\textit{area-disjoint}}. \quad \textit{area} \times \textit{area} \rightarrow \textit{bool } \textit{adjacent}$$

Here, the quantification selects first one particular partition of the plane as a type *area*. Hence, it is guaranteed that any two arguments for the operator **adjacent** are from the same partition and are either area-disjoint or equal. Note that, when a new partition is created in query processing, we can obtain a corresponding new anonymous type for it with the notation *new*(*regions*<sup>area-disjoint</sup>).

On the side of the database system, this should be supported by making it possible to define restriction types and to use them as attribute domains. For example, assume an operation *area\_disjoint*, applicable to values of type *regions*, has been made known to the DBMS. One might write:

```
type mycountries = restrict (regions, area_disjoint);
class states (name: string; region: mycountries; pop: integer)
```

An insertion of a new object into class *states* should then at least conceptually be viewed as preceded by an insertion of a new *regions* value into the extension of type *mycountries*. It should be checked that the new value is *area\_disjoint* with all values already present.

## 6. Object Model Interface

Spatial data types as such are of little use unless they are integrated into a DBMS data model and query language. On the other hand, the definition of SDTs should be valid regardless of any particular data model and, therefore, not dependent on it. Consequently, SDTs should not be firmly embedded into a particular DBMS data model. Instead, the SDT definition should be based on an abstract interface to the DBMS data model, which we call the *object model interface (OMI)*. Different DBMS data models can then use the spatial algebra as a provided resource for dealing with geometry. In this section, we define an object model interface for the ROSE algebra. In fact, there are two aspects of the interface: (1) There are basic concepts and operations in the object model that are needed to *define* the ROSE algebra, and (2) there are constructs and notations needed to embed the ROSE algebra into the query language, that is, to *use* the ROSE algebra.

## 6.1 Object Model Interface Concepts for Defining the ROSE Algebra

The concepts needed to define the ROSE algebra are the following:

- object types/classes
- collections of objects
- functions for accessing (attribute) values from objects
- data types *int*, *real*, *bool*
- a pool of names (for new objects/functions)
- an object aggregation function
- an object extension function

*Object types/classes.* We assume that each DBMS data model has some notion of one or more object types or classes. For example, in a relational system, this would be relations; in an object-oriented system we may have object class hierarchies, and objects may have a complex structure. In terms of our type system, we model this by a kind OBJ; each DBMS object class is represented as a type *obj* in OBJ.

*Collections of objects.* The structures manipulated in (and obtained as a result of) queries may be sets of tuples, nested relations, sequences of object identifiers, graphs, etc. The most simple, universally valid and data model-independent abstraction is that of a *set of objects*. If a set of objects is not directly available, the DBMS data model must provide functions to transform its structures containing objects into a set of objects, and vice versa. In the type system we have a type constructor *set* applicable to object types.

*Functions for accessing attribute values.* The OMI views an object as an abstract entity whose internal structure is hidden. It is assumed that objects may have associated values of standard or spatial data types, and that these values can be accessed by means of attribute functions of type  $obj \rightarrow data$ , for any type *obj* in OBJ and data type *data*.

*Data types int, real, bool.* We assume that standard data types for integers, real numbers, and boolean values exist. Some ROSE operations yield results of these types.

*A pool of names.* Some operations require (new) names as parameters, in particular for introducing derived attributes (attribute functions). We introduce this pool of names as a type *ident* in a kind IDENT.

*Object aggregation function.* Some spatial operations construct new objects as “aggregation objects.” For that purpose the DBMS data model has to provide a  $\otimes$  (product) function which for two objects  $o_1$  of type *obj*<sub>1</sub> and  $o_2$  of type *obj*<sub>2</sub> forms an aggregation object  $o_1 \otimes o_2$ . The same symbol is used to denote a corresponding type mapping operation; hence there is also a product type *obj*<sub>1</sub>  $\otimes$  *obj*<sub>2</sub> and object  $o_1 \otimes o_2$  is of type *obj*<sub>1</sub>  $\otimes$  *obj*<sub>2</sub>. On the product type all attribute functions defined on either *obj*<sub>1</sub> or *obj*<sub>2</sub> are valid; this should be expressed by the type mapping (defined within the object model). In a relational setting, this corresponds to concatenating

two tuples when forming a join; the result tuple has the attributes of both operand tuples.

*Object extension function.* Sometimes it is necessary to add an attribute to objects of a given object type. For that purpose the DBMS data model must offer an extension function denoted by  $\oplus$ . At the instance level, this operation adds a data type value to an object, hence  $o \oplus v$  is an object  $o$  extended by a value  $v$ . At the type level, the given object type  $\underline{obj}$  is extended by an attribute function  $attr$  mapping objects into values of some data type  $\underline{data}$ . Hence,  $\underline{obj} \oplus (attr, \underline{data})$  denotes such an extension type of which  $o \oplus v$  is an instance if  $o$  has type  $\underline{obj}$  and  $v$  has type  $\underline{data}$ .

## 6.2 Embedding the ROSE Algebra into a DBMS Query Language

This part of the object model interface contains requirements about certain DBMS query language notations and constructs needed to embed and fully use the ROSE algebra. Facilities are needed to

- denote a (spatial) data type value
- denote a collection of objects together with an attribute (attribute function)
- extend objects by derived (attribute) values
- allow naming of an SDT value or a new attribute
- offer a grouping operation.

To show why these facilities are needed, we give a brief preview of some operations of the ROSE algebra:

$\forall \underline{obj}$  in OBJ.  $\forall \underline{geo}, \underline{geo}_1, \underline{geo}_2$  in GEO.

$\underline{geo} \times \underline{regions}$	$\rightarrow \underline{bool}$	<b>inside</b>
$\underline{lines} \times \underline{lines}$	$\rightarrow \underline{points}$	<b>intersection</b>
$\underline{set}(\underline{obj}) \times (\underline{obj} \rightarrow \underline{geo}_1) \times \underline{geo}_2$	$\rightarrow \underline{set}(\underline{obj})$	<b>closest</b>
$\underline{set}(\underline{obj}) \times (\underline{obj} \rightarrow \underline{geo}_1) \times \underline{ident}$	$\rightarrow \underline{set}(o:OBJ)$	<b>decompose</b>

The meaning of the first two operations should be obvious. The **closest** operator takes a collection of objects together with a spatial attribute function and a further SDT value  $v$ , and returns those objects whose attribute value is closest to  $v$  (usually one object). The **decompose** operator also takes a collection of objects with a spatial attribute. It produces a new collection of objects as follows: For each object in the operand set, its attribute value is decomposed into its components (a component is a point, a block, or a face). If there are  $n$  components, then  $n$  copies of the original object are produced, each of which has one component as the value of a new attribute. The name of the new attribute is supplied as the third parameter of type *ident*.

We now discuss each of the mentioned facilities in turn and illustrate them in the context of the relational model by (1) showing corresponding notations from geo-relational algebra (Güting, 1988a, 1988b) and (2) by extensions that might be used for SQL. In examples, these relations are used:

cities (cname: string; center: points; pop: int)  
 states (sname: string; territory: regions; language: string)

*Denote a data type value.* This is needed to supply operands to operations such as **inside** or **intersection**. There are two cases: (1) within the scope of an “object set iteration,” and (2) without object set iteration. In the first case, each object in a set is considered in turn, and it suffices to write down the name of an attribute to denote a single data type value.

Q1: Calculate the population (in thousands) of all cities in Germany.

- (a) cities **select**[center **inside** Germany] **extend**[pop/1000 {thousands}]  
 (b) **select** cname, thousands: pop/1000  
       **from** cities  
       **where** center **inside** Germany

Here, within the scope of a **select** or **extend** operator of geo-relational algebra, or within the where-clause or select-clause of SQL, we have an “object set iteration.” An attribute name denotes a data type value.

In the second case (without object set iteration), one would like to refer to a single data type value, in particular, to the attribute value of some specific object. A notation is needed to identify a single object and to access one of its attributes. In the geo-relational algebra, this is done by an **extract** operator. An error message should appear if none or more than one object is identified by the condition.

Q2: Provide the geometry of the city Hagen (assuming there is only one “Hagen” in the cities relation).

- (a) cities **extract**[cname = “Hagen”; center] {Hagen}  
 (b) **let** Hagen  
       **extract** center  
       **from** cities  
       **where** cname = “Hagen”

Here, we have extracted a single points value from the cities relation. We have also assigned a name (Hagen) to this value so that it can be used in later queries.

*Denote a collection of objects together with an attribute.* This is needed for operations like **closest** or **decompose**. Recall the signature for **closest**:

$$\underline{set}(obj) \times (obj \rightarrow geo_1) \times geo_2 \rightarrow \underline{set}(obj) \quad \mathbf{closest}$$

We need a notation to supply the two related operands  $\underline{set}(obj)$  and  $(obj \rightarrow geo_1)$ .

Q3: Determine the city or cities closest to Hagen.

- (a) cities **select**[cname  $\neq$  “Hagen”] Hagen **closest**[center]  
 (b) **closest** Hagen  
       **column** center  
       **from** cities  
       **where** cname  $\neq$  “Hagen”

In this example, “cities” corresponds to the *set* (*obj*), and “center” to the (*obj*  $\rightarrow$  *geo*<sub>1</sub>) operand. In geo-relational algebra, the set of objects is written first, and then the *points* value (the *geo*<sub>2</sub> operand); the attribute is given separately in brackets. For an extended SQL, we suggest a **column  $\alpha$  from  $\beta$**  construct to denote a set of objects  $\beta$  with an attribute  $\alpha$ . This construct should be viewed as returning the two operands separately, because they are needed by the ROSE algebra. In contrast, writing **select  $\alpha$  from  $\beta$**  would yield a set (or multiset) of attribute values, that is, an operand of type *set* (*geo*<sub>1</sub>). This is not what the operator needs; in fact, a set of values is not even available in the ROSE type system given below.

*Extend objects by derived (attribute) values.* This is needed to make the results of spatial operations available. In geo-relational algebra, this is provided by the **extend** operator; in SQL, by expressions in the select-clause, as in query Q1.

*Allow naming of an SDT value or a new attribute.* We have already seen two instances of this. In query Q2, a name (Hagen) was assigned to an SDT value. An attribute name must also be provided for derived attributes, as in query Q1. Finally, new attribute names are needed by operations that construct new objects such as **decompose**.

Q4: Decompose all states into their basic areas.

- (a) states **decompose**[territory {basic\_area}]
- (b) **decompose into column** basic\_area  
**column** territory  
**from** states

Here, (a) shows the style for naming the new attribute that would be used in geo-relational algebra (although there was no **decompose** operator). For the extended SQL, we have invented an **into column  $\alpha$**  construct for the same purpose.

*Offer a grouping operation.* This is needed to support a “fusion” operation (which essentially groups a collection of objects and forms the union of the areas in each group).

Q5: Determine all regions of the states speaking the same language.

- (a) states **fusion**[language; territory]
- (b) **fusion** territory  
**from** states  
**group by** language

These applications of the fusion operator are really abbreviations of the use of grouping:

- (a) states **group.by**[language; group **sum**[territory]]
- (b) **select sum**(territory)  
**from** states  
**group by** language

In geo-relational algebra and in SQL such a grouping operation is available; it is used together with a **sum** aggregate function of the ROSE algebra. There may be several attributes for grouping and several aggregate expressions.

## 7. The ROSE Algebra

We are now ready to define the ROSE algebra itself. It is a *realm-based algebra*, since data types are defined on realms, and since operations operate on and produce realm-based spatial values. All values occurring as operands are assumed to be defined over the *same* realm.

Defining the ROSE algebra means that we will give a second-order signature with the types points, lines, and regions, as well as types of the object model interface. The algebra then consists of carrier sets for the types and functions for the operations. The carrier sets for the three spatial types have already been defined in Section 4. In this section, we formally define the functions for all operations.

The type system of the ROSE algebra, as discussed in Sections 5 and 6, is summarized in the following specification:

**kinds** IDENT, DATA, EXT, GEO, OBJ, SET  
**type constructors**

→ IDENT	<u>ident</u>
→ DATA	<u>int</u> , <u>real</u> , <u>bool</u> , ...
→ EXT	<u>lines</u> , <u>regions</u>
→ GEO	<u>points</u> , <u>lines</u> , <u>regions</u>
OBJ → SET	<u>set</u>

Kind DATA describes the (standard) data types of the object model interface; there will be other types in addition to the three that are required. There is a kind EXT just containing types lines and regions, which supports the definition of operations not suitable for points.

The operations of the ROSE algebra are divided into four groups. For each group we give an informal introduction, show the signature, and then define the semantics of the operations.

### 7.1 Spatial Predicates

These operations compare two spatial values with respect to their topological relationships and return a boolean value. The predicates' names are self-explanatory.

$\forall geo$  in GEO.  $\forall ext, ext_1, ext_2$  in EXT.  $\forall area$  in regions<sup>area-disjoint</sup>.

$geo \times geo$	$\rightarrow$ <u>bool</u>	$=, \neq, disjoint$
$geo \times \underline{regions}$	$\rightarrow$ <u>bool</u>	<b>inside</b>
$\underline{regions} \times \underline{regions}$	$\rightarrow$ <u>bool</u>	<b>area_disjoint, edge_disjoint,</b> <b>edge_inside, vertex_inside</b>
$ext_1 \times ext_2$	$\rightarrow$ <u>bool</u>	<b>intersects, meets</b>
$area \times area$	$\rightarrow$ <u>bool</u>	<b>adjacent, encloses</b>
$\underline{points} \times ext$	$\rightarrow$ <u>bool</u>	<b>on_border_of</b>
$ext_1 \times ext_2$	$\rightarrow$ <u>bool</u>	<b>border_in_common</b>

For each operator **op** of the ROSE algebra, we define a function  $f_{op}$  that gives the operator's semantics and that has domains and codomain according to the operator's signature entry. An underlying realm  $R$  is assumed in all definitions. Of course, we rely on the primitives introduced in Sections 3 and 4.

Let  $v_1, v_2$  be two values of the same type in GEO. Then

$$f_{=} (v_1, v_2) := (v_1 = v_2)$$

$$f_{\neq} (v_1, v_2) := (v_1 \neq v_2)$$

$$f_{disjoint}(v_1, v_2) := (v_1 \text{ and } v_2 \text{ are } disjoint)$$

Let  $v$  be a value of a type in GEO, and  $F$  be a value of type regions.

$$f_{inside}(v, F) := (v \text{ inside } F)$$

Let  $v_1, v_2$  be each either a lines or a regions value.

$$f_{intersects}(v_1, v_2) := (v_1 \text{ and } v_2 \text{ intersect})$$

$$f_{meets}(v_1, v_2) := (v_1 \text{ and } v_2 \text{ meet})$$

Let  $F$  and  $G$  be two regions values of a subtype area in regions<sup>area-disjoint</sup>.

$$f_{adjacent}(F, G) := (F \text{ and } G \text{ are adjacent})$$

$$f_{encloses}(F, G) := (F \text{ encloses } G)$$

The remaining definitions are omitted; they all just lift spatial algebra primitives to the ROSE level.

## 7.2 Operators Returning Spatial Data Type Values

The second group of operations consists of *operators returning atomic spatial values* as results. The operators **intersection**, **plus**, and **minus** realize the closure properties of the ROSE algebra with respect to intersection, union, and difference of two atomic spatial values. The **common\_border** operator finds the common boundary line(s) of two regions or lines values. The **vertices** operator returns the vertex (corner) points of a lines or regions value, and produces a points value. The **contour** operator calculates a lines value from a regions value's boundary. The **interior** operator is applied to a lines value and yields a regions value, which is composed of all regions that are enclosed by segments of the lines value. If  $F$  is a regions value, **interior(contour( $F$ ))** can be used to remove all holes of  $F$ ; both operators are not inverse to each other.

$\forall$  *geo* in GEO.  $\forall$  *ext*, *ext*<sub>1</sub>, *ext*<sub>2</sub> in EXT.

<u>points</u> × <u>points</u>	→ <u>points</u>	<b>intersection</b>
<u>lines</u> × <u>lines</u>	→ <u>points</u>	<b>intersection</b>
<u>regions</u> × <u>regions</u>	→ <u>regions</u>	<b>intersection</b>
<u>regions</u> × <u>lines</u>	→ <u>lines</u>	<b>intersection</b>
<i>geo</i> × <i>geo</i>	→ <i>geo</i>	<b>plus, minus</b>
<i>ext</i> <sub>1</sub> × <i>ext</i> <sub>2</sub>	→ <u>lines</u>	<b>common_border</b>
<i>ext</i>	→ <u>points</u>	<b>vertices</b>
<u>regions</u>	→ <u>lines</u>	<b>contour</b>
<u>lines</u>	→ <u>regions</u>	<b>interior</b>

Note that the **intersection** operator applied to two lines values yields a points value, not a lines value as the set-theoretic intersection of the underlying segment sets (see operator **common\_border**).

Let *P* and *Q* be two points values, *K* and *L* be two lines values, and *F* and *G* be two regions values.

$$f_{\text{intersection}}(P, Q) := \text{intersection}(P, Q)$$

$f_{\text{intersection}}(K, L) := \{p \in R \mid \exists s \in S(K) \exists t \in S(L) : s \text{ and } t \text{ meet in } p \wedge p \text{ is not a meeting point}\}$

$$f_{\text{intersection}}(F, G) := \text{intersection}(F, G)$$

$$f_{\text{intersection}}(E, L) := \text{blocks}(\{s \in S(L) \mid \exists f \in F : s \text{ inside } f\})$$

Let *v*<sub>1</sub> and *v*<sub>2</sub> both be either two points values, two lines values, or two regions values.

$$f_{\text{plus}}(v_1, v_2) := \text{union}(v_1, v_2)$$

$$f_{\text{minus}}(v_1, v_2) := \text{difference}(v_1, v_2)$$

Let *K* and *L* be two lines values, and *F* and *G* be two regions values.

$$f_{\text{common\_border}}(K, L) := \text{intersection}(K, L)$$

$$f_{\text{common\_border}}(F, L) := f_{\text{common\_border}}(L, F) := \text{blocks}(S(F) \cap S(L))$$

$$f_{\text{common\_border}}(F, G) := \text{blocks}(S(F) \cap S(G))$$

Let *v* be a lines or regions value.

$$f_{\text{vertices}}(v) := \{p \in R \mid \exists s \in S(v) : s = (p, q)\}$$

Let  $F = \{f_1, \dots, f_n\} = \{(c_1, H_1), \dots, (c_n, H_n)\}$  be a regions value.

$$f_{\text{contour}}(F) := \text{blocks} \left( \bigcup_{i=1}^n S(c_i) \right)$$

Let *L* be a lines value.

$$f_{\text{interior}}(L) := \text{regions} \left( \bigcup_{c \in \text{cycles}(S(L))} S(c) - \right.$$

$$\left. \{s \in S(L) \mid \exists c \in \text{cycles}(S(L)) : s \text{ edge-inside } c\} \right)$$

Forming the interior of a lines value  $L$  is a somewhat more complex operation. First, the union of all segments that occur in any cycles that can be formed from the segments of  $L$  is computed. From this set of segments, all segments are removed that lie properly within (edge-inside) some cycle. Hence, only segments of “outer cycles” remain. Since these segments describe a set of edge-disjoint  $R$ -faces, the regions function can be applied to return a corresponding regions value.

### 7.3 Spatial Operators Returning Numbers

The third group of operations contains *spatial operators returning numbers*. The **no\_of\_components** operator yields the number of components ( $R$ -points,  $R$ -blocks, or  $R$ -faces) of a spatial value. The **dist** operator calculates the minimal distance between any two spatial values. The **diameter** of a spatial value is defined as the largest distance between any of its components. The **length** operator calculates the length of all segments of a lines value. The **area** operator computes the sum of the areas of all faces of a regions value. The **perimeter** operator calculates the sum of the length of all cycles of a regions value. If we intend to compute only the sum of the length of the outer cycles and not of the holes of a regions value, we can use the **contour** operator to eliminate holes first.

$\forall geo, geo_1, geo_2$  in GEO.

$geo$	$\rightarrow$ <u>int</u>	<b>no_of_components</b>
$geo_1 \times geo_2$	$\rightarrow$ <u>real</u>	<b>dist</b>
$geo$	$\rightarrow$ <u>real</u>	<b>diameter</b>
<u>lines</u>	$\rightarrow$ <u>real</u>	<b>length</b>
<u>regions</u>	$\rightarrow$ <u>real</u>	<b>area, perimeter</b>

Let  $v$  and  $w$  be values of types in GEO. Let  $L$  be a lines value, and  $F$  be a regions value.

$$\begin{aligned}
 f_{\text{no\_of\_components}}(v) &:= \text{card}(v) \\
 f_{\text{diameter}}(v) &:= \max \{ \text{dist}(p, q) \mid p, q \in f_{\text{vertices}}(v) \} \\
 f_{\text{length}}(L) &:= \sum_{s \in S(L)} \text{length}(s) \\
 f_{\text{area}}(F) &:= \sum_{f \in F} \text{area}(f) \\
 f_{\text{perimeter}}(F) &:= \sum_{s \in S(F)} \text{length}(s)
 \end{aligned}$$

Note that the four operators **diameter**, **length**, **area**, and **perimeter** are not invariant against redrawing (i.e., each of these four operations applied before and after a necessary redrawing of one or more segments of a lines or regions value will yield slightly different results). We want to define the **dist** operator in a way that is

invariant against redrawing, since it has a topological as well as a numerical aspect. Consider a set of spatial objects with a spatial attribute and a spatial reference value for which the nearest spatial object has to be computed. If the distance calculations between spatial reference value and spatial attribute value vary depending on possible redrawings, the answer regarding the nearest spatial object may vary, too, and lead to topological inconsistency. Note the relationship to the **closest** operator discussed below. Therefore, we define the distance function as follows.  $GP$  will denote the set of grid points associated with a spatial value.

For a *points* value  $v$  let  $GP(v) := v$ , for a *lines* value  $v$  let  $GP(v) := E(S(v))$  (the union of the envelope points of all segments of  $v$ ) and, for a *regions* value  $v$  let  $GP(v) := E(S(v)) \cup P_{in}(v)$ . Then

$$f_{\text{dist}}(v,w) := \begin{cases} 0, & \text{if } GP(v) \cap GP(w) \neq \emptyset \\ \min\{\underline{\text{dist}}(p,q) \mid p \in GP(v), q \in GP(w)\} & \text{otherwise} \end{cases}$$

Although the sets of grid points used in the definition may be very large, this operation can be efficiently implemented, since it can be reduced to distance computations between a point  $p$  and a segment  $s$ . It is only necessary to consider those envelope points that are neighbors of the intersection point of  $s$  with a perpendicular line going through  $p$ .

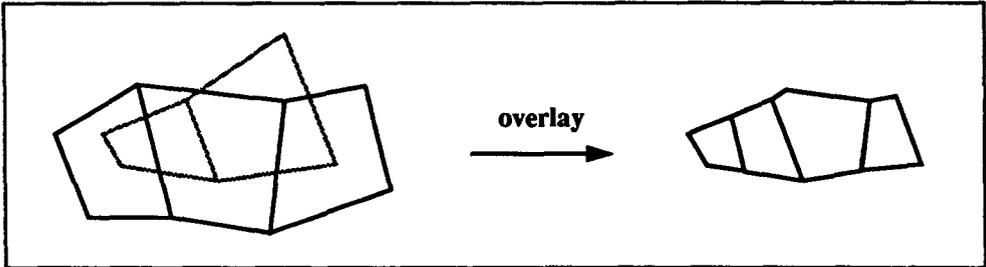
## 7.4 Spatial Operators on Sets of Objects

Operators of the last group take *sets of objects* as operands; some of them create new sets of objects as a result. The **sum** operator aggregates over the values of some spatial attribute of an object set and computes the geometric union of all these values. The **closest** operator yields that object of an object set whose spatial value is nearest to a spatial reference value. The **decompose** operator (Section 6.2) multiplies each object of an object set according to the number of components of its spatial attribute value, and adds this component as a new attribute. The **overlay** operator allows one partition of the plane to be superimposed on another, and allows them to be combined into area-disjoint regions. Partitions are given as sets of objects with an attribute of a type in *regions*<sup>area-disjoint</sup> (Section 5.2). The resulting set of objects contains one object for each new region obtained as the intersection of a region of the first partition with a region of the second partition. Note that the regions of a partition do not have to completely cover the plane. Thus, it is possible that a region of the first partition does not intersect any region of the second partition. In this case it will not be part of any new object<sup>4</sup> (Figure 15).

The **fusion** operator merges the values of a specified (set of) spatial attribute(s) on the basis of the equality of the values of another (set of) non-spatial attribute(s).

---

4. This corresponds with the standard join operation. If regions of one partition not intersecting a region of the other partition were in the result, it would be similar to an outer join.

**Figure 15. Overlaying two partitions of the plane**

For each group of equal non-spatial attribute values a (set of) new spatial value(s) is created as the geometric union of a set of spatial values of the group.<sup>5</sup> In Figure 16, a partition of districts with their land use is given. The task is to compute the regions with the same land use. Neighbor districts with the same land use are replaced by a single region (i.e., their common boundary line is erased). Each of the hatched areas on the left is part of an object describing a district. On the right, after the application of the **fusion** operator, all areas belonging to the same group  $g_i$  form a single *regions* value and are hatched in the same way.

The signature for these operations is as follows:

$\forall obj, obj_1, obj_2$  in OBJ.  $\forall geo, geo_1, geo_2$  in GEO.  $\forall area_1, area_2$  in *regions*<sup>area-disjoint</sup>.  
 $\forall data_i$  in DATA.  $\forall geo_j$  in GEO.

$\underline{set}(obj) \times (obj \rightarrow geo)$	$\rightarrow geo$	<b>sum</b>
$\underline{set}(obj) \times (obj \rightarrow geo_1) \times geo_2$	$\rightarrow \underline{set}(obj)$	<b>closest</b>
$\underline{set}(obj) \times (obj \rightarrow geo) \times \underline{ident}$	$\rightarrow \underline{set}(o: OBJ)$	<b>decompose</b>
$\underline{set}(obj_1) \times (obj_1 \rightarrow area_1) \times \underline{set}(obj_2) \times (obj_2 \rightarrow area_2) \times \underline{ident}$	$\rightarrow \underline{set}(o: OBJ)$	<b>overlay</b>
$\underline{set}(obj) \times (obj \rightarrow data_i)^+ \times (obj \rightarrow geo_j)^+$	$\rightarrow \underline{set}(o: OBJ)$	<b>fusion</b>

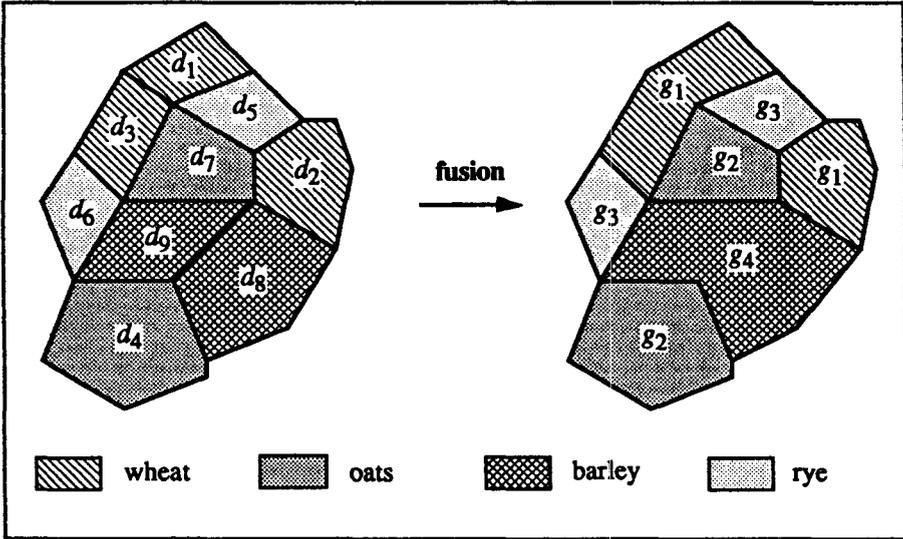
Since the operations of this group deal with sets of objects, the concepts of the object model interface are needed for their semantics definition.

For the definition of the **sum** operator let  $O = \{o_1, \dots, o_n\}$ , for  $n \geq 0$ , be the operand set of objects and *attr* the attribute function yielding an SDT value for each object.

$$f_{\text{sum}}(O, \text{attr}) := \begin{cases} \text{union}(\dots(\text{union}(\text{attr}(o_1), \text{attr}(o_2)), \dots), \text{attr}(o_n)) & \text{if } O \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

5. The **fusion** operator could be extended to allow grouping also by spatial attributes. For efficient implementation, this requires a capability of sorting by spatial data type values, which means the ROSE algebra would have to provide a “less-than” operator for each of the three SDTs imposing a linear order.

Figure 16. Merging a partition of districts with same land use



For the definition of the **closest** operator, let  $O$  be the set of objects,  $attr$  the attribute function, and  $rv$  the reference value for which the nearest spatial value has to be calculated. Then

$$f_{closest}(O, attr, rv) := \{o \in O \mid \forall o' \in O: f_{dist}(rv, attr(o)) \leq f_{dist}(rv, attr(o'))\}$$

The **decompose** operator has an unspecified result type in OBJ; hence, in addition to its semantics function  $f_{decompose}$ , it needs a type mapping  $\tau_{decompose}$  (Section 5.1). When an operator **alpha** with a type mapping is used in a query and applied to some operands (say **alpha**( $a, b, c$ )), then this leads to a call of its semantics function  $f_{alpha}(a, b, c)$  during query execution. Additionally, it leads to a call of the type mapping function  $\tau_{alpha}$  during query parsing; the type mapping function is not called with the actual operands (i.e.,  $a, b, c$ ), instead it is called with the *actual types of these operands*. These types can vary because of the polymorphic specification of operators, which is the reason why type mappings are needed at all. The only exceptions to this rule are operands of type *ident*; for them the actual identifier is passed to the type mapping function. This is because the main purpose of such operands is the use in type mappings.

$$f_{decompose}(O, attr, name) := \{o \oplus v \mid o \in O, v \in attr(o)\}$$

$$\tau_{decompose}(\underline{set}(obj), (obj \rightarrow geo), name) := obj \oplus (name, geo)$$

Hence, each object is extended by one of the components of its spatial attribute; the

new object type is an extension of the operand object type by a new attribute *name* of type *geo*. For example, the call in query Q4 (Section 6.2) “**decompose**(states, territory, basic\_area)” leads to the following calls of semantics function and type mapping:

$$f_{\text{decompose}}(\text{states}, \text{territory}, \text{basic\_area})$$

$$\tau_{\text{decompose}}(\underline{\text{set}}(\text{state}), (\text{state} \rightarrow \text{regions}), \text{basic\_area})$$

The overlay operator also needs a type mapping:

$$f_{\text{overlay}}(O_1, \text{attr}_1, O_2, \text{attr}_2, \text{name})$$

$$:= \{(o_1 \otimes o_2) \oplus v \mid \exists o_1 \in O_1 \exists o_2 \in O_2 :$$

$$f_{\text{intersects}}(\text{attr}_1(o_1), \text{attr}_2(o_2)) = \text{true} \wedge v = f_{\text{intersection}}(\text{attr}_1(o_1), \text{attr}_2(o_2))\}$$

$$\tau_{\text{overlay}}(\underline{\text{set}}(\text{obj}_1), (\text{obj}_1 \rightarrow \text{area}_1), \underline{\text{set}}(\text{obj}_2), (\text{obj}_2 \rightarrow \text{area}_2), \text{name})$$

$$:= (\text{obj}_1 \otimes \text{obj}_2) \oplus (\text{name}, \text{new}(\underline{\text{regions}}^{\text{area-disjoint}}))$$

Here, the resulting object type is the product of the two operand types extended by a new attribute *name* of a new type in the kind  $\underline{\text{regions}}^{\text{area-disjoint}}$ .

The **fusion** operator is not formally defined since it is only an abbreviation of a corresponding grouping operation (Section 6). The semantics definition would rely on a formalization of the semantics of the grouping operation.

## 8. Integration with a DBMS Query Language: O<sub>2</sub>SQL/ROSE

The purpose of this section is two-fold: (1) to show the integration of the ROSE algebra with one particular data model and query language, which further illustrates the concepts and requirements of the object model interface; (2) to demonstrate the “expressive power” of the ROSE algebra (within the context of a query language) by showing some example queries.

For the integration example, we select O<sub>2</sub> as one of the state-of-the-art object-oriented database systems, with O<sub>2</sub>SQL as its current and future standard query language (Bancilhon, 1989; Bancilhon et al., 1989, 1992; O<sub>2</sub>, 1993). O<sub>2</sub>SQL is a functional language that deals with and allows the construction of atomic values, tuples, sets and lists, provides operations on these structures, and allows one to define methods on classes. Flat as well as nested structures can be constructed, and all levels of a structure can be accessed. Elements of sets and lists and components of tuples may be of any type or class. The syntax of O<sub>2</sub>SQL has an SQL-like style through a *select-from-where* construct corresponding to the three algebraic operations projection, cartesian product, and selection, extended by object-oriented features.

In the sequel, we will demonstrate the integration of our ROSE algebra with O<sub>2</sub>SQL by presenting example queries. The notations regarding class definitions and queries comply with the notations in Bancilhon et al. (1992) and O<sub>2</sub> (1993). A few notational extensions are necessary. Examples are based on the following simple database, which models spatial aspects of Germany. The keyword **public** means that components of a tuple structure are “visible” and can be accessed.

```

class State
    public type tuple (name : string, territory : regions)
end;
class City
    public type tuple (name : string,
        zipcode : integer,
        statistical_data : tuple (foundation_date : integer,
            population : integer,
            unemployment_rate : real),
        municipal_area : regions)
end;
class Highway
    public type tuple (number : string, way : lines)
end;
class River
    public type tuple (name : string, route : lines)
end;
class District
    public type tuple (name : string, region : regions, land_use : string)
end;

```

A class is a description of a group of objects but not a persistent repository for them in a database. In  $O_2$  only objects associated with names are persistent. Therefore, we introduce for each class a named collection of objects:

```

name Cities : set(City); name States : set(State); name Highways : set(Highway);
name Rivers : set(River); name Districts : set(District);

```

Spatial attributes are defined in the same way as attributes of standard data types, using the SDTs of the ROSE algebra. However, note that we have compromised on the typing of regions attributes. In the example database, each of these attributes should really have its own type  $area_i$  within the kind regions<sup>area-disjoint</sup> to be able to model partitions of the plane. Such a sophisticated typing is not available in  $O_2$ . We will, therefore, assume that for the  $O_2$  integration the definition of the ROSE algebra is slightly changed so that all operators defined on area<sub>i</sub> types are defined on regions instead. This does not change the definition of syntax or semantics of these operators, because any value of some type  $area_i$  in regions<sup>area-disjoint</sup> is in fact a regions value; it just means that type checking cannot ensure any more that they are applied to partitions.

The syntax of the spatial operations of the ROSE algebra in a query language is not prescribed by the signature of the operations, but is part of the process of embedding the operations into the desired query language (i.e., dependent on the extended query language). Here, we select infix syntax for spatial predicates and the two operations **plus** and **minus** and a functional syntax for all other operations.

Q1: List the names and the land use of districts that are neighbors with the same land use.

```
select tuple (dname1: d1.name, dname2: d2.name, land_use: d1.land_use)
from d1 in Districts,
      d2 in Districts
where d1.region adjacent d2.region and d1.land_use = d2.land_use
```

All spatial predicates of the ROSE algebra (first group of spatial operations) can be used as selection criteria in the where clause, just like conventional predicates. The result of this query is a set of tuples, each formed by the tuple constructor *tuple*. Components of tuples are accessed by the field extraction operator, denoted by a dot. Here we have the facility of the OMI: *Denote a data type value (within an object set iteration).*

Q2: Which states are enclosed by which other states?

```
select tuple (state1: s1, state2: s2)
from s1 in States,
      s2 in States
where s1.territory encloses s2.territory
```

The result of the query is a set of tuples, each tuple being a pair of *State* objects.

Q3: Determine which highways cross which rivers and list their names, their geometries, and their crossings.

```
select tuple(name: r.name, route: r.route, number: h.number, way: h.way,
             crossing: intersection (r.route, h.way))
from r in Rivers,
      h in Highways
where r.route intersects h.way
```

Each tuple of the query result contains an attribute “crossing,” whose value is the intersection of a river and a highway value. (OMI: *Extend objects by derived attribute values, allow naming of a new attribute.*)

Q4: Associate with each state those cities lying inside that state.

```
select tuple(state: s, cities_in_state: select c
             from c in Cities
             where c.municipal_area inside s.territory)
from s in States
```

The result is a set of tuples, each tuple being a pair of a *State* object and a set of *City* objects whose geometry lies inside the geometry of the *State* object.

Q5: Which rivers partially form the boundary line of which states? In which parts do they agree?

```
select tuple(rname: r.name, sname: s.name, border:
    common_border(s.territory, r.route))
from s in States,
    r in Rivers
where s.territory border_in_common r.route
```

Q6: Compute the length of the river and highway network.

```
length (sum(select attribute way from h in Highways)
    plus sum(select attribute route from r in Rivers))
```

Here we have introduced a first extension to  $O_2$ SQL to fulfill the requirement of the OMI: *Denote a collection of objects, together with an attribute.* The notation is “**select attribute attr from s in S**” where *attr* is the name of the attribute and *S* the set of objects.<sup>6</sup> This is analogous to the “**column  $\alpha$  from  $\beta$** ” construct (Section 6.2).

In this query, it is interesting to observe that a single *lines* value is formed first, to which the length function then is applied. Using the **sum** aggregate function of  $O_2$  applicable to sets of reals, one might formulate the query as follows:

```
sum(select length(h.way) from h in Highways) +
sum(select length(r.route) from r in Rivers)
```

Actually, the result will only be the same if no two highways use the same piece of the highway network. But a more important issue to be discussed here is the view of aggregate functions. The **sum** aggregate function of  $O_2$  used in this last example is applied to a *set of values*. In contrast, the only aggregate function of the ROSE algebra (**sum**) is applied to a *set of objects with a spatial attribute*. The rationale behind this is to keep the type system of the object model interface as simple as possible. For example, in the relational model sets of values are not available. The ROSE algebra only assumes that collections of objects and atomic values exist.

Q7: Calculate the perimeter of Bavaria (class *State* is assumed to describe states within Germany).

```
perimeter(element(select s.territory from s in States where s.name = "Bavaria"))
```

The  $O_2$ SQL *element* operator extracts the unique element of a singleton set. This is exactly the facility “*denote a data type value (without object set iteration)*” of the OMI.

---

6. For the SQL embedding (Section 6.2), we have used a keyword “column.” This seemed to fit with SQL, which also speaks of “tables” rather than relations. For  $O_2$ , which used terms like “tuple,” a keyword “attribute” appears adequate. Of course, this is just a matter of taste.

The expression “**element ...**” denotes the territory of Bavaria.

Q8: Calculate the region outside Bavaria where wheat is cultivated.

```
sum(select attribute region
     from d in Districts
     where d.land_use = “wheat”)
minus
element(select s.territory
         from s in States
         where s.name = “Bavaria”)
```

This query yields an atomic spatial value.

Q9: Determine all cities that are located in areas which are completely enclosed by highways.

```
select c
from c in Cities
where c.municipal_area inside interior(sum(select attribute way from
                                           h in Highways))
```

This query yields a set of *City* objects fulfilling the where condition.

Q10: Check if the highways form a connected network.

```
no_of_components(sum(select attribute way from h in Highways)) = 1
```

Q11: List the name(s) of the highway(s) being closest to Munich.

```
define Munich as
element(select c.municipal_area from c in Cities where c.name = “Munich”);
select h.number
from h in closest(select attribute way from h in Highways, Munich)
```

In the first step, a *named query* of  $O_2$  defines Munich as a *regions* value. This is the facility “*allow naming of an SDT value*” of the OMI. The **closest** operator takes as operands a class or any other homogeneous set of objects together with a spatial attribute defined on that object type, and a spatial reference value (in this case Munich). It returns a set of objects that can be used in a query at all those positions where a set expression is allowed.

Q12: Determine the component regions of the state Schleswig-Holstein (which consists of a main land area as well as several islands in the North Sea).

```
select s.component
from s in
  decompose into component
  select attribute territory
  from s in States
  where s.name = “Schleswig-Holstein”
```

The **decompose** operator has three arguments: a class or any other homogeneous set of objects, an SDT attribute to be decomposed, and a name for the new attribute resulting from decomposition. The query yields a set of *regions* values. Here we have introduced a second extension to O<sub>2</sub>SQL to offer the facility “allow naming of a new attribute” of the OMI, using a phrase “into attr” (Section 6.2).

Q13: Partition the state Bavaria with respect to the districts of land use.

```
overlay into districts_within_Bavaria
(select attribute territory from s in States where s.name = “Bavaria,”
select attribute region from d in Districts)
```

The result is a set of objects with a new attribute “districts\_within\_Bavaria.” Each partition for the overlay is given as a set of objects with a *regions* attribute.

Q14: Compute the regions of the same land use.

```
fusion(Districts; land_use; region)
```

The **fusion** operator requires three arguments, which are syntactically separated by semicolons: a set of objects, a list of non-spatial attributes used for grouping, and a list of spatial attributes used for geometric union. In the query above the *District* objects are grouped according to equal land use and, for each group, the geometric union of the *regions* values of the “region” attribute is formed.

O<sub>2</sub>SQL offers a grouping operator **group** so that the query can be formulated without an explicit **fusion** operator:

```
group d in Districts by (land_use: d.land_use)
with (region: sum(select attribute region from p in partition))
```

Here the **group** operator is applied to a set of *District* objects. It groups *District* objects by values of their “land\_use” attribute, and produces for each group one result tuple with two attributes. The first attribute “land\_use” receives the value of the “land\_use” attribute of the group; the second attribute “region” is determined in the **with**-clause by an expression that computes for each group the geometric union of the “region” attribute values. One can refer to the current group by a keyword **partition**.

## 9. Conclusions

In this article and the companion article (Güting and Schneider, 1993), we have defined the ROSE algebra, a system of realm-based spatial data types. After the geo-relational algebra (Güting, 1988a), which was implemented in the Gral system<sup>7</sup> (Güting, 1989, Becker and Güting, 1992), this is a second attempt to define a spatial algebra for database systems and, in some sense, it represents what we have learned

---

7. The second author, Markus Schneider, took part in this implementation effort.

in the meantime. In closing, let us summarize the highlights of the ROSE algebra. It may also be interesting to compare it to the geo-relational algebra (in the sequel, geo-algebra for short).

*General types and operations.* The ROSE algebra has very general data types to represent points, lines, and regions in the plane. For example, it is now possible to represent the whole area of a state including islands or separate land areas in a single *regions* value, or a complete highway network in a single *lines* value. On the one hand, this generality makes the spatial objects and operations conceptually more difficult, requires a quite elaborate system of definitions, and needs more effort in the implementation. This is why, in the geo-algebra, a decision was made to deal only with simple polygons and single-component objects. On the other hand, the generality is needed in applications (with Gral this became obvious when the German state of Niedersachsen had to be represented which encloses—as a hole—the state of Bremen). We feel that, through the several layers of definitions of the ROSE algebra, we have managed the complexity. Apart from the better capability to model spatial objects, an important benefit is that the types are now closed under set operations of the underlying point sets—for any type one can form union, difference (**plus**, **minus**) or aggregate over its values (**sum**) which makes the rather complex fusion operation (Scholl and Voisard, 1989; Gargano et al., 1991) a simple by-product of grouping. Also, all operations are now defined in the most general way (e.g., the **closest** operation is available for all spatial types). In contrast, in the geo-algebra it was not possible to define a difference operator on regions since it would have led to holes, and intersection had to be defined as a relation operation because a resulting set of intersection values could not be represented as a single SDT value.

*Rigorous definition.* The carrier sets of the types and the semantics of all operations have been defined completely, down to the level of simple arithmetic primitives on integers. As a result, there is no ambiguity for a programmer about the precise meaning of operations or about allowed structures. We feel this is very important because, when dealing with complex spatial structures, questions about special cases invariably come up, such as “Is it allowed that the boundary of a hole in a region touches the outer boundary?” or “Qualify two adjacent regions as intersecting?” The ROSE algebra definition gives precise answers to all such questions to an implementor and, if not to end users, at least to people writing manuals for end users.

*Numerical robustness, finite resolution.* The underlying realm provides the ROSE algebra with a discrete basis and shields it from all problems of numerical robustness. Integer coordinates can be used for the representation of SDT values; critical operations such as testing whether points lie on the border of regions become feasible. In contrast, in the geo-algebra, operations such as **meets** or **common\_border** were omitted, because—with real numbers representing coordinates of SDT values—it was not clear how these operations could be implemented in a numerically robust

way. The discrete basis also greatly simplifies the implementation of geometric algorithms for the operations. For example, to implement **common\_border** one can keep for each *lines* or *regions* value its defining segments in (x,y)-lexicographic order, and then simply scan the two lists in parallel. Many operations can be implemented by plane-sweep algorithms (Nievergelt and Preparata, 1982; Bentley and Ottmann, 1979), which are simplified here because no intersection points need to be computed; all sweep stations are known beforehand and one does not need a dynamic structure to maintain the ordered list of sweep stations.

*Data model independence, clean object model interface.* The ROSE algebra is not tied to any particular data model, but can cooperate with many models and query languages. This might have been achieved in a trivial way by omitting all operations manipulating objects (like **closest**, **overlay**) and not caring how the results of geometric operations can be used in the DBMS. Instead, we have defined an object model interface and investigated quite carefully the issues arising with the integration of the ROSE algebra into a query language. Section 8 has demonstrated that a nice integration with, for example, an object-oriented model and query language can be achieved. To our knowledge, this is the first time that the problem of interfacing a general purpose query language with a complex application-specific sublanguage has been examined in some detail. Such interfaces will be important for cooperative database systems using external computation services (Schek and Weikum, 1991).

## Open Problems and Future Work

*Implementation of the ROSE algebra.* Data structures for the three SDTs and procedures for all operations except for **dist** and the set-manipulating operations of the last group (Section 7.4) have been realized (de Ridder, 1994), and are available from the authors as a module library written in Modula-2 (de Ridder, 1995). The implementation of operations makes heavy use of the following three techniques: (1) scan or parallel scan of the *halfsegment sequence*<sup>8</sup> of one or two objects, (2) plane-sweep, and (3) graph algorithms (de Ridder, 1994, 1995). Algorithms and practical aspects of the implementation are described in Güting et al. (1995). As a next step, we plan to encapsulate this implementation within a “data type extension package,” and then connect it to the Gral system as well as to another query processor called SECONDO.

A component that allows a realm to be represented, and that offers realm operations (updates with redrawing, Güting and Schneider, 1993) is almost finished. The points and segments of a realm are stored in an LSD-tree (Henrich et al., 1989). For the problems of interfacing realms and database systems see Güting and

---

8. For a *lines* or *regions* value, its *halfsegment sequence* contains each segment of its object twice, once for the left, and once for the right end point, called the *dominating point* for this segment. The halfsegment sequence is ordered xy-lexicographically by dominating points.

Schneider (1993).

Some problems remain with the realm-based approach and need to be further investigated:

*Invariance under redrawing.* We are not satisfied with the fact that some of the numeric ROSE operations (e.g., **length**, **area**) yield slightly different results before and after a redrawing due to an update of the realm. Whereas we feel that slight numerical errors are tolerable in contrast to topological errors, this may also lead to “discrete errors.” For example, when a collection of objects is sorted by area of its regions, the order may change through a realm update. Perhaps a definition of these operations can be found that is sufficiently consistent with the geometry of the objects, but invariant under redrawing.

*Objects and operations violating realm closure.* One is still interested in spatial objects that are not part of the given realm. For example, it should be possible to draw interactively a region and then to use it in a query. The new region cannot directly be compared with realm-based objects. One possible strategy might be to insert this region temporarily into the realm, and to remove it again when the query has been processed. There may be other solutions. So far we have restricted attention to operations that are closed with respect to the underlying realm, but there are also interesting operations that leave the given realm, for example, construction of a Voronoi diagram, a convex hull, or a buffer zone around a spatial object. One should study how these can be accommodated. One strategy might be to create a new realm for the new spatial values, select a set of SDT values in the database that might interact with the new geometries, and create another “small” realm for them, and then use a “merge” operation on realms to compute all intersections.

## Acknowledgments

The authors thank the referees who read the manuscript very carefully and provided a large number of detailed questions and suggestions that helped to improve the presentation. Thanks also to Andrew Frank, Michel Scholl, and Agnès Voisard for their comments. This work was supported by the DFG (Deutsche Forschungsgemeinschaft) under grant Gu 293/1-2.

## References

- Bancilhon, F. Query languages for object-oriented database systems: Analysis and a proposal. *Proceedings of the BTW (Datenbanksysteme in Büro, Technik und Wissenschaft)*, Zürich, 1989.
- Bancilhon, F., Cluet, S., and Delobel, C. A query language for the O<sub>2</sub> object-oriented database system. *Proceedings of the Second Workshop on Database Programming Languages*, Salishan, OR, 1989.

- Bancilhon, F., Delobel, C., and Kanellakis, P. The O<sub>2</sub>Book. San Mateo, CA: Morgan-Kaufmann, 1992.
- Becker, L., and Güting, R.H. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17:247-303, 1992.
- Bentley, J.L., and Ottmann, T. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643-647, 1979.
- Egenhofer, M.J., Frank, A., and Jackson, J.P. A topological data model for spatial databases. *Proceedings of the First International Symposium on Large Spatial Databases*, Santa Barbara, CA, 1989.
- Frank A. and Kuhn, W. Cell graphs: A provable correct method for the storage of geometry. *Proceedings of the Third International Symposium on Spatial Data Handling*, Seattle, 1986.
- Gargano, M., Nardelli, E., and Talamo, M. Abstract data types for the logical modeling of complex data. *Information Systems*, 16(5):565-584, 1991.
- Greene, D. and Yao, F. Finite-resolution computational geometry. *Proceedings of the Twenty-seventh IEEE Symposium on Foundations of Computer Science*, Toronto, 1986.
- Güting, R.H. Geo-relational algebra: A model and query language for geometric database systems. *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, 1988a.
- Güting, R.H. Modeling non-standard database systems by many-sorted algebras. Fachbereich Informatik, Universität Dortmund, Report 255, 1988b.
- Güting, R.H. Gral: An extensible relational database system for geometric applications. *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989.
- Güting, R.H. Second-order signature: A tool for specifying data models, query processing, and optimization. *Proceedings of the ACM SIGMOD Conference*, Washington, DC, 1993.
- Güting, R.H., de Ridder, T., and Schneider, M. Implementation of the ROSE algebra: Efficient algorithms for realm-based spatial data types. FernUniversität Hagen, Informatik-Report 170, 1995.
- Güting, R.H. and Schneider, M. Realms: A foundation for spatial data types in database systems. *Proceedings of the Third International Symposium on Large Spatial Databases*, Singapore, 1993.
- Henrich, A., Six, H.-W., and Widmayer, P. The LSD tree: Spatial access to multidimensional point- and non-point-objects. *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989.
- Joseph, T., and Cardenas, A. PICQUERY: A high level query language for pictorial database management. *IEEE Transactions on Software Engineering*, 14:630-638, 1988.

- Lipeck, U. and Neumann, K. Modelling and manipulating objects in geoscientific databases. *Proceedings of the Fifth International Conference on the Entity-Relationship Approach*, Dijon, France, 1986.
- Nievergelt, J., and Preparata, F.P. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25:739-747, 1982.
- Orenstein, J., and Manola, F. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14:611-629, 1988.
- O<sub>2</sub> User's Manual, Version 4.1. O<sub>2</sub> Technology, 1993.
- de Ridder, T. Die ROSE-Algebra: Implementierung geometrischer Datentypen und Operationen für erweiterbare Datenbanksysteme (The ROSE algebra: Implementation of geometric data types and operations for extensible database systems). Fernuniversität Hagen, Fachbereich Informatik, Diplomarbeit (Master Thesis), 1994.
- de Ridder, T. The ROSE system. Modula-2 Program System (Source Code). Fern-Universität Hagen, Praktische Informatik IV, Software Report 1, 1995. Available as a L<sup>A</sup>T<sub>E</sub>X file for printing and/or as a compressed collection of ASCII files.
- Rossopoulos, N., Faloutsos, C., and Sellis, T. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering*, 14:639-650, 1988.
- Schek, H.J. and Weikum, G. Erweiterbarkeit, Kooperation, Föderation von Datenbanksystemen (Extensibility, Cooperation, Federation of Database Systems). *Proceedings of the BTW (Datenbanksysteme in Büro, Technik und Wissenschaft)*, Kaiserslautern, 1991.
- Scholl, M. and Voisard, A. Thematic map modeling. *Proceedings of the First International Symposium on Large Spatial Databases*, Santa Barbara, CA, 1989.
- Svensson, P. and Huang, Z. Geo-SAL: A query language for spatial data analysis. *Proceedings of the Second International Symposium on Large Spatial Databases*, Zürich, 1991.
- Tomlin, C.D. *Geographic Information Systems and Cartographic Modeling*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- Voisard, A. Bases de données géographiques: du modèle de données à l'interface utilisateur. Ph.D. Thesis, University of Paris-Sud (Centre d'Orsay), 1992.

## Appendix: Definition Layers For Realm-Based Spatial Data Types

<p>ROSE Algebra Operations</p>	<p>Objects: <i>points, lines, regions</i></p> <p>Operations: =, ≠, <i>inside, edge_inside, vertex_inside, area_disjoint, edge_disjoint, disjoint, intersects, meets, adjacent, encloses, on_border_of, border_in_common, intersection, plus, minus, common_border, vertices, contour, interior, count, dist, diameter, length, area, perimeter, sum, closest, decompose, overlay, fusion</i></p>
<p>Spatial Data Types and Spatial Algebra Primitives</p>	<p>Objects: <i>points, lines, regions</i></p> <p>Operations: <i>union, intersection, difference, (area-)inside, edge-inside, vertex-inside, area-disjoint, edge-disjoint, (vertex-)disjoint, adjacent, meet, intersect, encloses, on_border_of, border_in_common</i></p>
<p>Realms, Realm-Based Structures and Realm- Based Primitives</p>	<p>Objects: <i>R-point, R-segment; R-cycle, R-face, R-unit, R-block</i></p> <p>Operations: <i>on, in, out, (area-)inside, edge-inside, vertex-inside, area-disjoint, edge-disjoint, (vertex-)disjoint, adjacent, meet, encloses, intersect, dist, area</i></p>
<p>Robust Geometric Primitives</p>	<p>Objects: <i>N-point, N-segment</i></p> <p>Operations: =, <i>meet, overlap, intersect, disjoint, on, in, intersection, parallel, aligned</i></p>
<p>Integer Arithmetic (see [GüS93])</p>	<p>Objects: integers in the range <math>[-2n^3, 2n^3]</math> (<math>n</math> integer grid size)</p> <p>Operations: +, -, *, <i>div, mod</i>, =, ≠, &lt;, ≤, ≥, &gt;</p>