LNCS 2520, pp117-176, October 2003.
(URL: http://www.springerlink.com/link.asp?id=5d3qg18792naa7vq)


# 4 Spatio-temporal Models and Languages: An Approach Based on Data Types

Ralf Hartmut Güting[1], Michael H. Böhlen[2], Martin Erwig[1], Christian S. Jensen[2], Nikos Lorentzos[3], Enrico Nardelli[4], Markus Schneider[1], and Jose R.R. Viqueira[3]

[1] Fern Universität, Hagen, Germany
[2] Aalborg University, Denmark
[3] Agricultural University of Athens, Greece
[4] Universita Degli Studi di L'Aquila, Italy

## 4.1 Introduction

In this chapter we develop DBMS data models and query languages to deal with geometries changing over time. In contrast to most of the earlier work on this subject, these models and languages are capable of handling continuously changing geometries, or *moving objects*. We focus on two basic abstractions called *moving point* and *moving region*. A *moving point* can represent an entity for which only the position in space is relevant. A *moving region* captures moving as well as growing or shrinking regions. Examples for moving points are people, polar bears, cars, trains, or air planes; examples for moving regions are hurricanes, forest fires, or oil spills in the sea.

The main line of research presented in this chapter takes a *data type oriented* approach. The idea is to view moving points and moving regions as three-dimensional (2D-space + time) or higher-dimensional entities whose structure and behavior is captured by modeling them as abstract data types. These data types can then be integrated as attribute types into relational, object-oriented, or other DBMS data models; they can be implemented as extension packages ("data blades") for suitable extensible DBMSs. Section 4.2 explains this idea in more detail and discusses some of the basic questions related to it.

Once the basic idea is established, the next task is to design precisely a collection of types and operations that adequately reflects the objects of the real world to be modeled and is capable of expressing all (or at least, many) of the questions one would like to ask about these objects. It turns out that besides the main types of interest, moving point and moving region, a relatively large number of auxiliary data types is needed. For example, one needs a line type to represent the projection of a moving point into the plane, or a "moving real" to represent the time-dependent distance of two moving points. It then becomes crucial to achieve (i) orthogonality in the design of the type system, i.e., type constructors can be applied uniformly, (ii) genericity and consistency of operations, i.e., operations range over as many types as possible and behave consistently, and (iii) closure and consistency between structure and operations of related non-temporal and temporal types. Examples of the last aspect are

T. Sellis et al. (Eds.): Spatio-temporal Databases, LNCS 2520, pp. 117–176, 2003.

that the value of a moving region, evaluated at a certain instant of time, should be consistent with the definition of a static (non-temporal) region type, or that the time-dependent distance function between two moving points, evaluated at instant $t_0$, yields the same distance value as determining for each of the two moving points their positions $p_1$ and $p_2$ at instant $t_0$, and then taking the distance between $p_1$ and $p_2$. Section 4.3 presents such a design of types and operations in some detail; it also illustrates the expressivity of the resulting query language by example applications and queries.

Of course, when we design data types and operations, we have to specify their semantics in some way. For each type, one has to define a suitable domain (the set of values allowed for the type), and for operations one needs to define functions mapping the argument domains into the result domain. One of the fundamental questions coming up is at what *level of abstraction* one should define semantics. For example, a moving point can be defined either as a function from time into the 2D plane, or as a polyline in the three-dimensional (2D + time) space. A (static) region can be defined either as a connected subset of the plane with non-empty interior, or as a polygon with polygonal holes. The essential difference is that in the first case we define the domains of the data types just in terms of infinite sets whereas in the second case we describe certain finite representations for the types.

We will discuss this issue in a bit more depth in Section 4.2 and introduce the terms *abstract model* for the first and *discrete model* for the second level abstraction. Both levels have their respective advantages. An abstract model is relatively clean and simple; it allows one to focus on the essential concepts and not get bogged down by implementation details. However, it has no straightforward implementation. A discrete model fixes representations and is generally far more complex. It makes particular choices and thereby restricts the range of values of the abstract model that can be represented. For example, a moving point could be represented not only by a 3D polyline but also by higher order polynomial splines. Both cases (and many more) are included in the abstract model. On the other hand, once such a finite representation has been selected, it can be translated directly to data structures.

Our conclusion is that both levels of modeling are needed and that one should first design an abstract model of spatio-temporal data types and then continue by defining a corresponding discrete model. Section 4.3 describes in fact an abstract model in this sense. The definitions of semantics are given generally in terms of infinite sets.

Section 4.4 then proceeds to develop a corresponding discrete model. Finite representations for all the data types of the abstract model are introduced. Spatial objects and moving spatial objects are described by linear approximations such as polygons or polyhedra. For all the "moving" types, a *sliced representation* is introduced which represents a temporal development as a set of *units* where a unit describes the development as a certain "simple" function of time during a given time interval. In Chapter 6 of this book it is shown how the representations of the discrete model can be mapped into data structures that can

be realistically used in a DBMS environment and how example algorithms can work on these data structures efficiently.

Section 4.4 concludes the main line of research presented in this chapter. Section 4.5 entitled "Outlook" presents four other pieces of work carried out within project CHOROCHRONOS. For lack of space, these developments are presented in the form of relatively brief summaries. The first two can be viewed as extensions of the approach described above, dealing with "spatio-temporal developments" and time varying partitions of the plane. The latter two have a different focus of interest and do not deal with moving objects; they develop a spatio-temporal model over a rasterized space, and address the problem of treating legacy databases and applications when a given database is changed to include the time dimension.

## 4.2 The Data Type Approach

In this section we describe the basic idea of representing moving objects by spatio-temporal data types. After some motivation, the approach to modeling is explained, and some example queries are shown. In the last subsection, we discuss two basic issues related to the approach. This section (4.2) is based on [7].

### 4.2.1 Motivation

We are interested in *geometries changing over time*, and in particular in geometries that can change continuously, and hence in *moving objects*. In spatial databases, three fundamental abstractions of spatial objects have been identified: A *point* describes an object whose location, but not extent, is relevant, e.g. a city on a large scale map. A *line* (meaning a curve in space, usually represented as a polyline) describes facilities for moving through space or connections in space (roads, rivers, power lines, etc.). A *region* is the abstraction for an object whose extent is relevant (e.g. a forest or a lake). These terms refer to two-dimensional space, but the same abstractions are valid in three or higher-dimensional spaces.

Since lines (curves) are themselves abstractions or projections of movements, it appears that they are not the primary entities whose movements should be considered. From a practical point of view, although line values can change over time, not too many examples for moving lines come into mind. Hence it seems justified to focus first[1] on *moving points* and *moving regions*. Table 4.1 shows a list of entities that can move, and questions one might ask about their movements.

Although we focus on the general case of geometries that may change in a continuous manner (i.e. move), one should note that there is a class of applications where geometries change only in discrete steps. Examples are boundaries of states, or cadastral applications, where e.g. changes of ownership of a piece of land can only happen through specific legal actions. Our proposed way of

[1] Nevertheless, in the systematic design of Section 4.2, time-dependent line values will come into play for reasons of closure.

**Table 4.1.** Moving objects and related queries

| Moving Points | Moving Regions |
|---|---|
| People<br>• Movements of a terrorist / spy / criminal<br><br>Animals<br>• Determine trajectories of birds, whales, ...<br>• Which distance do they traverse, at which speed? How often do they stop?<br>• Where are the whales now?<br>• Did their habitats move in the last 20 years?<br><br>Satellites, spacecraft, planets<br>• Which satellites will get close to the route of this spacecraft within the next 4 hours?<br><br>Cars<br>• Taxis: Which one is closest to a passenger request position?<br>• Trucks: Which routes are used regularly?<br>• Did the trucks with dangerous goods come close to a high risk facility?<br><br>Planes<br>• Were any two planes close to a collision?<br>• Are two planes heading towards each other (going to crash)?<br>• Did planes cross the air territory of state X?<br>• At what speed does this plane move? What is its top speed?<br>• Did Iraqi planes cross the 39th degree?<br><br>Ships<br>• Are any ships heading towards shallow areas?<br>• Find "strange" movements of ships indicating illegal dumping of waste.<br><br>Rockets, missiles, tanks, submarines<br>• All kinds of military analyses | Countries<br>• What was the largest extent ever of the Roman empire?<br>• On which occasions did any two states merge? (Reunification, etc).<br>• Which states split into two or more parts?<br>• How did the Serb-occupied areas in former Yugoslavia develop over time? When was the maximal extent reached? Was Ghorazde ever part of their territory?<br><br>Forests, Lakes<br>• How fast is the Amazone rain forest shrinking?<br>• Is the dead sea shrinking? What is the minimal and maximal extent of river X during the year?<br><br>Glaciers<br>• Does the polar ice cap grow? Does it move?<br>• Where must glacier X have been at time Y (backward projection)?<br><br>Storms<br>• Where is the tornado heading? When will it reach Florida?<br><br>High/low pressure areas<br>• Where do they go? Where will they be tomorrow?<br><br>Scalar functions over space, e.g. temperature<br>• Where has the 0-degree boundary been last midnight?<br><br>People<br>• Movements of the celts etc.<br><br>Troops, armies<br>• Hannibal going over the alps. Show his trajectory. When did he pass village X?<br><br>Cancer<br>• Can we find in a series of X-ray images a growing cancer? How fast does it grow? How big was it on June 1, 1995? Why was it not discovered then?<br><br>Continents<br>• History of continental shift. |

modeling is general and includes these cases, but for them also more traditional strategies could be used.

Also, if we consider transaction time (or bitemporal) databases, it is clear that changes to geometries happen only in discrete steps through updates to the database. Hence it is clear that the description of moving objects refers first of all to valid time. So we assume that complete descriptions of moving objects are put into the database by the applications, which means we are in the framework of historical databases reflecting the current knowledge about the past[2] of the real world. Transaction time databases about moving objects may be feasible, but will not be considered initially.

There is also an interesting class of applications that can be characterized as artifacts involving space and time, such as interactive multimedia documents, virtual reality scenarios, animations, etc. The techniques developed here might be useful to keep such documents in databases and ask queries related to the space and time occurring in these documents.

### 4.2.2 Modeling

Let us assume that a database consists of a set of *object classes* (of different *types* or *schemas*). Each object class has an associated set of *objects*; each object has a number of *attributes* with values drawn from certain *domains* or *atomic data types*. Of course, there may be additional features, such as object (or oid-) valued attributes, methods, object class hierarchies, etc. But the essential features are the ones mentioned above; these are common to all data models and already given in the relational model.

We now consider extensions to the basic model to capture time and space. As far as objects are concerned, an object may be created at some time and destroyed at some later time. So we can associate a validity interval with it. As a simplification, and to be able to work with standard data models, we can even omit this validity interval, and just rely on time-dependent attribute values described next.

Besides objects, attributes describing *geometries changing over time* are of particular interest. Hence we would like to define collections of *abstract data types*, or in fact *many-sorted algebras* containing several related types and their operations, for spatial values changing over time. Two basic types are *mpoint* and *mregion*, representing a moving point and a moving region, respectively. Let us assume that purely spatial data types called *point* and *region* are given that describe a point and a region in the 2D-plane[3] (a region may consist of several disjoint areas which may have holes) as well as a type *time* that describes the valid time dimension. Then we can view the types *mpoint* and *mregion* as

[2] For certain kinds of moving objects with predetermined schedules or trajectories (e.g. spacecraft, air planes, trains) the expected future can also be recorded in the database.

[3] We restrict attention to movements in 2D space, but the approach can, of course, be used as well to describe time-dependent 3D space.

mappings from time into space, that is

$$\underline{mpoint} = \underline{time} \rightarrow \underline{point}$$
$$\underline{mregion} = \underline{time} \rightarrow \underline{region}$$

More generally, we can introduce a type constructor $\tau$ which transforms any given atomic data type $\alpha$ into a type $\tau(\alpha)$ with semantics

$$\tau(\alpha) = \underline{time} \rightarrow \alpha$$

and we can denote the types $\underline{mpoint}$ and $\underline{mregion}$ also as $\tau(\underline{point})$ and $\tau(\underline{region})$, respectively.

A value of type $\underline{mpoint}$ describing a position as a function of time can be represented as a curve in the three-dimensional space $(x, y, t)$ shown in Figure 4.1. We assume that space as well as time dimensions are continuous, i.e., isomorphic to the real numbers. (It should be possible to insert a point in time between any two given times and ask for e.g. a position at that time.)

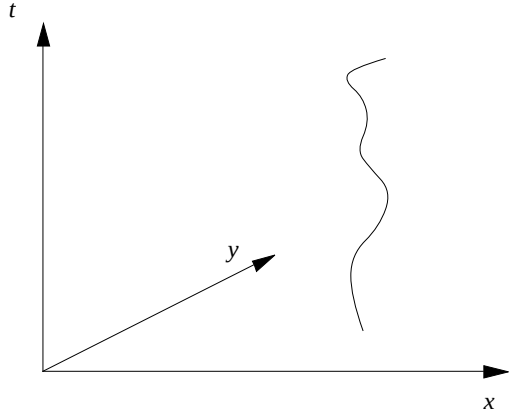



**Fig. 4.1.** A moving point

A value of type $\underline{mregion}$ is a set of volumes in the 3D space $(x, y, t)$. Any intersection of that set of volumes with a plane $t = t_0$ yields a region value, describing the moving region at time $t_0$. Of course, it is possible that this intersection is empty, and an empty region is also a proper region value.

We now describe a few example operations for these data types. For the moment, these are purely for illustrative purposes; this is in no way intended to be a closed or complete design. Such a complete design is developed in Section 4.3.

Generic operations for moving objects are, for example:

| | | |
|---|---|---|
| $\tau(\alpha) \times \underline{time}$ | $\rightarrow \alpha$ | **at** |
| $\tau(\alpha)$ | $\rightarrow \alpha$ | **minvalue**, **maxvalue** |
| $\tau(\alpha)$ | $\rightarrow \underline{time}$ | **start**, **stop** |
| $\tau(\alpha)$ | $\rightarrow \underline{real}$ | **duration** |
| $\alpha$ | $\rightarrow \tau(\alpha)$ | **const** |

Operation **at** gives the value of a moving object at a particular point in time. **Minvalue** and **maxvalue** give the minimum and maximum values of a moving object. Both functions are only defined for types $\alpha$ on which a total order exists. **Start** and **stop** return the minimum and maximum of a moving value's (time) domain, and **duration** gives the total length of time intervals a moving object is defined. We shall also use the functions *startvalue(x)* and *stopvalue(x)* as an abbreviation for **at**($x$, **start**($x$)) and **at**($x$, **stop**($x$)), respectively. Whereas all these operations assume the existence of moving objects, **const** offers a canonical way to build spatio-temporal objects: **const**($x$) is the "moving" object that yields $x$ at any time.

In particular, for moving spatial objects we may have operations such as

$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mreal}$ **mdistance**
$\underline{mpoint} \times \underline{mregion} \rightarrow \underline{mpoint}$ **visits**

**Mdistance** computes the distance between the two moving points at all times and hence returns a time changing real number, a type that we call $\underline{mreal}$ ("moving real"; $\underline{mreal} = \tau(\underline{real})$), and **visits** returns the positions of the moving point given as a first argument at the times when it was inside the moving region provided as a second argument. Here it becomes clear that a value of type $\underline{mpoint}$ may also be a partial function, in the extreme case a function where the point is undefined at all times.

Operations may also involve pure spatial or pure temporal types and other auxiliary types. For the following examples, let $\underline{line}$ be a data type describing a curve in 2D space which may consist of several disjoint pieces; it may also be self-intersecting. Let $\underline{region}$ be a type for regions in the plane which may consist of several disjoint faces with holes. Let us also have operations

$\underline{mpoint} \rightarrow \underline{line}$ **trajectory**
$\underline{mregion} \rightarrow \underline{region}$ **traversed**
$\underline{point} \times \underline{region} \rightarrow \underline{bool}$ **inside**
$\underline{line} \rightarrow \underline{real}$ **length**

Here **trajectory** is the projection of a moving point onto the plane. The corresponding projection for moving regions is the operation **traversed** that gives the total area the moving region ever has covered. **Inside** checks whether a point lies inside a region, and **length** returns the total length of a line value.

### 4.2.3 Some Example Queries

The presented data types can now be embedded into any DBMS data model as attribute data types, and the operations be used in queries. For example, we can integrate them into the relational model and have a relation

`flights (id:`$\underline{string}$`, from:`$\underline{string}$`, to:`$\underline{string}$`, route:`$\underline{mpoint}$`)`

We can then ask a query "Give me all flights from Düsseldorf that are longer than 5000 km":

```
SELECT id
FROM flights
WHERE from = "DUS" AND length(trajectory(route)) > 5000
```

This query uses projection into space. Dually, we can also formulate queries projecting into time. For example, "Which destinations can be reached from San Francisco within 2 hours?":

```
SELECT to
FROM flights
WHERE from = "SFO" AND duration(route) <= 2.0
```

Beyond projections into space and time, there are also genuine spatio-temporal questions that cannot be solved on projections. For example, "Find all pairs of planes that during their flight came closer to each other than 500 meters!":

```
SELECT A.id, B.id
FROM flights A, flights B
WHERE A.id <> B.id AND
  minvalue(mdistance(A.route, B.route)) < 0.5
```

This is in fact an instance of a spatio-temporal join.

The information contained in spatio-temporal data types is very rich. In particular, relations that would be used in traditional or spatial databases can be readily derived. For instance, we can easily define views for flight schedules and airports:

```
CREATE VIEW schedule AS
SELECT id, from, to, start(route) AS departure,
  stop(route) AS arrival
FROM flights

CREATE VIEW airport AS
SELECT DISTINCT from AS code, startvalue(route) AS location
FROM flights
```

The above examples use only one spatio-temporal relation. Even more interesting examples arise if we consider relationships between two or more different kinds of moving objects. To demonstrate this we use a further relation consisting of weather information, such as high pressure areas, storms, or temperature maps.

```
weather (kind:string, area:mregion)
```

The attribute "kind" gives the type of weather event, such as, "snow-cloud" or "tornado", and the "area" attribute provides the evolving extents of the weather features.

We can now ask, for instance, "Which flights went through a snow storm?"

```
SELECT id
FROM flights, weather
WHERE kind = "snow storm" AND duration(visits(route, area)) > 0
```

Here the expression **visits**(*route, area*) computes for each flight/storm combination a moving point that gives the movement of the plane inside this particular storm. If a flight passed a storm, this moving point is not empty, that is, it exists for a certain amount of time, which is checked by comparing the duration with 0. Similarly, we can find out which airports were affected by snow storms:

```
SELECT DISTINCT from
FROM airport, weather
WHERE kind = "snow storm" AND inside(location, traversed(area))
```

Finally, we can extend the previous query to find out which airports are most affected by snow storms. We can intersect the locations of airports with all snow storms by means of **visits** and determine the total durations:

```
SELECT code, SUM(duration(visits(const(location), area)))
  AS storm_hours
FROM airport, weather
WHERE kind = "snow storm"
GROUP BY code
HAVING storm_hours > 0
ORDER BY storm_hours
```

### 4.2.4 Some Basic Issues

Given this approach to spatio-temporal modeling and querying, several basic questions arise:

- We have seen spatio-temporal data types that are mappings from time into spatial data types. Is this realistic? How can we store them? Don't we need finite, discrete representations?
- If we use discrete representations, what do they mean? Are they observations of the moving objects?
- If we use discrete representations, how do we get the infinite entities from them that we really want to model? What kind of interpolation should be used?

In the following subsections we discuss these questions.

*Abstract vs. Discrete Modeling.* What does it mean to develop a data model with spatio-temporal data types? Actually, this is a design of a *many-sorted algebra*. There are two steps:

1. Invent a number of types and operations between them that appear to be suitable for querying. So far these are just names, which means one gives a *signature*. Formally, the signature consists of *sorts* (names for the types) and *operators* (names for the operations).
2. Define semantics for this signature, that is, associate an algebra, by defining *carrier sets* for the sorts and *functions* for the operators. So the carrier set for a type $\alpha$ contains the possible values for $\alpha$, and the functions are mappings between the carrier sets.

For a formal definition of many-sorted signature and algebra see [24] or [18]. Now one can make such designs at two different levels of abstraction, namely as *abstract* or as *discrete models*.

Abstract models allow us to make definitions in terms of infinite sets, without worrying whether finite representations of these sets exist. This allows us to view a moving point as a continuous curve in the 3D space, as an *arbitrary* mapping from an infinite time domain into an also infinite space domain. All the types that we get by application of the type constructor $\tau$ are functions over an infinite domain, hence each value is an infinite set.

This abstract view is the conceptual model that we are interested in. The curve described by a plane flying over space is continuous; for any point in time there exists a value, regardless of whether we are able to give a finite description for this mapping (or relation). In Section 4.2.2 we have in fact described the types mentioned under this view. In an abstract model, we have no problem in using types like "moving real", *mreal*, and operations like

$$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mreal} \ \textbf{mdistance}$$

since it is quite clear that at any time some distance between the moving points exists (when both are defined).

The only trouble with abstract models is that we cannot store and manipulate them in computers. Only finite and in fact reasonably small sets can be stored; data structures and algorithms have to work with *discrete (finite) representations* of the infinite point sets. From this point of view, abstract models are entirely unrealistic; only discrete models are usable.

This means we somehow need discrete models for moving points and moving regions as well as for all other involved types (*mreal*, *region*, ...). We can view discrete models as *approximations*, finite descriptions of the infinite shapes we are interested in. In spatial databases there is the same problem of giving discrete representations for in principle continuous shapes; there almost always *linear approximations* have been used. Hence, a region is described in terms of polygons and a curve in space (e.g. a river) by a polyline. Linear approximations are attractive because they are easy to handle mathematically; most algorithms in computational geometry work on linear shapes such as rectangles, polyhedra, etc. A linear approximation for a moving point is a polyline in 3D space; a linear approximation for a moving region is a set of polyhedra (see Figure 4.2). Remember that a moving point can be a partial function, hence it may disappear at times, the same is true for the moving region.

Suppose now we wish to define the type *mreal* and the operation **mdistance**. What is a discrete representation of the type *mreal*? Since we like linear approximations for the reasons mentioned above, the obvious answer would be to use a sequence of pairs (*value, time*) and use linear interpolation between the given values, similarly as for the moving point. If we now try to define the **mdistance** operator

$$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mreal} \ \textbf{mdistance}$$

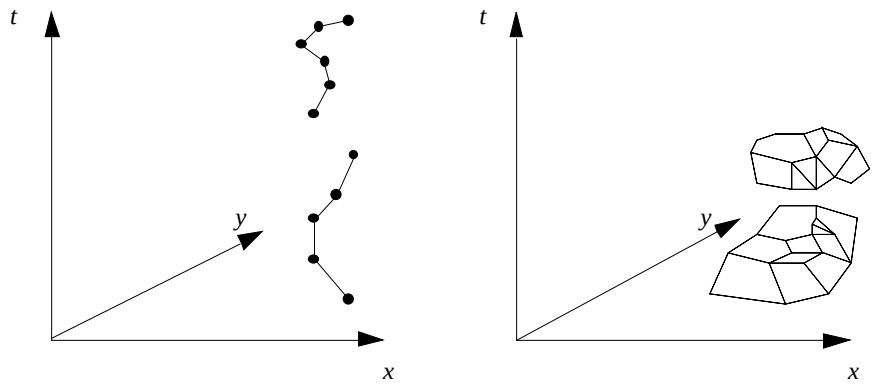



**Fig. 4.2.** Discrete representations for moving points and moving regions

we have to determine the time-dependent distance between two moving points represented as polylines. To see what that means, imagine that through each vertex of each of the two polylines we put a plane $t = t_i$ parallel to the xy-plane. Within each plane $t = t_i$ we can easily compute the distance; this will result in one of the vertices for the resulting *mreal* value. Between two adjacent planes we have to consider the distance between two line segments in 3D space (see Figure 4.3). However, this is not a linear but a quadratic function.

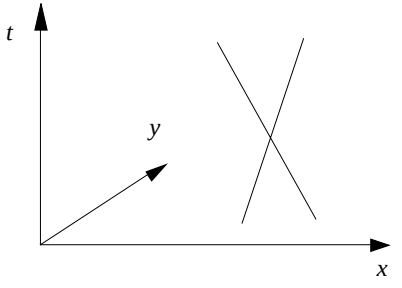



**Fig. 4.3.** Distance between segments of two moving points represented by polylines

So it seems that linear functions are not enough to represent moving reals. Maybe quadratic polynomials need to be introduced to represent the development between two vertices. But this immediately raises other questions. Why just quadratic functions motivated by the **mdistance** operation, perhaps other operations need other functions? And all kinds of operations that we need on moving reals must then be able to deal with these functions.

This example illustrates that choosing finite representations leads into difficult tradeoffs. Other choices for a moving point could be polynomial splines which are capable of describing changes in speed or acceleration much better

(with polylines, speed is stepwise constant, and acceleration is either 0 or infinite, which seems quite unnatural). For moving regions, an alternative to the polyhedral representation could be sequences of affine mappings (where each transition from one state of a region to the next can be described by translation, rotation, and scaling). This model can describe rotations much better, but does not support arbitrary changes of shape.

We have concluded from such considerations that both levels of modeling are indispensable. For the discrete model this is clear anyway, as only discrete models can be implemented. However, if we restrict attention directly to discrete models, there is a danger that a conceptually simple, elegant design of query operations is missed. This is because the representational problems might lead us to prematurely discard some options for modeling.

For example, from the discussion above one might conclude that moving reals are a problem and no such type should be introduced. But then, instead of operations **minvalue**, **maxvalue**, etc. on moving reals one has to introduce corresponding operations for each time-dependent numeric property of a moving object. Suppose we are interested in **distance** between two moving points, **speed** of a moving point, and **size** and **perimeter** of a moving region. Then we need operators **mindistance**, **maxdistance**, **minspeed**, **maxspeed**, and so forth. Clearly, this leads to a proliferation of operators and to a bad design of a query language. So the better strategy is to start with a design at the abstract level, and then to aim for that target when designing discrete models.

*Observations vs. Description of Shape.* Looking at the sequence of 3D points describing a moving point in a discrete model, one may believe that these are observations of the moving object at a certain position at a specific time. This may or may not be the case. Our view is that it is, first of all, an adequate description of the shape of a continuous curve (i.e., an approximation of that curve). We assume that the application has complete knowledge about the curve, and puts into the database a discrete description of that curve.

What is the difference to observations? Observations could mean that there are far too many points in the representation, for example, because a linear movement over an hour happens to have been observed every second. Observations could also be too few so that arbitrarily complex movements have happened between two recorded points; in that case our (linear or other) interpolation between these points could be arbitrarily wrong. Hence we assume that the application, even if it does make observations, employs some preprocessing of observations and also makes sure that enough observations are taken. Note that it is quite possible that the application adds points other than observations to a curve description, as it may know some physical laws governing the movements of this particular class of objects.

The difference in view becomes even more striking if we consider moving regions. We require the application to have complete knowledge about the 3D shape of a moving region so that it can enter into the database the polyhedron (or set of polyhedra) as a good approximation. In contrast, observations could only be a sequence of region values. But whereas for moving points it is always

possible to make a straight line interpolation between two adjacent positions, there is no way that a database system could, in general, deduce the shape of a region between two arbitrary successive observations. Hence, it is the job of the application to make enough observations and otherwise have some knowledge how regions of this kind can behave and then apply some preprocessing in order to produce a reasonable polyhedral description. How to get polyhedra from a sequence of observations, and what rules must hold to guarantee that the sequence of observations is "good enough" may be a research issue in its own right. We assume this is solved when data are put into the database.

The next two sections of this chapter will present first a careful and formal design of an abstract model, and then a discrete model offering finite representations for the types of the abstract model.

## 4.3 An Abstract Model: A Foundation for Representing and Querying Moving Objects

This section aims to offer a precise and conceptually clean foundation for implementing a spatio-temporal DBMS. It presents a simple and expressive system of abstract data types, comprising data types and encapsulating operations, that may be integrated into a query language, to yield a powerful language for querying spatio-temporal data such as moving objects. In addition to presenting the data types and operations, insight into the considerations that went into the design is offered, and the use of the abstract data types is exemplified using SQL. This section is based on [21] where complete formal definitions of all the concepts presented here can be found.

The next section defines the foundation's data types. As a precursor to defining the operations on these, Section 4.3.2 briefly presents the SQL-like language, the abstract data types are embedded into. An overview of the operations is provided in Section 4.3.3, and Sections 4.3.4 and 4.3.5 present the specific operations. Section 4.3.6 demonstrates the use of the abstract data types in a forest management application. Finally, Section 4.3.7 summarizes the section.

### 4.3.1 Spatio-temporal Data Types

This section presents a type system, constructed by introducing basic types and type constructors. Following an overview, the specific types are presented.

**Overview.** The signature (see, e.g., [18,24]) given in Table 4.2 is used in defining the type system. In this signature, *kinds* are capitalized and denote sets of types, and type constructors are in italics. This signature generates a set of terms, which are the types in the system. Terms include *int*, *region*, *moving*(*point*), *range*(*int*), etc. Type constructor *range* is applicable to all types in kinds *BASE* and *TIME*, and hence the types that can be constructed by it are *range*(*int*), *range*(*real*), *range*(*string*), *range*(*bool*), and *range*(*instant*). Type constructors with no arguments, for example *region*, are types already and are called *constant*.

**Table 4.2.** Signature describing the type system

| | | |
|---|---|---|
| | → BASE | *int*, *real*, *string*, *bool* |
| | → SPATIAL | *point*, *points*, *line*, *region* |
| | → TIME | *instant* |
| BASE ∪ TIME | → RANGE | *range* |
| BASE ∪ SPATIAL | → TEMPORAL | *intime*, *moving* |

Although the focus is on spatio-temporal types, especially *moving*(*point*) and *moving*(*region*), to obtain a closed system, it is necessary to include the other types given in the table. We proceed to describe these types in more detail, covering also their semantics.

**Base Types.** The base types are *int*, *real*, *string*, and *bool*. These have the usual semantics, and they all include an undefined value. The semantics of a type $\alpha$ is given by its carrier set, $A_\alpha$. For example, $A_{string} \triangleq V^* \cup \{\perp\}$, where $V$ is a finite alphabet. As a shorthand, we define $\bar{A}_\alpha$ to mean $A_\alpha \setminus \{\perp\}$, i.e., the carrier set without the undefined value.

**Spatial Types.** The four spatial types in the system, *point*, *points*, *line*, and *region* (cf. [19]), are illustrated in Figure 4.4. Informally, these types have the following meaning. A value of type *point* represents a point in the Euclidean plane or is undefined. A *points* value is a finite set of points. A *line* value is a finite set of continuous curves in the plane. A *region* is a finite set of disjoint parts, termed *faces*, each of which may have holes. A face may lie within a hole of another face. Each of the three set types may be empty.

a *point* value a *points* value a *line* value a *region* value

**Fig. 4.4.** The Spatial Data Types

The formal definitions, given elsewhere [21], are based on the point set paradigm and on point set topology, which provides concepts of continuity and closeness, as well as allows us to identify special topological structures of a point set, such as its interior, closure, boundary, and exterior. We assume that the reader has some familiarity with basic concepts of topology [17].

The point and point set types are quite simple to define formally. Specifically, $A_{point} \triangleq \mathbb{R}^2 \cup \{\perp\}$ and $A_{points} \triangleq \{P \subseteq \mathbb{R}^2 \mid P \text{ is finite}\}$.

The definition of *line* is based on curves (continuous mappings from $[0, 1]$ to $\mathbb{R}^2$) that are simple in the sense that the intersection of two curves yields only a

finite number of proper intersection points (disregarding common parts that are curves themselves). The *line* data type is to represent any finite union of such simple curves. When the abstract design of data types given here is implemented by some discrete design, some class of curves will be selected for representation, for example polygonal lines, curves described by cubic functions, etc. We just require that the class of curves selected has this simplicity property. This is needed, for example, to ensure that the **intersection** operation between two *line* values yields a finite set of points, representable by the *points* data type.

A finite union of curves basically yields a planar graph structure (whose nodes are intersections of curves and whose edges are intersection-free pieces of curves). Given a set of points of such a graph, there are many different sets of curves resulting in this point set. For example, a path over the graph could be interpreted as a single curve or as being composed of several curves. A design is chosen where (i) a *line* value is a point set in the plane that can be described as a finite union of curves, and (ii) there is a unique collection of curves that can serve as a "canonical" representation of this *line* value. For a *line* $Q$, we let $sc(Q)$ (the *simple curves* of $Q$) denote this representation.

When defining certain operations, we need a notion of components of a *line* value. Let *meet** denote the transitive closure of the *meet* relationship on curves. This relation partitions the components of a *line* value $Q$ into connected components, denoted as $components(sc(Q))$. The decomposition into corresponding point sets is defined as $blocks(Q) = \{points(C) \mid C \in components(sc(Q))\}$, where $points(C)$ returns the points in a partition.

A *region* value is defined as a point set in the plane with a certain structure. *region* values do not have geometric anomalies such as isolated or dangling line or point features, and missing lines and points in the form of cuts and punctures. Specifically, a *region* can be viewed as a finite set of so-called *faces*, where any two faces are disjoint except for finitely many "touching points" at their boundaries. Moreover, boundaries of faces are simple (as for lines). For example, the intersection of two regions will also produce only finitely many isolated intersection points. The boundary of a face may have outer as well as inner parts, i.e., a face may have holes.

We require that the same class of curves is used in defining the *line* and the *region* type. We can denote a given *region* value $Q$ by $faces(Q)$. We also extend the shorthand $\bar{A}$ to the spatial data types and all other types $\alpha$ whose carrier set contains sets of values. For these, we define $\bar{A}_\alpha \triangleq A_\alpha \setminus \{\emptyset\}$.

**Time Type.** Type *instant* represents time points. Time is considered to be linear and continuous, i.e., isomorphic to the real numbers. Specifically, the carrier set for *instant* is $A_{instant} \triangleq \mathbb{R} \cup \{\perp\}$.

**Temporal Types.** From the base and spatial types, we derive corresponding temporal types. Type constructor *moving* yields, for any given type $\alpha$, a mapping from time to $\alpha$.

**Definition 1.** Let $\alpha$, with carrier set $A_\alpha$, be a data type to which the *moving* type constructor is applicable. Then the carrier set for *moving*($\alpha$) is defined as follows:

$$A_{moving(\alpha)} \triangleq \{f | f : \bar{A}_{instant} \to \bar{A}_\alpha \text{ is a partial function} \wedge \Gamma(f) \text{ is finite}\}$$

Hence, each value $f$ is a function describing the development over time of a value from the carrier set of $\alpha$. The condition "$\Gamma(f)$ is finite" ensures that $f$ consists of only a finite number of continuous components (the notion of continuity used here is defined elsewhere [21]). As a result, projections of moving objects (e.g., into the 2D plane) have only a finite number of components. This is needed in the **decompose** operation (for lack of space not defined in this section, but in [21]), and it serves to making the design implementable.

For all "moving" types we introduce extra names by prefixing the argument type with an "$m$", that is, *mpoint*, *mpoints*, *mline*, *mregion*, *mint*, *mreal*, *mstring*, and *mbool*. This is just to shorten some signatures.

The temporal types obtained through the *moving* type constructor are functions, or infinite sets of pairs (instant, value). It is practical to have a type for representing any single element of such a function, i.e., a single (instant, value)-pair, for example, to represent the result of a time-slice operation. The *intime* type constructor converts a given type $\alpha$ into a type that associates *instant* values with values of $\alpha$.

**Definition 2.** Let $\alpha$ be a data type to which the *intime* type constructor is applicable, with carrier set $A_\alpha$. The carrier set for *intime*($\alpha$), is defined as follows:

$$A_{intime(\alpha)} \triangleq A_{instant} \times A_\alpha$$

**Range Types (Sets of Intervals).** For all temporal types, we desire operations that project into their domains and ranges. For the moving counterparts of the base types, e.g., *moving*(*real*) (whose values come from a one-dimensional domain), the projections are, or can be compactly represented as, sets of intervals over the one-dimensional domain. Hence, we are interested in types to represent sets of intervals over the real numbers, over the integers, etc. The *range* type constructor provides these types.

**Definition 3.** Let $\alpha$ be a data type to which the *range* type constructor is applicable (and hence on which a total order $<$ exists). An *$\alpha$-interval* is a set $X \subseteq \bar{A}_\alpha$ such that $\forall x, y \in X\ \forall z \in \bar{A}_\alpha\ (x < z < y \Rightarrow z \in X)$.

Two $\alpha$-intervals are *adjacent* if they are disjoint and their union is an $\alpha$-interval. An *$\alpha$-range* is a finite set of disjoint, non-adjacent intervals. For an $\alpha$-range $R$, *points*($R$) denotes the union of all its intervals.

**Definition 4.** Let $\alpha$ be any data type to which the *range* type constructor is applicable. Then the carrier set for *range*($\alpha$) is:

$$A_{range(\alpha)} \triangleq \{X \subseteq \bar{A}_\alpha \mid \exists \text{ an } \alpha\text{-range } R\ (X = points(R))\}$$

A *range* value $X$ has a unique associated $\alpha$-range denoted by *intvls*($X$). We use *periods* as a shorthand for ranges over the time domain, *range*(*instant*).

**Design Rationale.** We have attempted to ensure consistency and closure between non-temporal and temporal types. The former is ensured by introducing temporal types for all base types and all spatial types through the *moving* constructor. Closure under projection is ensured: For all temporal types, data types are available to represent the results of projections into (time) domain and range.

The type system offers uniform support for the point vs. point set view. All data types belong to either a one- or two-dimensional space. This principle requires that each space includes data types that represent a single value (a "point") and a set of values (a "point set"). This is the basis for the definitions of generic operations in the next sections and is explained in more detail there.

Additional discussions of design considerations may be found in [20].

### 4.3.2 Language Embedding of Abstract Data Types

In order to illustrate the use in queries of the operations to be defined in the next section, these must be embedded in a query language. We use an SQL-like language, with which most readers should be familiar. It is convenient to employ a few constructs, which are expressible in one form or another in most object-oriented or object-relational query languages. We briefly explain these constructs.

*Assignment.* The construct `LET <name> = <query>` assigns the result of `query` to `name`.

*Multistep Queries.* A query may encompass a list of initial assignments and one or more subsequent query expressions.

*Conversions between Sets of Objects and Atomic Values.* A single-attribute, single-tuple relation may be converted into a typed, atomic value and vice-versa, using the notations `ELEMENT(<query>)` and `SET(<attrname>, <value>)`. For example, expression `SET(name, "John Smith")` returns a relation with an attribute `name` and a single tuple with value `John Smith`.

*Defining Derived Attributes.* We allow arbitrary abstract data type operations in the `WHERE` clause, where they form predicates, and in the `SELECT` clause, where they produce new attributes. The notation `<new attrname> AS <expression>` is used.

*Defining Operations.* New operations may be derived from existing ones, using `LET <name> = <functional expression>`. A functional expression has the form `FUN (<parameter list>) <expression>` and corresponds to the lambda abstraction in functional languages. For example, a new operation **square** can be defined and used as follows: LET **square** = FUN (m:*integer*) m * m; **square**(5)

*Defining Aggregate Functions.* Any binary, associative, and commutative operation defined on a data type can be used as an aggregate function over a column of that data type, using the notation `AGGR(<attrname>, <operator>, <neutral element>)`. In case the argument relation is empty, the neutral element is returned. In case it has a single tuple, then that single attribute value is returned; otherwise, the existing values are combined by the given operator. Moreover, an aggregate function may be named, using the notation `LET <name> = AGGREGATE(<operator>, <neutral element>)`.

With these constructs and given a relation `employee(name:`*string*`, salary:`*int*`, permanent:`*bool*`)`, we can sum all salaries and determine whether all employees have permanent positions.

```
SELECT AGGR(salary, +, 0) FROM employee

LET all = AGGREGATE(and, TRUE);
SELECT all(permanent) FROM employee
```

### 4.3.3 Overview of Data Type Operations

The design of the operations adheres to three principles: (i) Design operations as generic as possible. (ii) Achieve consistency between operations on non-temporal and temporal types. (iii) Capture the interesting phenomena.

The first principle is crucial, as the type system is quite large. To avoid a proliferation of operations, a unifying view of collections of types is mandatory. This is enabled by relating each type to either a one-dimensional or a two-dimensional space and by considering all values as either single elements or subsets of the respective space. For example, type *int* describes single elements of the one-dimensional space of integers, while *range*(*int*) describes sets of integers. Similarly, *point* describes single elements of two-dimensional space, whereas *points*, *line*, and *region* describe subsets of this space.

Next, in order to achieve consistency of operations on non-temporal and temporal types, we first define operations on non-temporal types and then systematically *lift* these operations to become temporal variants of the respective types.

Finally, to obtain a powerful query language, it is necessary to include operations that address the most important concepts from various domains (or branches of mathematics). Whereas simple set theory and first-order logic are certainly the most fundamental and best-understood parts of query languages, operations based on order relationships, topology, metric spaces, etc., are also needed. While there is no clear recipe for achieving closure of "interesting phenomena," this motivates the inclusion of concepts and operations such as distance, size of a region, and relationships of boundaries.

Section 4.3.4 develops operations on non-temporal types, based on the generic point and point set (value vs. subset of space) view of these types. Section 4.3.5 defines operations on temporal types.

### 4.3.4 Operations on Non-temporal Types

The classes of operations on non-temporal types are given in Table 4.3, which also lists the names of the operations on these types. Although the focus is on moving objects, and hence on temporal types, the definitions of operations on non-temporal types are essential, as these operations will later be *lifted*, to obtain operations on temporal types.

**Table 4.3.** Classes of operations on non-temporal types

| Class | Operations |
|---|---|
| Predicates | **isempty**<br>$=, \neq$, **intersects**, **inside**<br>$<, \leq, \geq, >$, **before**<br>**touches**, **attached**, **overlaps**, **on_border**, **in_interior** |
| Set Operations | **intersection**, **union**, **minus**<br>**crossings**, **touch_points**, **common_border** |
| Aggregation | **min**, **max**, **avg**, **center**, **single** |
| Numeric | **no_components**, **size**, **perimeter**, **duration**, **length**, **area** |
| Distance and Direction | **distance**, **direction** |
| Base Type Specific | **and**, **or**, **not** |

We take the view that we are dealing with single values and sets of these values in one- and two-dimensional space. The types can then be classified according to Table 4.4. (Remember that "temporal types" are functions of time. Types *instant* and *periods* are not temporal types in this sense.) The table contains five

**Table 4.4.** Classification of non-temporal types

| | 1D Spaces | | | | | 2D Space |
|---|---|---|---|---|---|---|
| | discrete | | | continuous | | |
| | Integer | Boolean | String | Real | Time | 2D |
| point | *int* | *bool* | *string* | *real* | *instant* | *point* |
| point set | *range*(*int*) | *range*(*bool*) | *range* (*string*) | *range* (*real*) | *periods* | *points*, *line*, *region* |

one-dimensional spaces, Integer, Boolean, etc., and one two-dimensional space, 2D. For example, space Integer has two types, *int* and *range*(*int*). We distinguish between 1D and 2D spaces because only the 1D spaces have a (natural) total order. The distinction between discrete and continuous one-dimensional spaces is important for certain numeric operations. To have a uniform terminology, in any of the respective spaces, we call a single element a point and a subset of the space a point set; and we classify types as point types or point set types.

*Example 1.* We introduce the following example relations for use within this section, representing cities, countries, rivers, and highways in Europe.

```
city(name:string, pop:int, center:point)
country(name:string, area:region)
river(name:string, route:line)
highway(name:string, route:line)
```

**Notations for Signatures.** The notations for signatures used when defining the data type operations next are partly based on Table 4.4. We let $\pi$ and $\sigma$ be type variables that range over all point and point set types of Table 4.4, respectively. If several type variables occur in a signature (e.g., for binary operations), then they are assumed to range over types of the same space. For example, in signature $\pi \times \sigma \rightarrow \alpha$ we can select one-dimensional space Integer and instantiate $\pi$ to *int* and $\sigma$ to *range*(*int*); or we can select two-dimensional space 2D and then instantiate $\pi$ to *point* and $\sigma$ to either *points*, *line*, or *region*.

In signature $\sigma \times \sigma \rightarrow \alpha$, both arguments have to be the same type. However, in signature $\sigma_1 \times \sigma_2 \rightarrow \alpha$, type variables $\sigma_1$ and $\sigma_2$ can be instantiated independently, but must range over the same space. The notation $\alpha \otimes \beta \rightarrow \gamma$ indicates that any order of the two argument types is valid.

Some operations are restricted to certain classes of spaces, namely 1D = {Integer, Boolean, String, Real, Time}, 2D = {2D}, 1Dcont = {Real, Time}, 1Dnum = {Integer, Real, Time}, and cont = {Real, Time, 2D}. A signature is restricted to a class of spaces by putting the name of the class behind it in square brackets. For example, a signature $\alpha \rightarrow \beta$ [1D] is valid for all one-dimensional spaces.

Generic operations with generic names may have more appropriate, specific names when applied to specific types. For example, a generic **size** operation exists for point set types. For type *periods* the name **duration** is more appropriate. In this case, we introduce the more specific name as an *alias* with the notation **size**[**duration**].

In defining semantics, $u, v, \ldots$ denote single values of a $\pi$ type, and $U, V, \ldots$ generic sets of values (point sets) of a $\sigma$ type. For binary operations, $u$ or $U$ will refer to the first and $v$ or $V$ to the second argument. Furthermore, $b$ ($B$) ranges over values (sets of values) of base types, and predicates are denoted by $p$. We use $\mu$ to range over moving objects and $t$ ($T$) to range over instant values (periods).

For the definition of the semantics of operations we generally assume strict evaluation, i.e., for any function $f_{op}$ defining the semantics of an operation *op* we assume $f_{op}(\ldots, \perp, \ldots) = \perp$. Undefined arguments are therefore not handled explicitly in definitions.

**Predicates.** We consider unary and binary predicates. At this abstract level, we can ask whether a single point is undefined, and whether a point set is empty. The generic predicate **isempty**[**undefined**] is used for this purpose.

The design of binary predicates is based on the following strategy. First, we consider possible relationships between two points (single values), two point sets, and a point vs. a point set in the respective space. Second, orthogonal to this, predicates are based on three different concepts, namely set theory, order relationships, and topology. This design is shown in Table 4.5. The signatures

**Table 4.5.** Analysis of binary predicates

| | Sets | Order (1D) | Topology |
|---|---|---|---|
| point vs. point | $u = v, u \neq v$ | $u < v, u \leq v$<br>$u \geq v, u > v$ | |
| point set vs. point set | $U = V, U \neq V$<br>$U \cap V \neq \emptyset$ (**intersects**)<br>$U \subseteq V$ (**inside**) | $U$ **before** $V$ | $\partial U \cap \partial V \neq \emptyset$ (**touches**)<br>$\partial U \cap V^o \neq \emptyset$ (**attached**)<br>$U^o \cap V^o \neq \emptyset$ (**overlaps**) |
| point vs. point set | $u \in U$ (**inside**) | $u$ **before** $V$<br>$U$ **before** $v$ | $u \in \partial U$ (**on_border**)<br>$u \in U^o$ (**in_interior**) |

and definitions for these predicates are as expected and have been omitted. Predicates related to distance or direction (e.g., "north") can be obtained via numeric evaluations (see Section 4.3.4).

**Set Operations.** Set operations are fundamental and are available for all point-set types. Where feasible, we also allow set operations on point types, thus allowing expressions such as $u$ **minus** $v$ and $U$ **minus** $u$. Resulting singleton or empty sets are interpreted as point values.

Defining set operations on the combination of one- and two-dimensional point sets is more involved. This is because we are using arbitrary closed or open sets in one-dimensional space, whereas only closed point sets (*points*, *line*, and *region*) exist in two-dimensional space. This renders it necessary to apply a closure operation after applying the set operations on such entities which adds all points on the boundary of an open set.

Because there are three point set types in 2D space, an analysis of which argument type combinations make sense (return interesting results) and of what the result types are is required.

Generally, set operations may return results that intermix zero-, one-, and two-dimensional point sets, i.e., points, lines, and proper regions. Usually one is interested mainly in the result of the highest dimension. This is reflected in the concept of *regularized* set operations [32]. For example, the regularized intersection removes all lower-dimensional pieces from the corresponding intersection result. We adopt regularization in the semantics of the three "standard" 2D set operations, **union**, **minus**, and **intersection**.

The union of arguments of equal types has the usual semantics, and union on different types is not defined. Difference always results in the type of the first argument. Closure has to be applied to the result. Intersection produces

results of all dimensions smaller than or equal to the dimension of the lowest-dimensional argument. For example, the intersection of a *line* value with a *region* value may result in points and lines. We define the **intersection** operator for all type combinations with regularized semantics. To make the other parts of results available, we introduce specialized operators, e.g., **common_border** and **touch_points**.

The resulting set operations are given in Table 4.6. The notation $min(\sigma_1, \sigma_2)$ refers to taking the minimum in an assumed "dimensional" order: *points* < *line* < *region*. The table uses predicates, e.g., $is2D$, with the obvious meaning, as well as the notations $\rho(Q)$, $Q^o$, and $\partial Q$ for the closure, interior, and boundary of $Q$, respectively.

**Table 4.6.** Set operations

| Operation | Signature | | Semantics |
|---|---|---|---|
| **intersection** | $\pi \times \pi$ | $\rightarrow \pi$ | if $u = v$ then $u$ else $\perp$ |
| **minus** | $\pi \times \pi$ | $\rightarrow \pi$ | if $u = v$ then $\perp$ else $u$ |
| **intersection** | $\pi \otimes \sigma$ | $\rightarrow \pi$ | if $u \in V$ then $u$ else $\perp$ |
| **minus** | $\pi \times \sigma$ | $\rightarrow \pi$ | if $u \in V$ then $\perp$ else $u$ |
| | $\sigma \times \pi$ | $\rightarrow \sigma$ | if $is2D(U)$ then $\rho(U \setminus \{v\})$ else $U \setminus \{v\}$ |
| **union** | $\pi \otimes \sigma$ | $\rightarrow \sigma$ | if $is1D(V)$ or $type(V) =$ *points* then $V \cup \{u\}$ else $V$ |
| **intersection**, **minus**, **union** | $\sigma \times \sigma$ | $\rightarrow \sigma$ [1D] | $U \cap V, U \setminus V, U \cup V$ |
| **intersection** | $\sigma_1 \times \sigma_2$ | $\rightarrow min(\sigma_1, \sigma_2)$ [2D] | [21] |
| **minus** | $\sigma_1 \times \sigma_2$ | $\rightarrow \sigma_1$ [2D] | $\rho(Q_1 \setminus Q_2)$ |
| **union** | $\sigma \times \sigma$ | $\rightarrow \sigma$ [2D] | $Q_1 \cup Q_2$ |
| **crossings** | *line* × *line* | → *points* | [21] |
| **touch_points** | *region* ⊗ *line* | → *points* | |
| | *region* × *region* | → *points* | |
| **common_border** | *region* × *region* | → *line* | |

The following example shows how, with **union** and **intersection**, we obtain the corresponding aggregate functions over sets of objects (relations).

*Example 2.* "Determine the region of Europe from the regions of its countries."

```
LET sum = AGGREGATE(union, TheEmptyRegion);
LET Europe = SELECT sum(area) FROM country
```

**Aggregation.** Aggregation reduces sets of points to points (Table 4.7). For open and half-open intervals, we use the infimum and supremum values, i.e., the maximum and minimum of their closure, for computing minimum and maximum values. This is preferable over returning undefined values. In all domains that have addition, we can compute the average (**avg**). In 2D, the average is based

**Table 4.7.** Aggregate operations

| Operation | Signature | | | Semantics |
|---|---|---|---|---|
| **min**, **max** | $\sigma$ | $\to \pi$ | [1D] | $\min(\rho(U)), \max(\rho(U))$ |
| **avg** | $\sigma$ | $\to \pi$ | [1Dnum] | $\frac{1}{\lvert intvls(U)\rvert} \sum_{T \in intvls(U)} \frac{\sup(T)+\inf(T)}{2}$ |
| **avg**[**center**] | $\underline{points}$ | $\to \pi$ | [2D] | $\frac{1}{n} \sum_{p \in U} \overrightarrow{p}$ |
| **avg**[**center**] | $\underline{line}$ | $\to \pi$ | [2D] | $\frac{1}{\lVert U\rVert} \sum_{c \in sc(U)} \overrightarrow{c}\, \lVert c\rVert$ |
| **avg**[**center**] | $\underline{region}$ | $\to \pi$ | [2D] | $\frac{1}{M} \int_U \overrightarrow{p}\, dA$ where $M = \int_U dA$ |
| **single** | $\sigma$ | $\to \pi$ | | if $\exists u : U = \{u\}$ then $u$ else $\perp$ |

on vector addition and is usually called **center** (of gravity). It is often useful to have a "casting" operation available to transform a singleton set into its single value; operation **single** does this conversion.

*Example 3.* The query "find the point where highway A1 crosses the river Rhine" can be expressed as:

```
SELECT single(crossings(R.route, H.route))
FROM river R, highway H
WHERE R.name = "Rhine" and H.name = "A1"
  and R.route intersects H.route
```

**Numeric Properties of Sets.** For sets of points, a number of well-known numeric properties may be computed (Table 4.8). For example, the number of

**Table 4.8.** Numeric operations

| Operation | Signature | | | Semantics |
|---|---|---|---|---|
| **no_components** | $\sigma$ | $\to \underline{int}$ | [1D] | $\lvert intvls(U)\rvert$ |
| **no_components** | $\underline{points}$ | $\to \underline{int}$ | | $\lvert U\rvert$ |
| **no_components** | $\underline{line}$ | $\to \underline{int}$ | | $\lvert blocks(U)\rvert$ |
| **no_components** | $\underline{region}$ | $\to \underline{int}$ | | $\lvert faces(U)\rvert$ |
| **size**[**duration**] | $\sigma$ | $\to \underline{real}$ | [1Dcont] | $\sum_{T \in intvls(U)} \sup(T) - \inf(T)$ |
| **size**[**length**] | $\underline{line}$ | $\to \underline{real}$ | | $\lVert U\rVert$ |
| **size**[**area**] | $\underline{region}$ | $\to \underline{real}$ | | $\int_U dA$ |
| **perimeter** | $\underline{region}$ | $\to \underline{real}$ | | $f_{\mathbf{length}}(\partial U)$ |

components (**no_components**) is the number of disjoint maximal connected subsets, i.e., the number of faces for a region, connected components for a line graph, and intervals for a 1D point set. The **size** is defined for all continuous set types (i.e., for $\underline{range}(\underline{real})$, $\underline{periods}$, $\underline{line}$, and $\underline{region}$). For 1D types, the size is the sum of the lengths of component intervals; for $\underline{line}$, it is the length, and for $\underline{region}$, it is the area.

*Example 4.* "List for each country its total size and the number of disjoint land areas."

```
SELECT name, area(area), no_components(area) FROM country
```

**Distance and Direction.** A distance measure exists for all continuous types. The **distance** function determines the distance between the closest pair of a point from the first and the second argument.

The direction between points is sometimes of interest. The **direction** function returns the angle of the line from the first to the second point, measured in degrees ($0 \leq angle < 360$). Hence, if $q$ is exactly north of $p$ then **direction**$(p, q) = 90$. If $p = q$ then the undefined value $\perp$ is returned.

*Example 5.* "Find all cities north of and within 200 kms of Munich!"

```
LET Munich = ELEMENT(SELECT center FROM city
  WHERE name = "Munich");
SELECT name FROM city
WHERE distance(center, Munich) < 200
  and direction(Munich, center) >= 45
  and direction(Munich, center) <= 135
```

**Specific Operations for Base Types.** The operations **and**, **or**, and **not** on base types are also needed, although they are not related to the point vs. point set view. We mention them because they will be subject to *lifting* described below and so become applicable to temporal types.

### 4.3.5 Operations on Temporal Types

Values of temporal types (i.e., types $\underline{moving}(\alpha)$) are partial functions of the form $f : A_{instant} \rightarrow A_\alpha$. There are four classes of operations on such functions.

**Projection to Domain and Range.** For values of all $\underline{moving}$ types—which are functions—operations are provided that yield the domain and range of these functions. The domain function **deftime** : $\underline{moving}(\alpha) \rightarrow \underline{periods}$ returns the times for which a function is defined.

In 1D space, operation **rangevalues** : $\underline{moving}(\alpha) \rightarrow \underline{range}(\alpha)$ returns values assumed over time as a set of intervals. For the 2D types, operations are offered to return the parts of the projections corresponding to our data types. For example, the projection of a moving point into the plane may consist of points and lines; these can be obtained by operations **locations** and **trajectory** respectively.

For $\underline{intime}$ types, the two trivial projection operations, **inst** : $\underline{intime}(\alpha) \rightarrow \underline{instant}$ and **val** : $\underline{intime}(\alpha) \rightarrow \alpha$, are offered.

All the infinite point sets that result from domain and range projections are represented in collapsed form by the corresponding point set types. For example, a set of instants is represented as a $\underline{periods}$ value, and an infinite set of regions

is represented by the union of the points of the regions, which is represented in turn as a *region* value. This finite representation is enabled by the continuity condition required for types *moving*($\alpha$) (see Section 4.3.1).

The resulting design is complete in that all projection values in domain and range can be obtained.

*Example 6.* For illustration of operations on temporal types, we use the following relations (a slight variation of those of Section 4.2).

```
flight(airline:string, no:int, from:string, to:string, route:mpoint)
weather(name:string, kind:string, area:mregion)
site(name:string, pos:point)
```

In the first, attributes `airline` and `no` identify a flight, and the names of the departure and destination cities and the route taken for each flight are also recorded. A route is defined only for the times the plane is in flight and not when it is on the ground. Relation `weather` records named weather phenomena. Attribute `kind` gives the type of phenomenon, such as, "snow-cloud" or "tornado," and attribute `area` provides the phenomenon's evolving extent. Relation `site` contains positions of well-known sites.

*Example 7.* "How far does flight LH 257 travel in French air space?"

```
LET route257 = ELEMENT(SELECT route FROM flight
  WHERE airline = "LH" and no = 257);
length(intersection(France, trajectory(route257)))
```

"What are the departure and arrival times of flight LH 257?"

```
min(deftime(route257)); max(deftime(route257))
```

*Example 8.* "When and at distance does flight 257 pass the Eiffel tower?"
We assume a **closest** operator with signature *mpoint* $\times$ *point* $\rightarrow$ *intime*(*point*), which returns the time and position when a moving point is closest to a given fixed point in the plane. In [21] it is shown how such an operator can be derived from others.

```
LET EiffelTower =
  ELEMENT(SELECT pos FROM site WHERE name = "Eiffel Tower");
LET pass = closest(route257, EiffelTower);
inst(pass); distance(EiffelTower, val(pass))
```

**Interaction with Points and Point Sets in Domain and Range.** Some operations relate the functional values of *moving* types with values either in their (time) domain or their range. For example, such functions allow us to determine whether a moving point passes a given point or region. With these, one may also restrict a moving entity to given domain or range values. As an example, one may determine the value(s) of a moving real at time $t$ or in time interval $[t_1, t_2]$.

**Table 4.9.** Interaction of temporal values with values in domain and range

| Operation | Signature | | |
|---|---|---|---|
| **atinstant** | $\underline{moving}(\alpha) \times \underline{instant}$ | $\rightarrow \underline{intime}(\alpha)$ | |
| **atperiods** | $\underline{moving}(\alpha) \times \underline{periods}$ | $\rightarrow \underline{moving}(\alpha)$ | |
| **initial** | $\underline{moving}(\alpha)$ | $\rightarrow \underline{intime}(\alpha)$ | |
| **final** | $\underline{moving}(\alpha)$ | $\rightarrow \underline{intime}(\alpha)$ | |
| **present** | $\underline{moving}(\alpha) \times \underline{instant}$ | $\rightarrow \underline{bool}$ | |
| **present** | $\underline{moving}(\alpha) \times \underline{periods}$ | $\rightarrow \underline{bool}$ | |
| **at** | $\underline{moving}(\alpha) \times \alpha$ | $\rightarrow \underline{moving}(\alpha)$ | [1D] |
| **at** | $\underline{moving}(\alpha) \times \underline{range}(\alpha)$ | $\rightarrow \underline{moving}(\alpha)$ | [1D] |
| **at** | $\underline{moving}(\alpha) \times \underline{point}$ | $\rightarrow \underline{mpoint}$ | [2D] |
| **at** | $\underline{moving}(\alpha) \times \beta$ | $\rightarrow \underline{moving}(min(\alpha, \beta))$ | [2D] |
| **atmin** | $\underline{moving}(\alpha)$ | $\rightarrow \underline{moving}(\alpha)$ | [1D] |
| **atmax** | $\underline{moving}(\alpha)$ | $\rightarrow \underline{moving}(\alpha)$ | [1D] |
| **passes** | $\underline{moving}(\alpha) \times \beta$ | $\rightarrow \underline{bool}$ | |

The first and second groups of operations in Table 4.9 concern interactions with the (time) domain and range values, respectively.

In the first group, operations **atinstant** and **atperiods** restrict a moving entity to a given instant, resulting in an (instant, value) pair, or to a given set of time intervals, respectively. The **atinstant** operation is similar to the timeslice operator found in many temporal relational algebras. Operations **initial** and **final** return the first and last (instant, value) pair, respectively. Operation **present** allows one to check whether the moving value exists at a given instant, or is ever present during a given set of time intervals.

In the second group, the purpose of **at** is again restriction (like **atinstant**, **atperiods**), here to values in the range. For 1D space, restriction to a point or a point-set value returns a value of the given moving type. For example, we can restrict a moving real to the times when its value was between 3 and 4. In 2D, the resulting moving type is obtained by taking the minimum of the two argument types $\alpha$ and $\beta$ with respect to the order $\underline{point} < \underline{points} < \underline{line} < \underline{region}$. For example, the restriction of a $\underline{moving}(\underline{region})$ by a $\underline{point}$ will result in a $\underline{moving}(\underline{point})$. This is analogous to the definitions **intersection** in 2D in Section 4.3.4.

In one-dimensional spaces, operations **atmin** and **atmax** restrict the moving value to the times when it was minimal or maximal with respect to the total order on this space. Operation **passes** determine whether the moving value ever assumed (one of) the value(s) in the second argument.

All of these operations are of interest from a language design point of view. Some of them may also be expressed in terms of other operations in the framework. For example, we have $\mathbf{present}(f, t) = \mathbf{not}(\mathbf{isempty}(\mathbf{val}(\mathbf{atinstant}(f, t))))$.

*Example 9.* "When and where did flight 257 enter French air space?"

```
LET entry = initial(at(route257, France));
inst(entry); val(entry)
```

*Example 10.* "When was the Eiffel Tower within snow storm 'Lizzy'?"

```
LET Lizzy = ELEMENT(SELECT area FROM weather
  WHERE name = "Lizzy" and kind = "snow storm");
deftime(at(Lizzy, EiffelTower))
```

**Lifting Operations to Time-Dependent Operations.** Section 4.3.4 systematically defines operations on non-temporal types. This section uniformly lifts these operations to apply to the corresponding *moving* (temporal) types. The idea is to allow any argument of a non-temporal operation to be made temporal and to return a temporal type. More specifically, the lifted version of an operation with signature $\alpha_1 \times \ldots \times \alpha_k \to \beta$ has signatures $\alpha'_1 \times \ldots \times \alpha'_k \to \underline{moving}(\beta)$ with $\alpha'_i \in \{\alpha_i, \underline{moving}(\alpha_i)\}$.

So, each argument type may be changed into a time-dependent type, which will then transform the result type into a time-dependent type. The new operations are given the same name as the operation they originate from. As an example of lifting, the **intersection** operation with signature $\underline{region} \times \underline{point} \to \underline{point}$ is lifted to the signatures $\underline{mregion} \times \underline{point} \to \underline{mpoint}$, $\underline{region} \times \underline{mpoint} \to \underline{mpoint}$, and $\underline{mregion} \times \underline{mpoint} \to \underline{mpoint}$.

To define the semantics of lifting, we note that an operation $op : \alpha_1 \times \ldots \times \alpha_k \to \beta$ can be lifted with respect to any combination of argument types. The set of lifted parameters may be described by a set $L \subseteq \{1, \ldots, k\}$, and we define:

$$\alpha_i^L = \begin{cases} \underline{moving}(\alpha_i) & \text{if } i \in L \\ \alpha_i & \text{otherwise} \end{cases}$$

Thus, the signature of any lifted version of $op$ can be written as $op : \alpha_1^L \times \ldots \times \alpha_k^L \to \underline{moving}(\beta)$. If $f_{op}$ is the semantics of $op$, we now have to define the semantics of $f_{op}^L$ for each possible lifting $L$. For this we define what it means to apply a possibly lifted value to an *instant*-value:

$$x_i^L(t) = \begin{cases} x_i(t) & \text{if } i \in L \\ x_i & \text{otherwise} \end{cases}$$

This enables a point-wise definition of the functions $f_{op}^L$.

$$f_{op}^L(x_1, \ldots, x_k) = \{(t, f_{op}(x_1^L(t), \ldots, x_k^L(t))) \mid t \in A_{instant}\}$$

This lifting generalizes existing operations, which perhaps did not appear to be of great utility, to new and quite useful operations. For example, an operator that determines the intersection of a region with a point may not be of great interest, but the operation that determines the intersection between a *region* and an *mpoint* ("get the part of the *mpoint* within the region") is quite useful. This explains why Section 4.3.4 defined the set operations for all argument types, including single points.

Time-dependent operations enable a powerful query language. Examples follow.

*Example 11.* We can formulate involved queries such as "For how long did the moving point `mp` move along the boundary of region `r`?"

```
duration(deftime(at(on_border(mp, r), TRUE)))
```

Predicate **on_border** yields a result of type *mbool*, which is defined for all times when `mp` is defined and has value *TRUE* or *FALSE*. Operation **at** restricts this *mbool* to the times when it has value *TRUE*.

*Example 12.* "When did snow storm 'Lizzy' consist of exactly three separate areas."

```
deftime(at(no_components(Lizzy) = 3, TRUE))
```

Here, 'Lizzy' is of type *mregion*, and the lifted versions of **no_components** and equality apply.

**Rate of Change.** An important property of any time-dependent value is its rate of change, i.e., its **derivative**. This concept is applicable to types *mreal* and *mpoint*. For the latter, we include three operators, namely **speed**, based on Euclidean distance, **turn**, based on the direction between two points, and **velocity**, based on the vector difference (viewing points as 2D vectors). The acceleration

**Table 4.10.** Derivative operations

| Operation | Signature | Semantics |
|---|---|---|
| **derivative** | *mreal* $\rightarrow$ *mreal* | $\mu'$ where $\mu'(t) = \lim_{\delta \to 0}(f(t+\delta) - f(t))/\delta$ |
| **speed** | *mpoint* $\rightarrow$ *mreal* | $\mu'$ where $\mu'(t) = \lim_{\delta \to 0} f_{\mathbf{distance}}(f(t+\delta), f(t))/\delta$ |
| **turn** | *mpoint* $\rightarrow$ *mreal* | $\mu'$ where $\mu'(t) = \lim_{\delta \to 0} f_{\mathbf{direction}}(f(t+\delta), f(t))/\delta$ |
| **velocity** | *mpoint* $\rightarrow$ *mpoint* | $\mu'$ where $\mu'(t) = \lim_{\delta \to 0}(\overrightarrow{f(t+\delta)} - \overrightarrow{f(t)})/\delta$ |

of a moving point `mp` may be obtained as a number by **derivative**(**speed**(`mp`)) and as a vector, or moving point, by **velocity**(**velocity**(`mp`)).

*Example 13.* One can observe the growth rate of a moving region: "When did 'Lizzy' expand the most?"

```
inst(initial(atmax(derivative(area(Lizzy)))))
```

*Example 14.* "Show on a map the parts of the route of flight 257 when the plane's speed exceeds 800 km/h."

```
trajectory(atperiods(route257,
  deftime(at(speed(route257) > 800, TRUE))))
```

The background of the map has to be produced by a different tool or query.

### 4.3.6 Application Example

To illustrate the use of the data types in querying, we consider an example application that concerns forest fire analysis and which allows us to explore advanced aspects of moving point and region objects.

In a number of countries, fire is one of the main agents of forest damage. Forest fire control management mainly pursues the two goals of learning from past fires and their evolution and of preventing fires in the future, by studying weather and other factors such as cover type, elevation, slope, distance to roads, and distance to human settlements. In a very simplified manner, this example considers the first goal of learning from past fires and their evolution in space and time. We assume a database containing relations with schemas

```
forest(forestname:string, territory:mregion)
forest_fire(firename:string, extent:mregion)
fire_fighter(fighternname:string, location:mpoint)
```

Relation `forest` records the extents of forests, which grow and shrink over time due to, e.g., clearing, cultivation, and destruction processes. Relation `forest_fire` captures the evolution of fires, from ignition to extinction. Relation `fire_fighter` describes the motions of fire fighters on duty, from their start at the fire station up to their return. The following four queries illustrate enhanced spatio-temporal database functionality.

*Example 15.* "When and where did the fire called 'The Big Fire' have its largest extent?"

```
LET TheBigFire = ELEMENT(SELECT extent FROM forest_fire
  WHERE firename = "The Big Fire");
LET max_area = initial(atmax(area(TheBigFire)));
atinstant(TheBigFire, inst(max_area));
val(max_area)
```

The second argument of **atinstant** computes the time when the area of the fire was at its maximum. The area operator is used in its lifted version.

*Example 16.* "Determine the size of all forest areas destroyed by 'The Big Fire'." We assume that a fire may reach several, perhaps adjacent, forests.

```
LET ever = FUN (mb:mbool) passes(mb, TRUE);
LET burnt =
  SELECT size AS area(traversed(intersection(territory, extent)))
  FROM forest_fire, forest
  WHERE firename = "The Big Fire"
    and ever(intersects(territory, extent));
SELECT SUM(size)
FROM burnt
```

The **intersects** predicate of the join condition is lifted. Since the join condition expects a Boolean value, the **ever** predicate checks whether there is at least one intersection between the two *mregion* values just considered.

*Example 17.* "When and where was the spread of fires larger than 500 $km^2$?"

```
LET big_part =
  SELECT big_area AS extent when[FUN (r:region) area(r) > 500]
  FROM forest_fire;
SELECT *
FROM big_part
WHERE not(isempty(deftime(big_area)))
```

The first subquery reduces the moving region of each fire to the parts when it was large. For some fires, this may never be the case; for them, `bigarea` may be empty (always undefined). These are eliminated in the second subquery.

*Example 18.* " How long was fire fighter Th. Miller enclosed by 'The Big Fire' and which distance did the fire fighter cover there?

```
SELECT time AS
    duration(deftime(intersection(location, TheBigFire))),
  distance AS
    length(trajectory(intersection(location, TheBigFire)))
FROM fire_fighter
WHERE fightername = "Th. Miller"
```

We assume that the value 'TheBigFire' has already been determined as in Example 15, and that we know that Th. Miller was in this fire (otherwise, time and distance will be returned as zero).

### 4.3.7 Summary

This section offers an integrated, comprehensive design of abstract data types involving base types, spatial types, time types, as well as consistent temporal and spatio-temporal versions of these. Embedding this in a DBMS query language, one obtains a query language for spatio-temporal data, and moving objects in particular.

The strong points are several. The framework emphasizes genericity, closure, and consistency. An abstract level of modeling is adopted, with the design including the first comprehensive model of spatial data types (going beyond the study of just topological relationships) formulated entirely at the abstract, infinite point-set level. To our knowledge, the framework is the first to systematically and coherently use continuous functions as values of attribute data types. Finally, the idea of defining operations over non-temporal types and then uniformly lift these to operations over temporal types seems to be a new and important concept that achieves consistency between non-temporal and temporal operations.

## 4.4 A Discrete Model: Data Structures for Moving Objects Databases

### 4.4.1 Overview

In this section, which is based on [16], we define data types that can represent values of corresponding types of the abstract model just presented in Section 4.3.

Of course, the discrete types can in general only represent a subset of the values of the corresponding abstract type.

All type constructors of the abstract model will have direct counterparts in the discrete model except for the *moving* constructor. This is, because it is impossible to introduce at the discrete level a type constructor that automatically transforms types into corresponding temporal types. The type system for the discrete model therefore looks quite the same as the abstract type system shown in Table 4.2 up to the *intime* constructor, but then introduces a number of new type constructors to implement the *moving* constructor, as shown in Table 4.11.

**Table 4.11.** Signature describing the discrete type system

| | | |
|---|---|---|
| | → BASE | *int*, *real*, *string*, *bool* |
| | → SPATIAL | *point*, *points*, *line*, *region* |
| | → TIME | *instant* |
| BASE ∪ TIME | → RANGE | *range* |
| BASE ∪ SPATIAL | → TEMPORAL | *intime* |
| BASE ∪ SPATIAL | → UNIT | *const* |
| | → UNIT | *ureal*, *upoint*, *upoints*, *uline*, *uregion* |
| UNIT | → MAPPING | *mapping* |

Let us give a brief overview of the meaning of the discrete type constructors. The base types *int*, *real*, *string*, *bool* can be implemented directly in terms of corresponding programming language types. The spatial types *point* and *points* also have direct discrete representations whereas for the types *line* and *region* linear approximations (i.e., polylines and polygons) are introduced. Type *instant* is also represented directly in terms of programming language real numbers. The *range* and *intime* types represent sets of intervals, or pairs of time instants and values, respectively. These representations are also straightforward.

The interesting part of the model is how temporal ("moving") types are represented. We here describe the so-called *sliced representation*. The basic idea is to decompose the temporal development of a value into fragments called "slices" such that within the slice this development can be described by some kind of "simple" function. This is illustrated in Figure 4.5.

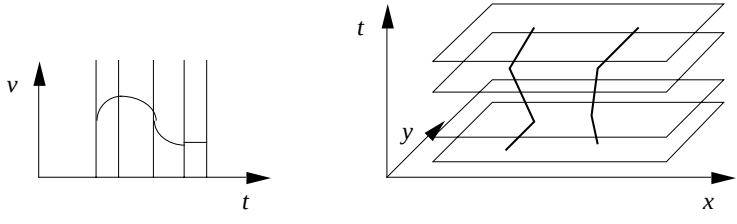



**Fig. 4.5.** Sliced representation of moving *real* and moving *points* value

The sliced representation is built by a type constructor *mapping* parameterized by the type describing a single slice which we call a *unit* type. A value of a unit type is a pair $(i, v)$ where $i$ is a time interval and $v$ is some representation of a simple function defined within that time interval. We define unit types *ureal*, *upoint*, *upoints*, *uline*, and *uregion*. For values that can only change discretely, there is a trivial "simple" function, namely the constant function. It is provided by a *const* type constructor which produces units whose second component is just a constant of the argument type. This is in particular needed to represent moving *int*, *string*, and *bool* values. The *mapping* data structure basically just assembles a set of units and makes sure that their time intervals are disjoint.

In summary, we obtain the correspondence between abstract and discrete temporal types shown in Table 4.12.

**Table 4.12.** Correspondence between abstract and discrete temporal types

| Abstract Type | Discrete Type |
|---|---|
| *moving*(*int*) | *mapping*(*const*(*int*)) |
| *moving*(*string*) | *mapping*(*const*(*string*)) |
| *moving*(*bool*) | *mapping*(*const*(*bool*)) |
| *moving*(*real*) | *mapping*(*ureal*) |
| *moving*(*point*) | *mapping*(*upoint*) |
| *moving*(*points*) | *mapping*(*upoints*) |
| *moving*(*line*) | *mapping*(*uline*) |
| *moving*(*region*) | *mapping*(*uregion*) |

In Table 4.12 we have omitted the representations *mapping*(*const*(*real*)), etc. which can be used to represent discretely changing real values and so forth, but are not so interesting for us.

In the remainder of this section we formally define the data types of the discrete model. That means, for each type we define its *domain* of values in terms of some finite representation. From an algebraic point of view, we define for each *sort* (type) a *carrier set*. For a type $\alpha$ we denote its carrier set as $D_\alpha$.

Of course, each value in $D_\alpha$ is supposed to represent some value of the corresponding abstract domain, that is, the carrier set of the corresponding abstract type. For a type $\alpha$ of the abstract model, let $A_\alpha$ denote its carrier set. We can view the value $a \in A_\alpha$ that is represented by $d \in D_\alpha$ as the *semantics* of $d$. We will always make clear which value from $A_\alpha$ is meant by a value from $D_\alpha$. Often this is obvious, or an informal description is sufficient. Otherwise we provide a definition of the form $\sigma(d) = a$ where $\sigma$ denotes the "semantics" function.

The following Section 4.4.2 contains definitions for all non-temporal types and for the temporal types in the sliced representation. For the spatial temporal data types *moving*(*points*), *moving*(*line*), and *moving*(*region*) one can also define direct three-dimensional representations in terms of polyhedra etc.; these representations will be treated elsewhere.

In Chapter 6 of this book we will present some examples of how this high level specification translates into physical data structures and algorithms.

### 4.4.2 Definition of Discrete Data Types

**Base Types and Time Type.** The carrier sets of the *discrete base types* and the type for time rest on available programming language types. Let $Instant = \texttt{real}$.

$$D_{\underline{int}} = \texttt{int} \cup \{\bot\} \qquad D_{\underline{real}} = \texttt{real} \cup \{\bot\} \qquad D_{\underline{string}} = \texttt{string} \cup \{\bot\}$$
$$D_{\underline{bool}} = \texttt{bool} \cup \{\bot\} \quad D_{\underline{instant}} = Instant \cup \{\bot\}$$

The only special thing about these types is that they always include the undefined value $\bot$ as required by the abstract model. Since we are interested in continuous evolutions of values, type *instant* is defined in terms of the programming language type `real`.

We sometimes need to speak about only the defined values of some carrier set and therefore introduce a notation for it: Let $D'_\alpha = D_\alpha \setminus \{\bot\}$. We will later introduce carrier sets whose elements are sets themselves; for them we extend this notation to mean $D'_\alpha = D_\alpha \setminus \{\emptyset\}$.

**Spatial Data Types.** Next, we define finite representations for single points, point collections, lines, and regions in two-dimensional (2D) Euclidean space. A point is, as usual, given by a pair $(x, y)$ of coordinates. Let $Point = \texttt{real} \times \texttt{real}$ and

$$D_{\underline{point}} = Point \cup \{\bot\}$$

The semantics of an element of $D_{\underline{point}}$ is obviously an element of $A_{\underline{point}}$. We assume lexicographical order on points, that is, given any two points $p, q \in Point$, we define: $p < q \Leftrightarrow (p.x < q.x) \vee (p.x = q.x \wedge p.y < q.y)$.

A value of type *points* is simply a set of points.

$$D_{\underline{points}} = 2^{Point}$$

Again it is clear that a value of $D_{\underline{points}}$ represents a value of the abstract domain $A_{\underline{points}}$.

The definition of discrete representations for the types *line* and *region* is based on linear approximations. A value of type *line* is essentially just a finite set of line segments in the plane. Figure 4.6 shows the correspondence between the abstract type for *line* and the discrete type. The abstract type is a set of curves in the plane which was viewed in Section 4.3 as a planar graph whose nodes are intersections of curves and whose edges are intersection-free pieces of curves. The discrete *line* type represents curves by polylines. However, one can assume a less structured view and consider the same shape to be just a collection of line segments. At the same time, any collection of line segments in the plane defines a valid collection of curves (or planar graph) of the abstract model (see Figure 4.6 (c)). Hence, modeling *line* as a set of line segments is no less expressive than the polyline view. It has the advantage that computing the projection of a (discrete representation) moving point into the plane can be done

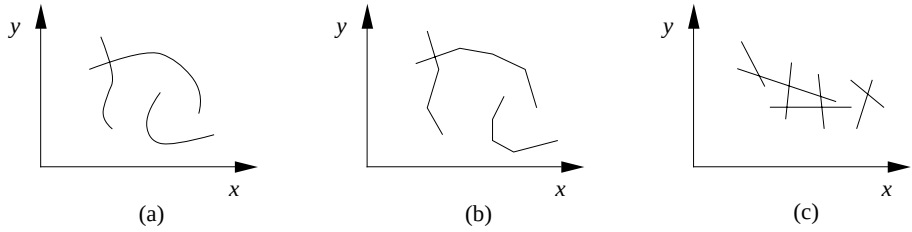



**Fig. 4.6.** (a) *line* value of the abstract model (b) *line* value of the discrete model (c) any set of line segments is also a *line* value

very efficiently as it is not necessary to compute the polyline or graph structure. Hence we prefer to use this unstructured view. Let

$$Seg = \{(u, v) \mid u, v \in Point, u < v\}$$

be the set of all line segments.

$$D_{line} = \{S \subset Seg \mid \forall s, t \in Seg : s \neq t \wedge collinear(s, t) \Rightarrow disjoint(s, t)\}$$

The predicate *collinear* means that two line segments lie on the same infinite line in 2D space. Hence for a set of line segments to be a *line* value we only require that there are no collinear, overlapping segments. This condition ensures unique representation, as collinear overlapping segments could be merged into a single segment. The semantics of a *line* value is, of course, the union of the points on all of its segments.

A *region* value at the discrete level is essentially a collection of polygons with polygonal holes (Figure 4.7). Formal definitions are based on the notions

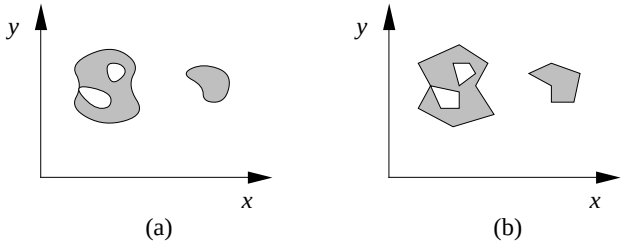



**Fig. 4.7.** (a) *region* value of the abstract model (b) *region* value of the discrete model

of *cycles* and *faces*. These definitions are similar to those of the ROSE algebra [22]. We need to reconsider such definitions here for two reasons: (i) They have to be modified a bit because here we have no "realm-based" [22] environment any more, and (ii) we are going to extend them to the "moving" case in the following sections.

A *cycle* is a simple polygon, defined as follows:

$$
\begin{aligned}
&\mathit{Cycle} = \{S \subset \mathit{Seg} \mid |S| = n, n \geq 3, \text{ such that} \\
&\quad (i)\ \ \forall s,t \in S : s \neq t \Rightarrow \neg \mathit{p\text{-}intersect}(s,t) \wedge \neg \mathit{touch}(s,t) \\
&\quad (ii)\ \ \forall p \in \mathit{points}(S) : \mathit{card}(p,S) = 2 \\
&\quad (iii)\ \exists \langle s_0, \ldots, s_{n-1} \rangle : \{s_0, \ldots, s_{n-1}\} = S \\
&\qquad\qquad \wedge (\forall i \in \{0, \ldots, n-1\} : \mathit{meet}(s_i, s_{(i+1) \mod n}))\}
\end{aligned}
$$

Two segments *p-intersect* ("properly intersect") if they intersect in their interior (a point other than an end point); they *touch* if one end point lies in the interior of the other segment. Two segments *meet* if they have a common end point. The set $\mathit{points}(S)$ contains all end points of segments, hence is $\mathit{points}(S) = \{p \in \mathit{Point} \mid \exists s \in S : s = (p,q) \vee s = (q,p)\}$. The function $\mathit{card}(p,S)$ tells how often point $p$ occurs in $S$ and is defined as $\mathit{card}(p,S) = |\{s \in S \mid s = (p,q) \vee s = (q,p)\}|$. Hence a collection of segments is a cycle, if (i) no segments intersect properly, (ii) each end point occurs in exactly two segments, and (iii) segments can be arranged into a single cycle rather than several disjoint ones (the notation $\langle s_0, \ldots, s_{n-1} \rangle$ refers to an ordered list of segments).

A *face* is a pair consisting of an outer cycle and a possibly empty set of hole cycles.

$$
\begin{aligned}
&\mathit{Face} = \{(c,H) \mid c \in \mathit{Cycle}, H \subset \mathit{Cycle}, \text{ such that} \\
&\quad (i)\ \ \forall h \in H : \mathit{edge\text{-}inside}(h,c) \\
&\quad (ii)\ \ \forall h_1, h_2 \in H : h_1 \neq h_2 \Rightarrow \mathit{edge\text{-}disjoint}(h_1, h_2) \\
&\quad (iii)\ \text{any cycle that can be formed from the segments of } c \text{ or } H \text{ is either} \\
&\qquad\quad c \text{ or one of the cycles of } H
\end{aligned}
$$

A cycle $c$ is *edge-inside* another cycle $d$ if its interior is a subset of the interior of $d$ and no edges of $c$ and $d$ overlap. They are *edge-disjoint* if their interiors are disjoint and none of their edges overlap. Note that it is allowed that a segment of one cycle *touches* a segment of another cycle. Overlapping segments are not allowed, since then one could remove the overlapping parts entirely (e.g. two hole cycles could be merged into one hole). The last condition (iii) ensures unique representation, that is, there are no two different interpretations of a set of segments as sets of faces. This implies that a face cannot be decomposed into two or more edge-disjoint faces.

A *region* is then basically a set of disjoint faces.

$$D_{\underline{region}} = \{F \subset \mathit{Face} \mid f_1, f_2 \in F \wedge f_1 \neq f_2 \Rightarrow \mathit{edge\text{-}disjoint}(f_1, f_2)\}$$

More precisely, faces have to be *edge-disjoint*. Two faces $(c_1, H_1)$ and $(c_2, H_2)$ are *edge-disjoint* if either their outer cycles $c_1$ and $c_2$ are edge-disjoint, or one of the outer cycles, e.g. $c_1$, is *edge-inside* one of the holes of the other face (some $h \in H_2$). Hence faces may also touch each other in an isolated point, but must not have overlapping boundary segments.

The semantics of a region value should be clear: A cycle $c$ represents all points of the plane enclosed by it as well as the points on the boundary. Given $\sigma(c)$,

we have for a face $\sigma((c, H)) = closure(\sigma(c) \setminus \bigcup_{h \in H} \sigma(h))$, that is, hole areas are subtracted from the outer cycle area, but then the resulting point set is closed again in the abstract domain. The area of a region is then obviously the union of the area of its faces.

**Sets of Intervals.** In this subsection, we introduce the *non-constant* $\underline{range}$ type constructor which converts a given type $\alpha \in \text{BASE} \cup \text{TIME}$ into a type whose values are finite sets of intervals over $\alpha$. Note that on all such types $\alpha$ a total order exists. Range types are needed, for example, to represent collections of time intervals, or the values taken by a moving real.

Let $(S, <)$ be a set with a total order. The representation of an interval over $S$ is given by the following definition.

$$\begin{aligned} Interval(S) = \{(s, e, lc, rc) | s, e \in S, lc, rc \in \texttt{bool}, \\ s \leq e, (s = e) \Rightarrow (lc = rc = true)\}. \end{aligned}$$

Hence an interval is represented by its end points $s$ and $e$ and two flags $lc$ and $rc$ indicating whether it is left-closed and/or right-closed. The meaning of an interval representation $(s, e, lc, rc)$ is

$$\sigma((s, e, lc, rc)) = \{u \in S | s < u < e\} \cup LC \cup RC$$

where the two sets $LC$ and $RC$ are defined as

$$LC = \begin{cases} \{s\} & \text{if } lc \\ \emptyset & \text{otherwise} \end{cases} \quad \text{and} \quad RC = \begin{cases} \{e\} & \text{if } rc \\ \emptyset & \text{otherwise} \end{cases}$$

Given an interval $i$, we denote with $\sigma'(i)$ the semantics expressed by $\sigma(i)$ restricted to the open part of the interval.

Whether intervals $u = (s_u, e_u, lc_u, rc_u)$ and $v = (s_v, e_v, lc_v, rc_v) \in Interval(S)$ are *disjoint* or *adjacent* is defined as follows:

$$\begin{aligned} \textit{r-disjoint}(u, v) &\Leftrightarrow e_u < s_v \vee (e_u = s_v \wedge \neg(rc_u \wedge lc_v)) \\ \textit{disjoint}(u, v) &\Leftrightarrow \textit{r-disjoint}(u, v) \vee \textit{r-disjoint}(v, u) \\ \textit{r-adjacent}(u, v) &\Leftrightarrow \textit{disjoint}(u, v) \wedge (e_u = s_v \wedge (rc_u \vee lc_v)) \vee \\ &\qquad ((e_u < s_v \wedge rc_u \wedge lc_v) \wedge \neg(\exists w \in S \,|\, e_u < w < s_v)) \\ \textit{adjacent}(u, v) &\Leftrightarrow \textit{r-adjacent}(u, v) \vee \textit{r-adjacent}(v,\ u) \end{aligned}$$

The last condition for *r-adjacent* is important for discrete domains such as $\underline{int}$. Representations of finite sets of intervals over $S$ can now be defined as

$$\begin{aligned} IntervalSet(S) = \{V \subseteq Interval(S) \,| \\ (u, v \in S \wedge u \neq v) \Rightarrow \textit{disjoint}(u, v) \wedge \neg \textit{adjacent}(u, v)\} \end{aligned}$$

The conditions ensure that a set of intervals has a unique and minimal representation. The $\underline{range}$ type constructor can then be defined as:

$$D_{\underline{range}(\alpha)} = IntervalSet(D'_\alpha) \quad \forall \alpha \in \text{BASE} \cup \text{TIME}$$

We also define the *intime* type constructor in this subsection which yields types whose values consist of a time instant and a value, as in the abstract model.

$$D_{intime(\alpha)} = D_{instant} \times D_\alpha \quad \forall \alpha \in \text{BASE} \cup \text{SPATIAL}$$

**Sliced Representation for Moving Objects.** In this subsection we introduce and formalize the *sliced representation* for moving objects. The sliced representation is provided by the *mapping* type constructor which represents a moving object as a set of so-called *temporal units* (*slices*). Informally speaking, a temporal unit for a moving data type $\alpha$ is a maximal interval of time where values taken by an instance of $\alpha$ can be described by a "simple" function. A temporal unit therefore records the evolution of a value $v$ of some type $\alpha$ in a given time interval $i$, while ensuring the maintenance of type-specific constraints during such an evolution.

For a set of temporal units representing a moving object their time intervals are mutually disjoint, and if they are adjacent, their values are distinct. These requirements ensure unique and minimal representations.

Temporal units are described as a generic concept in this subsection. Their specialization to various data types is given in the next two subsections. Let $S$ be a set. The concept of temporal unit is defined by:

$$Unit(S) = Interval(Instant) \times S$$

A pair $(i, v)$ of $Unit(S)$ is called a *temporal unit* or simply a *unit*. Its first component is called the *unit interval*, its second component the *unit function*.

The *mapping* type constructor allows one to build sets of units with the required constraints. Let

$$\begin{aligned} &Mapping(S) = \{U \subseteq Unit(S) \mid \forall (i_1, v_1) \in U, \forall (i_2, v_2) \in U : \\ &(i)\ \ i_1 = i_2 \Rightarrow v_1 = v_2 \\ &(ii)\ i_1 \neq i_2 \Rightarrow (disjoint(i_1, i_2) \wedge (adjacent(i_1, i_2) \Rightarrow v_1 \neq v_2))\} \end{aligned}$$

The *mapping* type constructor is defined for any type $\alpha \in UNIT$ as:

$$D_{mapping(\alpha)} = Mapping(D_\alpha) \quad \forall \alpha \in UNIT.$$

In the next subsections we will define the types *ureal*, *upoint*, *upoints*, *uline*, and *uregion*. Since all of them will have the structure of a unit, the just introduced type constructor *mapping*($\alpha$) can be applied to all of them.

Units describe certain simple functions of time. We will define a generic function $\iota$ on units which evaluates the unit function at a given time instant. More precisely, let $\alpha$ be a non-temporal type (e.g. *real*) and $u_\alpha$ the corresponding unit type (e.g. *ureal*) with $D_{u_\alpha} = Interval(Instant) \times S_\alpha$, where $S_\alpha$ is a suitably defined set. Then $\iota_\alpha$ is a function

$$\iota_\alpha : S_\alpha \times Instant \to D_\alpha$$

Usually we will omit the index $\alpha$ and just denote the function by $\iota$. Hence, $\iota$ maps a discrete representation of a unit function for a given instant of time into a discrete representation of the function value at that time. The $\iota$ function serves three purposes: (i) It allows us to express constraints on the structure of a unit in terms of constraints on the structure of the corresponding non-temporal value. (ii) It allows us to express the semantics of a unit by reusing the semantics definition of the corresponding non-temporal value. (iii) It can serve as a basis for the implementation of the **atinstant** operation on the unit.

The use of $\iota$ will become clear in the next subsections when we instantiate it for the different unit types.

**Temporal Units for Base Types.** For a type $\alpha \in \text{BASE} \cup \text{SPATIAL}$, we introduce the type constructor *const* that produces a temporal unit for $\alpha$. Its carrier set is defined as:

$$D_{\underline{const}(\alpha)} = Interval(Instant) \times D'_\alpha$$

Recall that the notation $D'_\alpha$ refers to the carrier set of $\alpha$ without undefined elements or empty sets. A unit containing an undefined or empty value makes no sense as for such time intervals we can simply let no unit exist (within a *mapping*).

Note that, even if we introduce the type constructor *const* with the explicit purpose of defining temporal units for *int*, *string*, and *bool*, it can nevertheless be applied also to other types. This may be useful for applications where values of such types change only in discrete steps.

The trivial temporal function described by such a unit can be defined as

$$\iota(v, t) = v$$

Note that in defining $\iota$ for a specific unit type we automatically define the semantics of the unit which should be a temporal function in the abstract model. For example, for a value $u$ of a unit type *const*(*int*) the semantics $\sigma(u)$ should be a partial function $f : A'_{\underline{instant}} \to A'_{\underline{int}}$. This is covered by a generic definition of the semantics of unit types: Let $u = (i, v)$ be a value of a unit type $u_\alpha$. Then

$$\sigma(u) = f_u : A'_{\underline{instant}} \cap \sigma(i) \to A'_\alpha \quad \text{where}$$
$$f_u(t) = \sigma(\iota(v, t)) \quad \forall t \in \sigma(i)$$

Hence we reuse the semantics defined for the discrete value $\iota(v, t) \in D'_\alpha$.

This semantics definition will in most cases be sufficient. However, for some unit types (namely, *uline* and *uregion*) the discrete value obtained in the end points of the time interval by $\iota$ may be an incorrect one due to degeneracies: in such a case it has to be "cleaned up." We will below slightly extend the generic semantics definition to accommodate this. For all other units, this semantics definition suffices so that we will only define the $\iota$ function in each case.

For the representation of moving reals we introduce a unit type *ureal*. The "simple" function we use for the sliced representation of moving reals is either

a polynomial of degree not higher than two or a square root of such a polynomial. The motivation for this choice is a trade-off between richness of the representation (e.g. square roots of degree two polynomials are needed to express time-dependent distance functions in the Euclidean metric) and simplicity of the representation of the discrete type and of its operations. With this particular choice one can implement (i.e., the discrete model is closed under) the lifted versions of **size**, **perimeter**, and **distance** operations; one cannot implement the **derivative** operation of the abstract model. The carrier set for type *ureal* is

$$D_{\underline{ureal}} = \mathit{Interval}(\mathit{Instant}) \times \{(a, b, c, r) \,|\, a, b, c \in \texttt{real}, r \in \texttt{bool}\}$$

and evaluation at time $t$ is defined by:

$$\iota((a, b, c, r), t) = \begin{cases} at^2 + bt + c & \text{if } \neg r \\ \sqrt{at^2 + bt + c} & \text{if } r \end{cases}$$

**Temporal Units for Spatial Data Types.** In this subsection we specialize the concept of unit to moving instances of spatial data types.

Similar to moving reals, the temporal evolution of moving spatial objects is characterized by continuity and smoothness and can be approximated in various ways. Again we have to find the balance between richness and simplicity of representation. As indicated before, in this chapter we make the design decision to base our approximations of the temporal behavior of moving spatial objects on linear functions. Linear approximations ensure simple and efficient representations for the data types and a manageable complexity of the algorithms. Nevertheless, more complex functions like polynomials of a degree higher than one are conceivable as the basis of representation but are not considered in this paper.

Due to the concept of sliced representation, also for moving spatial objects we have to specify constraints in order to describe the permitted behavior of a value of such a type within a temporal unit. Since the end points of a time interval mark a change in the description of the data type, we require that constraints are satisfied only for the respective open interval. In the end points of the time interval a collapse of components of the moving object can happen. This is completely acceptable, since one of the reasons to introduce the sliced representation is exactly to have "simple" and "continuous" description of the moving value within each time interval and to limit "discontinuities" in the description to a finite set of instants.

*Moving Points and Point Sets.* The structurally simplest spatial object that can move is a single point. Hence, we start with the definition of the spatial unit type *upoint*. First we introduce a set *MPoint* which defines 3D lines that describe unlimited temporal evolution of 2D points.

$$\mathit{MPoint} = \{(x_0, x_1, y_0, y_1) \,|\, x_0, x_1, y_0, y_1 \in \texttt{real}\}$$

This describes a linearly moving point for which evaluation at time $t$ is given by:

$$\iota((x_0, x_1, y_0, y_1), t) = (x_0 + x_1 \cdot t, y_0 + y_1 \cdot t) \quad \forall t \in \mathit{Instant}$$

The carrier set of *upoint* can then be very simply defined as:

$$D_{\mathit{upoint}} = \mathit{Interval}(\mathit{Instant}) \times \mathit{MPoint}$$

We pass now to describe a set of moving points. The carrier set of *upoints* can be defined as:

$$D_{\mathit{upoints}} = \{(i, M) \mid i \in \mathit{Interval}(\mathit{Instant}), M \subset \mathit{MPoint}, |M| \geq 1, \text{ and}$$
$$(i)\ \forall t \in \sigma'(i), \forall l, k \in M : l \neq k \Rightarrow \iota(l, t) \neq \iota(k, t)$$
$$(ii)\ i = (s, e, lc, rc) \wedge s = e \Rightarrow (\forall l, k \in M : l \neq k \Rightarrow \iota(l, s) \neq \iota(k, s))\}$$

Here we encounter for the first time a constraint valid during the open time interval of the unit (condition (i)). Namely, a *upoints* unit is a collection of linearly moving points that do not intersect within the open unit interval. Condition (ii) concerns units defined only in a single time instant; for them all points have to be distinct at that instant.

For $(i, M) \in D_{\mathit{upoints}}$, evaluation at time $t$ is given by

$$\iota(M, t) = \bigcup_{m \in M} \{\iota(m)\} \quad \forall t \in \sigma(i)$$

which is clearly a set of points in $D'_{\mathit{points}}$. We will generally assume that $\iota$ distributes through sets and tuples so that $\iota(M, t)$ is defined for any set $M$ as above, and for a tuple $r = (r_1, \ldots, r_n)$, we have $\iota(r, t) = (\iota(r_1), \ldots, \iota(r_n))$.

*Moving Lines.* We now introduce the unit type for *line* called *uline*. Here we restrict movements of segments so that in the time interval associated to a value of *uline* each segment maintains its direction in the 2-dimensional space. That is, segments which rotate during their movement are not admitted. See in Figure 4.8 an example of a valid *uline* value. This constraint derives from the need of

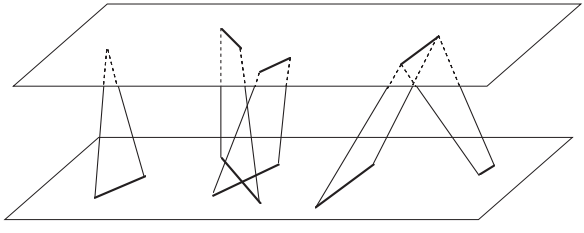


**Fig. 4.8.** An instance of *uline*

keeping a balance between ease of representation and manipulation of the data

type and its expressive power. Rotating segments define curved surfaces in the 3D space that, even if they constitute a more accurate description, can always be approximated by a sequence of plane surfaces.

The carrier set of *uline* is therefore based on a set of moving segments with the above restriction and which never overlaps at any instant internal to the associated open time interval. Overlapping has a meaning equivalent to the one used for *line* values: to be collinear and to have a non-empty intersection.

To prepare the definition of *uline* we introduce the set of all pairs of lines in a 3D space that are coplanar, which will be used to represent moving segments:

$$MSeg = \{(s, e) \mid s, e \in MPoint, s \neq e, s \text{ is coplanar with } e\}.$$

The carrier set for *uline* can now be defined as:

$$\begin{aligned} D_{uline} = \{(i, M) \mid i \in Interval(Instant), M \subset MSeg, |M| \geq 1, \text{ such that} \\ (i)\ \forall t \in \sigma'(i) : \iota(M, t) \in D'_{line} \\ (ii)\ i = (s, e, lc, rc) \wedge s = e \Rightarrow \iota(M, s) \in D'_{line}\} \end{aligned}$$

Here again the first condition defines constraints for the open time interval and the second treats the case of units defined only at a single instant. Note that $\iota(M, t)$ is defined due to the fact that $\iota$ distributes through sets and tuples. A *uline* value therefore inherits the structural conditions on *line* values and segments. For example, condition (i) requires that

$$(s, e) \in M \Rightarrow (\iota(s, t), \iota(e, t)) \in Seg \quad \forall t \in \sigma'(i)$$

and therefore $\iota(s, t) < \iota(e, t) \quad \forall t \in \sigma'(i)$.

The semantics defined for *uline* via $\iota$ according to the generic definition given earlier needs to be slightly changed to cope with degeneracies in the end points of a unit time interval, as we anticipated. In these points, in fact, moving segments can degenerate into points and different moving segments can overlap. We accommodate this by defining separate $\iota$ functions for the start time and the end time of the time interval, called $\iota_s$ and $\iota_e$, respectively. Let $((s, e, lc, rc), M) \in D_{uline}$. Then

$$\iota_s(M, t) = \iota_e(M, t) = \textit{merge-segs}(\{(p, q) \in \iota(M, t) \mid p < q\}$$

This definition removes pairs of points returned by $\iota(M, t)$ that are not segments (i.e., segments degenerated into a single point); it also merges overlapping segments into maximal ones (this is the meaning of the *merge-segs* function). The generic semantics definition is then extended as follows:

$$\sigma(u) = f_u : A'_{instant} \cap \sigma(i) \to A'_\alpha$$

where for $u = (i, v)$ and $i = (s, e, lc, rc)$

$$f_u(t) = \begin{cases} \sigma(\iota(v, t)) & \text{if } t \in \sigma'(i) \\ \sigma(\iota_s(v, t)) & \text{if } t = s \wedge lc \\ \sigma(\iota_e(v, t)) & \text{if } t = e \wedge rc \end{cases}$$

A final remark on the design decisions for the discrete type for moving lines is the following. Assume we choose instance $u_1$ (resp., $u_2$) of *uline* as the discrete representation at the initial (resp., final) time $t_1$ ($t_2$) of a unit for the (continuously) moving line $l$. Then the constraint that segments making up the discrete representation of $l$ cannot rotate during the unit does not restrict too much the fidelity of the discrete representation. Indeed, since members of *MSeg* in a unit can be triangles, this leaves the possibility of choosing among many possible mappings between endpoints of their segments in $t_1$ and those in $t_2$, as long as the non-rotation constraint is satisfied. In Figure 4.9 an example of a discrete representation of a continuously moving line by means of an instance of *uline* is shown. If this approach causes a too rough approximation internally to

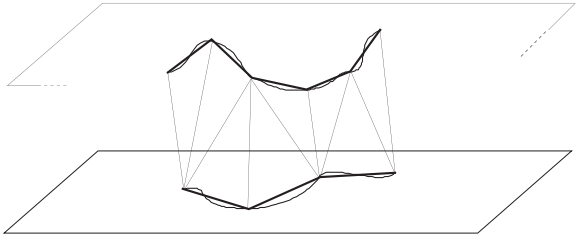


**Fig. 4.9.** A discrete representation of a moving line

the time unit, then possibly an additional instant, internal to the unit, has to be chosen and an additional discrete representation of $l$ at that instant has to be introduced so that a better approximation is obtained. It can be easily seen that in the limit this sequence of discrete representations can reach an arbitrary precision in representing $l$.

*Moving Regions.* We now introduce the moving counterpart for *region*, namely the *uregion* data type. We adopt the same restriction used for moving lines, i.e., that rotation of segments in the 3-dimensional space is not admitted. We therefore base the definition of *uregion* on the same set of all pairs of lines in a 3D space that are coplanar, namely *MSeg*, with additional constraints ensuring that throughout the whole unit we always obtain a valid instance of the *region* data type. Figure 4.10 shows an example of a valid *uregion* value. (It also shows the degeneracies that can occur in the end points of a unit interval.)

As for a *region* value, we can have moving regions with (moving) holes, hence the basic building blocks are given by the concepts of *cycle* and *face* already introduced in the definition of *region*.

The carrier set of *uregion* is therefore based, informally speaking, on a set of (possibly nested) faces which never intersect at any instant internal to the associated time interval. For the formal definition of *uregion*, we first introduce a set intended to describe the moving version of a cycle, without restriction on

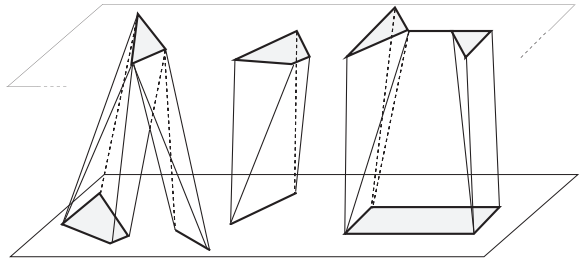


**Fig. 4.10.** An instance of *uregion*

time:

$$MCycle = \{\{s_0, \ldots, s_{n-1}\} \mid n \geq 3, \forall i \in \{0, \ldots, n-1\} : \ s_i \in MSeg\}$$

We then introduce a set for the description of the moving version of a face, without restriction on time:

$$MFace = \{(c, H) \mid c \in MCycle, H \subset MCycle\}.$$

Note that in the definitions of *MCycle* and *MFace* we have not given the constraints to impose on the sets the semantics of cycles and faces because this will be done directly in the moving region definition. The carrier set for *uregion* is now defined as

$$\begin{aligned} D_{uregion} = \{(i, F) \mid\ & i \in Interval(Instant), F \subset MFace, \text{ such that} \\ & (i)\ \ \forall t \in \sigma'(i) : \iota(F, t) \in D'_{region} \\ & (ii)\ i = (s, e, lc, rc) \wedge s = e \Rightarrow \iota(F, s) \in D'_{region}\} \end{aligned}$$

For the end points of the time interval again we have to provide separate functions $\iota_s$ and $\iota_e$. Essentially these work as follows. From the pairs of points $(p, q)$ (segments) obtained by evaluating $\iota(F, s)$ or $\iota(F, e)$, remove all pairs that are no proper segments (as for *uline*). Next, for all collections of overlapping segments on a single line, partition the line into fragments belonging to the same set of segments (e.g. if segment $(p, q)$ overlaps $(r, s)$ such that points are ordered on the line as $\langle p, r, q, s\rangle$ then there are fragments $(p, r), (r, q)$, and $(q, s)$). For each fragment, count the number of segments containing it. If this number is even, remove the fragment; if it is odd, put the fragment as a new segment into the result. A complete formalization of this is lengthy and omitted.

We have now concluded the formal definition of data types of the discrete model. In Chapter 6 we will show some examples of translation into physical data structures of the above specifications and we will provide some examples of algorithms implementing operations on discrete data types.

## 4.5 Outlook

This section presents two extensions of the approach presented so far in this chapter, and two other approaches to spatio-temporal modeling with a different focus of interest.

Section 4.5.1 addresses the problem of defining spatio-temporal predicates and their composition, in order to describe developments of relationships between (moving) objects. For example, one might want to ask in a query for moving region objects that were first disjoint, then overlapped, and finally were disjoint again. Section 4.5.2 considers time-varying partitions of the plane, for example the countries of Europe over the last centuries, and operations on such dynamic partitions.

Section 4.5.3 presents a data model based on "quanta" where space is rasterized. Consider a chess board. Atomic spatial entities or "quanta" would be all corners of fields (point quanta), horizontal or vertical edges of fields (line quanta), and fields themselves (surface quanta). Spatial data types are defined as unions of such quanta and relational algebra is extended to allow grouping ("fold") or decomposition of spatial values. By adding time intervals, this model can also describe (discretely changing) spatio-temporal data.

The final subsection Section 4.5.4 addresses the treatment of legacy databases and their applications when a "dimension", which could be a time or a space dimension, is added. For example, a static table is extended by a valid time attribute. The goal is that applications that did not know about the time attribute can run unchanged, and will yield the same results as before.

### 4.5.1 Spatio-temporal Predicates and Developments

Spatio-temporal predicates characterize temporal changes of relationships between spatio-temporal objects. In the following we briefly discuss some of the design issues that arise with spatio-temporal predicates.

**What Are Spatio-temporal Predicates?** A basic spatio-temporal predicate can be thought of as a lifted spatial predicate yielding a temporal boolean, which is aggregated by determining whether that temporal boolean was sometimes or always true. In general, a spatio-temporal predicate can be viewed as a function that aggregates the values of a (possibly changing) spatial predicate as it evolves over time. Thus, a spatio-temporal predicate is a function of type $\tau(\alpha) \times \tau(\beta) \rightarrow \mathbb{B}$ for $\alpha, \beta \in \{point, region\}$.

Consider the definition of *inside* from Section 4.3. We can define two spatio-temporal predicates *sometimes-inside* and *always-inside* that yield true if *inside* yields true at some time, respectively, at all times.

Whereas the definition for *sometimes-inside* is certainly reasonable, the definition for *always-inside* is questionable since it yields false whenever the point or the region is undefined. This is not what we would expect. For example, when the moving point has a shorter lifetime than the evolving region but is always

inside the region, we would expect *always-inside* to yield true. In fact, we can distinguish different kinds of universal quantification that result from different time intervals over which aggregation can be defined to range. In the case of *inside* the expected behavior is obtained if the aggregation ranges over the lifetime of the first argument, the moving point. This is not true for all spatial predicates; actually, it depends on the nature and use of each individual predicate. For example, two spatio-temporal objects are considered as being *equal* only if they are equal on both objects' lifetimes, that is, the objects must have the same lifespans and must be always equal during these.

We denote different kinds of $\forall$-aggregation by parameterized quantifiers $\forall_\gamma$ where $\gamma \in \{\cup, \cap, \pi_1, \pi_2\}$ and where $\pi_i(x_1, \ldots, x_i, \ldots, x_n) = x_i$. These quantifiers are defined as follows.

$$\forall_\gamma p \ := \ \lambda(S_1, S_2).\forall t \in \gamma(dom(S_1), dom(S_2)) : p(S_1(t), S_2(t))$$

This means that, for example, $\forall_{\pi_1}.inside$ denotes the spatio-temporal predicate

$$\lambda(S_1, S_2).\forall\ t \in dom(S_1).inside(S_1(t), S_2(t))$$

In general, $\lambda(x_1, x_2, \ldots).e$ denotes a function that takes arguments $x_1, x_2, \ldots$ and returns a value determined by the expression $e$.

With this aggregation notation we can give the definitions for the spatio-temporal versions of the eight basic spatial predicates (for two regions).

$$\begin{array}{lll}
Disjoint & := & \forall_\cap.disjoint \\
Meet & := & \forall_\cup.meet \\
Overlap & := & \forall_\cup.overlap \\
Equal & := & \forall_\cup.equal \\
Covers & := & \forall_{\pi_2}.covers \\
CoveredBy & := & \forall_{\pi_1}.coveredBy \\
Contains & := & \forall_{\pi_2}.contains \\
Inside & := & \forall_{\pi_1}.inside
\end{array}$$

For a moving point and a moving region we have just the three basic predicates *Disjoint*, *Meet*, and *Inside*, which are defined as above.

The chosen aggregations (and possible variations) are motivated and discussed in great detail in [11].

**Developments: Sequences of Spatio-temporal Predicates.** Consider a plane entering a storm. This scenario is abstractly characterized by a moving point that initially is disjoint from an evolving region for some period of time, then touches its border, and finally remains inside of it. In other words, the described development is characterized by a sequence of spatio-temporal (and spatial) predicates: *Disjoint*, *meet*, and *Inside*. In order to define such predicate sequences we need a way of restricting the temporal scope of basic spatio-temporal predicates. This becomes possible by *predicate constrictions*: let $P$ be a spatio-temporal predicate, and let $I$ be a (half-) open or closed interval. Then

$$P_I \ := \ \lambda(S_1, S_2).P(S_1|_I, S_2|_I)$$

Here $S|_I$ denotes the partial function that yields $S(t)$ for all $t \in I$ and is undefined otherwise.

When we now consider more closely how spatial situations can change over time, we observe that certain relationships can be observed only for a period of time and not for only a single time point (given that the participating objects do exist for a period of time) while other relationships can hold at instants as well as on time intervals. Predicates that can hold at time points and intervals are: *equal*, *meet*, *covers*, *coveredBy*; these are called *instant predicates*. Predicates that, in general, can only hold on intervals are: *disjoint*, *overlap*, *inside*, *contains*; these are called *period predicates*.

It is interesting to note that (in satisfiable developments for continuously moving objects) instant and period predicates always occur in alternating patterns, for example, there cannot be two spatio-temporal objects that satisfy *Inside* immediately followed by *Disjoint*. In contrast, *Inside* first followed by *meet* (or *Meet*) and then followed by *Disjoint* can be satisfied.

Next we define three operations for combining spatio-temporal and spatial predicates: $p \vdash P$ (*from*) defines a spatio-temporal predicate that for some time $t$ checks $p$ and then enforces $P$ for all times after $t$; $P \dashv p$ (*until*) is defined dually, that is, $P$ must hold until $p$ is true at some time $t$. Finally, $P \dashv p \vdash Q$ (*then*) is true if there is some time point $t$ when $p$ is true so that $P$ holds before and $Q$ holds after $t$. Below we abbreviate open intervals $]t, \infty[$ and $]\infty, t[$ by simply writing $>t$ and $<t$. (Note that variable $t$ ranges over *time*.) Let $p$ be a spatial predicate, and let $P$ and $Q$ be spatio-temporal predicates. Then

$$
\begin{array}{lcl}
p \vdash P & := & \lambda(S_1, S_2).\exists t : p(S_1(t), S_2(t)) \wedge P_{>t}(S_1, S_2) \\
P \dashv p & := & \lambda(S_1, S_2).\exists t : p(S_1(t), S_2(t)) \wedge P_{<t}(S_1, S_2) \\
P \dashv p \vdash Q & := & \lambda(S_1, S_2).\exists t : p(S_1(t), S_2(t)) \wedge P_{<t}(S_1, S_2) \wedge Q_{>t}(S_1, S_2)
\end{array}
$$

These combinators obey several interesting laws; these and others are presented in [11]. In particular, the composition of predicate sequencing is associative, that is,

$$P \dashv p \vdash (Q \dashv q \vdash R) = (P \dashv p \vdash Q) \dashv q \vdash R$$

This enables us to use a succinct sequencing syntax for developments, that is, we can simply write $P \triangleright p \triangleright Q$ for $P \dashv p \vdash Q$. For example, we can define predicates for capturing the scenario of a point entering or crossing a region by:

$$
\begin{array}{lcl}
\mathit{Enter} & := & \mathit{Disjoint} \triangleright \mathit{meet} \triangleright \mathit{Inside} \\
\mathit{Cross} & := & \mathit{Disjoint} \triangleright \mathit{meet} \triangleright \mathit{Inside} \triangleright \mathit{meet} \triangleright \mathit{Disjoint}
\end{array}
$$

Sequential temporal composition is just one possibility to build new spatio-temporal predicates. Temporal alternative, negation, and reflection provide further powerful means to specify developments. These and many other combinators are defined and investigated in [11].

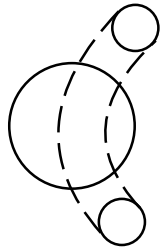


**Fig. 4.11.** Visual specification of the *Cross* predicate

**Further Work.** Spatio-temporal predicates lay the foundation for further research in spatio-temporal query languages. Two aspects have already been investigated:

First, it is, in fact, fairly simple to integrate spatio-temporal predicates into existing query languages. For example, we have shown in [10] how extending SQL by (i) the set of eight basic spatio-temporal predicates and (ii) by a macro facility to compose new predicates leads to a powerful spatio-temporal query language. Let us reconsider the example query of finding out all planes that ran into a storm. We assume having defined two relations `flights` and `weather` containing, respectively, a moving point attribute `Route` representing the flights' movements and an evolving region attribute `Extent` describing the developments of weather areas. With a predicate combinator `>>` that has the semantics of temporal composition $\triangleright$ we can formulate the query simply as:

```
SELECT id FROM flights, weather
WHERE kind = "storm"
  AND Route Disjoint>>meet>>Inside Extent
```

A second line of future work is motivated by the fact that the number of different spatio-temporal predicates is actually unlimited due to the sequencing possibility to generate new developments. Since we cannot invent names for all possible predicates we need some kind of language for specifying developments. Now an (additional) textual language for predicate specifications is not very convenient for the (end) user. Moreover, the specification of predicates can become quite longish. Hence, visual notations can be very useful to keep the specification of developments manageable by the user. Consider, for example, the predicate *Cross* which is defined for two evolving regions as follows:

$$\begin{aligned} \mathit{Cross} \;:=\; & \mathit{Disjoint} \triangleright \mathit{meet} \triangleright \mathit{Overlap} \triangleright \mathit{coveredBy} \triangleright \mathit{Inside} \triangleright \\ & \mathit{coveredBy} \triangleright \mathit{Overlap} \triangleright \mathit{meet} \triangleright \mathit{Disjoint} \end{aligned}$$

In contrast, this can be specified very easily and intuitively by a simple two-dimensional picture as shown in Figure 4.11.

The rationale behind this visual notation is described in more detail in [13,14]. The key idea is to infer from the intersections of two-dimensional traces of mov-

ing/evolving objects the temporal changes of their relationships. The visual notation is mainly intended to be used as a supplement to textual languages and can be integrated, for example, along the lines described in [8].

### 4.5.2 Spatio-temporal Partitions

While we have so far in this chapter dealt with the temporal evolution of *single* spatial entities, we now study the temporal evolution of *spatial partitions* as an important example of a *collection* of spatial entities satisfying specific constraints. This leads to a concept of *spatio-temporal partitions*.

**Spatial Partitions.** The metaphor of a *map* has turned out to be a fundamental and ubiquitous spatial concept in geography, cartography, and other related disciplines as well as in computer-assisted systems like GIS and spatial database systems. The central element of a map is a *spatial partition* which is a subdivision of the plane into pairwise disjoint *regions* where regions are separated from each other by *boundaries* and where each region is associated with an attribute or *label* having simple or even complex structure. That is, a region (possibly composed of several disconnected parts) with an attribute incorporates all points of a spatial partition having this attribute. Examples are the subdivision of the world map into countries, classification of rural areas according to their agricultural use, areas of different degrees of air pollution, etc. A spatial partition implicitly models topological relationships between the participating regions which can be regarded as integrity constraints. First, it expresses neighborhood relationships for different regions that have common boundaries. Second, different regions of a partition are always disjoint (except for common boundaries). Both topological properties are denoted as *partition constraints*. As a purely geometric structure, a map yields only a *static* description of spatial entities and required constraints between them.

A rigorous and thorough formal definition of spatial partitions and of application-specific operations defined on them has been given in [9]. The basic idea is that a spatial partition is a mapping from the Euclidean space $\mathbb{R}^2$ to some *label type*, that is, regions of a partition are assigned single labels. Adjacent regions have different labels in their interior, and a boundary is assigned the pair of labels of both adjacent regions.

Many application-specific operations on spatial partitions like *overlay*, *reclassify*, *fusion*, *cover*, *clipping*, *difference*, *superimposition*, *window*, and variations of them have been described in the literature (see for example, [1,23,29,33]). In [9] all these operations have been reduced to the three fundamental and powerful operations *intersection*, *relabel*, and *refine*. Intersecting two spatial partitions means to compute the geometric intersection of all regions and to produce a new spatial partition; each resulting region is labeled with the pair of labels of the original two intersecting regions, and the values on the boundaries are derived from these. Relabeling a spatial partition has the effect of changing the labels of its regions. This can happen by simply renaming the label of each region. Or, in

particular, distinct labels of two or more regions are mapped to the same new label. If some of these regions are adjacent in the partition, the border between them disappears, and the regions are fused in the result partition. Relabeling has then a coarsening effect. Refining a partition means to look with a finer granularity on regions and to reveal and to enumerate the connected components of regions.

**Spatio-temporal Partitions.** Spatio-temporal partitions [12] or "temporal maps" describe the temporal development of spatial partitions and have a wide range of interesting applications. They represent collections of evolving regions satisfying the partition constraints for each time of their lifespan and maintaining these constraints over time. That is, for each time of their lifespan we obtain a stationary, two-dimensional spatial partition which changes over time due to altering shapes, sizes, or attribute values of regions. This corresponds to our temporal object view which is based on the observation that everything that changes over time can be considered as a function over time. Spatio-temporal partitions can then be viewed as functions from time to a two-dimensional spatial partition.

Temporal changes of spatial partitions can occur either in discrete and stepwise constant steps or continuously and smoothly. Examples of the first category are the reunification of West and East Germany, the splitting of Yugoslavia, the temporal development of any hierarchical decomposition of space into administrative or cadastral units like the world map into countries or districts into land parcels, or the classification of rural areas according to their agricultural use over time. A characteristic feature of these applications is that the number of discrete temporal changes is finite and that there is no change between any two subsequent *temporal change points* which is a special form of continuity.

The open issue now is what happens at temporal change points with their abrupt transition from one spatial partition to another. If we consider the time point when West and East Germany were reunified, did the spatial partition before or after the reunification belong to this time point? Since we cannot come to an objective decision, we have to decide arbitrarily and to assign one of both spatial partitions to it. This, in particular, maintains the functional character of our temporal object view. We have chosen to ascribe the temporally later spatial partition to a temporal change point. Mathematically this means that we permit a finite set of time points where the temporal function is not continuous. The application examples reveal that after a temporal change point the continuity of the temporal function proceeds for some time interval up to the next temporal change point; there are no "thin, isolated slices" containing single spatial partitions at temporal change points. Hence, we have to tighten our requirement in the sense that mathematically the temporal function has to be *upper semicontinuous* at each time.

Examples of the second category are the temporal evolution of climatic phenomena like temperature zones or high/low pressure areas, areas of air pollution with distinct degrees of intensity, or developments of forest fires in space and

time. They all show a very dynamic and attribute-varying behavior over time. Application examples which have by far slower temporal changes are the increasing spread of ethnic or religious groups, the decreasing extent of mineral resources like oil fields during the course of time due to exploitation, or the subdivision of space into areas with different sets of spoken languages over time.

Application-specific spatio-temporal operations rest on the transfer of the two basic spatial partition operations *intersection* and *relabel* to the spatio-temporal case. They are temporally lifted and generalized versions of the application-specific operations on spatial partitions. The *overlay* operation is based on a spatio-temporal *intersection* operation and can be used to analyze the temporal evolution of two (or more) different attribute categories. Consider a temporal map indicating the extent of mineral resources like oil fields or coal deposits and another temporal map showing the country map over time. Then an overlay of both temporal maps can, for instance, reveal the countries that had or still have the richest mineral resources, it can show the grade of decline of mineral deposits in the different countries, and it can expose the countries which most exploited their mineral resources.

The *clipping* operation is a special case of the spatio-temporal *intersection* operation and works as a *spatio-temporal filter*. An application is a temporal map about the development of diseases. As a clipping window we use a temporal map of urban areas developing in space over time. The task is to analyze whether there is a connection between the increase or decrease of urban space and the development of certain diseases. Hence, all areas of disease outside of urban regions are excluded from consideration.

The *reclassification* operation is a special case of a spatio-temporal *relabel* operation. Consider a temporal map marking all countries of the world with their population numbers. A query can now ask for the proportion of each country's population on the world population over time, a task that can be performed by temporal relabeling. This corresponds to a reclassification of attribute categories over time without changing geometry.

The *fusion* operation is a kind of grouping operation with subsequent geometric union over time. Assume that a temporal map of districts with their land use is given. The task is to identify regions with the same land use over time. At each time neighboring districts with the same land use are replaced by a single region, that is, their common boundary line is erased. We obtain a temporal fusion operator which is based on relabeling. Reclassification and fusion are examples of *static relabeling* since the relabeling function does not change over time. We generalize this to *dynamic relabeling*. Consider the classification of income to show poor and rich areas over time. Due to the changing value of money, due to inflation, and due to social changes, the understanding of wealth and poorness varies over time. Hence, we need different and appropriate relabeling functions that are applied to distinct time intervals.

Additionally, some new operations are introduced that are more directed to the time dimension. The operation *dom* determines the domain of a temporal map, that is, all times where the map does not yield the completely undefined

partition. An example is a temporal map of earthquakes and volcanic eruptions in the world as they are interesting for seismological investigations. Applying the operation *dom* on this map returns the time periods of earthquake and volcanic activity in the world.

The operation *restrict* realizes a function restriction on spatio-temporal partitions and computes a new partition. As parameters it obtains a temporal map and a set of (right half-open) time intervals describing the time periods of interest. Imagine that we have a temporal map of birth rates, and we are only interested in the birth rates between 1989 and 1991 and between 1999 and 2001 ("millennium baby"). Then we can exclude all the other time periods and compare the change of birth rates in these two time intervals.

The operation *select* allows one to scan spatio-temporal partitions over time and to check for each time whether a specified predicate is fulfilled or not. Consider a map showing the spread of fires. We could be interested in when and where the spread of fires occupied an area larger than 300 $km^2$.

The operation *aggregate* collects all labels of a point over time and combines them with the aid of a binary function into one label. The result is a two-dimensional spatial partition. For example, if the population numbers, the birth rates, the death rates, the population density, the average income, etc. of the countries in the world are available, we can aggregate over them and compute the maximum or minimum value each country ever had for one of these attributes.

A special kind of aggregation is realized by the *project* operation which computes the projection of a spatio-temporal partition onto the Euclidean space and which yields a spatial partition. For each point in space, all labels, except for the undefined label denoting the outside, are collected over time. That is, if a point has always had the same single label over its lifetime, this single label will appear in the resulting partition and indicate a place that has never changed. On the other hand, points of the resulting partition with a collection of labels describe places where changes occurred. An example is the projection of a temporal map illustrating the water levels of lakes onto the Euclidean plane. The result shows those parts of lakes that have always, sometimes, and never been covered with water.

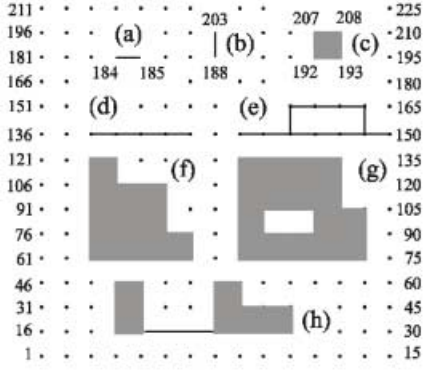
For a much more detailed description and, in particular, formal definition of spatio-temporal partitions, the reader is referred to [12]. There as well as in [15] especially a concept of "spatial continuity" and a "difference measure" for regions are defined.

### 4.5.3 On a Spatio-temporal Relational Model Based on Quanta

In this section an outline is given of a formalism for the definition of a spatio-temporal extension to the relational model. The formalism considers *temporal* and *spatial quanta* and, based on them, defines relevant data types. This way, a series of relational algebra operations can be defined, that are closed and enable the uniform management of either conventional or spatial or spatio-temporal data.

**Quanta and Data Types of Time.** A *generic* data type for *time* is defined as the set $I_n = \{1, 2, \ldots, n\}, n > 0$ [25]. The elements of $I_n$ are called *quanta of time* or (*time*) *instants*. Based on this data type, another *generic* data type is defined, PERIOD($I_n$), with elements of the form $[p, q] \equiv \{i \mid i \in I_n, p \leq i \leq q\}$ that are called *periods* (*of time*). If the elements of $I_n$ are replaced by $n$ consecutive dates, then the respective data types for time are DATE and PERIOD(DATE). In a similar manner, a variety of data types can be defined like TIME and TIMESTAMP, which are supported in SQL.

**Spatial Quanta and Spatial Data Types.** If $I_m = \{1, 2, \ldots, m\}, m > 0$, is a subset of the integers, then $I_m^2$ is finite. Each element of $I_m^2$ is a 2D point that can be identified uniquely by an integer (see Figure 4.12 for $m = 15$). If $p \equiv (i, j)$ is such a point, then $p_N \equiv (i, j + 1)$ and $p_E \equiv (i + 1, j)$ are the *neighbors* of $p$. Points $p, p_E, p_{NE} \equiv \{(i+1, j+1)\}$ and $p_N$ are *corner* points. In Figure 4.12, 193 and 207 are neighbors of 192, whereas 192, 193, 208 and 207 are *corner* points. Based on this terminology, the following *spatial quanta* are defined [26].




**Points**
{184}, {185}

**Lines**
Pure: (a), (b), (d), (e)
Degenerate: {184}, {185}

**Surfaces**
Pure: (c), (f), (g)
Hybrid: (h)
Degenerated to pure line:
(a), (b), (d), (e)
Degenerated to point: {184}, {185}

**Fig. 4.12.** Spatial quanta and spatial objects

*Quantum Point* : It is any set $\{p\}$, where $p$ is a 2D point (see {192}, {193}, {208} in Figure 4.12). The set of all the quantum points is denoted by $Q_{POINT}$.

*Quantum Line* : It is either a line segment $ql_{p,q}$ with edges two neighbor 2D points, $p$ and $q$, or a quantum point $\{p\}$ (see $ql_{184,185}, ql_{188,203}$ and {184} in Figure 4.12). Clearly, $ql_{p,q}$ consists of an infinite number of $R^2$ elements. The set of all the quantum lines is denoted by $Q_{LINE}$.

*Quantum Surface* : It is either the surface of a square $qs_{p,q,r,s}$, where $p$, $q$, $r$ and $s$ are corner 2D points or a quantum line (see $qs_{192,193,208,207}, ql_{184,185}$ and {184} in Figure 4.12). Clearly, $qs_{p,q,r,s}$ consists of infinitely many elements of $R^2$. The set of all these surfaces is denoted by $Q_{SURFACE}$.

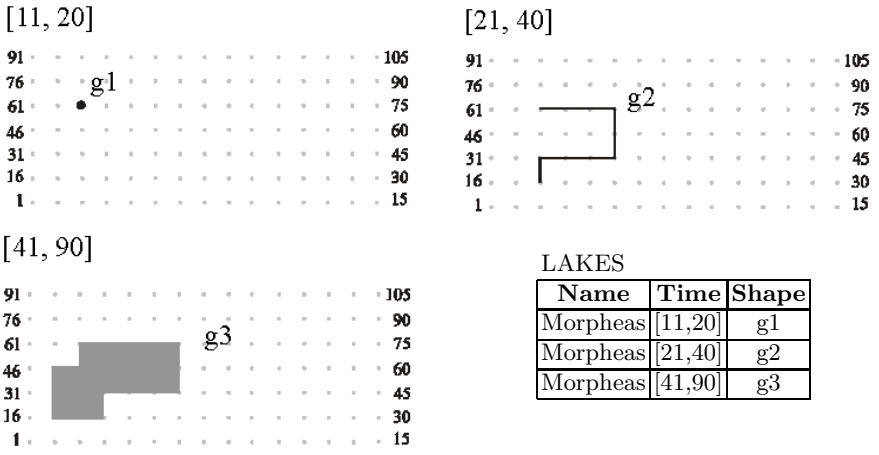
Assuming now that the concept of a *connected* set is known, it is defined that a non-empty connected subset $S = \bigcup_i q_i$ of $R^2$ is of a (2D)

- POINT data type if $q_i \in Q_{POINT}$
- LINE data type if $q_i \in Q_{LINE}$
- SURFACE data type if $q_i \in Q_{SURFACE}$

Given that $Q_{POINT} \subset Q_{LINE} \subset Q_{SURFACE}$, it follows that POINT $\subset$ LINE $\subset$ SURFACE. Examples of elements of the above data types are given in Figure 4.12. A point or line or surface element is called a *geo* or *spatial* object.

**Modeling of Spatio-temporal Data.** Based on the above data types, Figure 4.13 illustrates the evolution of a spatial object, Morpheas, with respect to time. As can be seen on the relevant plots, during the periods [11, 20], [21, 40] [41, 90], Morpheas was just a spring, a river and an actual lake, respectively. Relation LAKES, in the same figure, is used to record this evolution. The domain of attribute Shape is SURFACE and each of g1, g2 and g3 is a shorthand of the description of the geometry of Morpheas. In this model therefore, a *map* matches the *geometric interpretation* of the content of a relation that contains spatial data.

In [25] it has been shown that *period* is a special case of a more generic data type, interval. Functions and predicates for such data have been defined. A set of relational algebra operations has also been defined. It includes the well-known primitive operations, *Union, Except, Project, Cartesian Product, Select.* It also includes *Fold, Unfold* and some derived operations, whose formalism and functionality can also be found in [25]. Hence, only the application of *Fold* and *Unfold* on spatial data is illustrated below.




LAKES

| Name | Time | Shape |
|---|---|---|
| Morpheas | [11,20] | g1 |
| Morpheas | [21,40] | g2 |
| Morpheas | [41,90] | g3 |

**Fig. 4.13.** Representation of spatio-temporal data

*Fold* : Consider a table $R(\mathbf{A}, G)$, where $\mathbf{A}$ is a (possibly empty) list of attributes and $G$ is an attribute of some geo data type. Assume also that $(\mathbf{a}, g_i), i = 1, 2, \ldots, n$ are all those tuples of $R$ which satisfy the property that the *spatial union* of all $g_i$ yields a new spatial object $g$. Then all these tuples result in one tuple, $(\mathbf{a}, g)$, in relation $F = \mathit{Fold}[G](R)$. Consider for example the plane in Figure 4.12 and assume that $R = \{(x, \{2\}), (x, l_{2,3}l_{3,4}), (x, s_{3,4,19,18}), (y, s_{6,7,22,21}$ $s_{7,8,23,22}), (y, l_{6,7})\}$. Then $F = \{(x, l_{2,3}s_{3,4,19,18}), (y, s_{6,7,22,21}s_{7,8,23,22})\}$.

*Unfold* : Consider $R(\mathbf{A}, G)$ as before and let $g$ be a geo object. Consider also any geo object $g_i, i = 1, 2, \ldots, n$, of one quantum, which is a subset of $g$. Then a tuple $(\mathbf{a}, g)$ of $R$ yields in $U = \mathit{Unfold}[G](R)$ the tuples $(\mathbf{a}, g_i), i = 1, 2, \ldots, n$. Assuming for example that $R$ is the previous relation, $U = \{(x, \{2\}), (x, \{3\}), (x, \{4\}),$ $(x, \{18\}),(x, \{19\}), (x, l_{2,3}), (x, l_{3,4}), (x, l_{4,19}), (x, l_{19,18}), (x, l_{18,3}), (x, s_{3,4,19,18}),$ $(y, \{6\}),(y, \{7\}), (y, \{8\}), (y, \{21\}), (y, \{22\}), (y, \{23\}), (y, l_{6,7}), (y, l_{7,22}),$ $(y, l_{22,21}), (y, l_{21,6}),(y, l_{7,8}), (y, l_{8,23}), (y, l_{23,22}), (y, s_{6,7,22,21}), (y, s_{7,8,23,22})\}$. Based on the above two operations, a series of useful derived operations have also been defined [27], that enable the management of spatial data and, in conjunction with [25], spatio-temporal data. Regarding the management of spatial data, the functionality of these operations is relevant to that of *Spatial Union, Spatial Exception, Spatial Intersection, Overlay* etc that either have been defined by other researchers [5,22,34,35,29] or are supported by commercial Geographic Information Systems.

**Conclusions.** The advantages of the proposed model can be summarized as follows: All the algebraic operations are closed and, in conjunction with [25], they can be applied uniformly for the management of either spatio-temporal or spatial or temporal or conventional data. It has been identified, in particular, that certain operations, originally defined solely for the management of spatial data, are also of practical interest for the handling of temporal or conventional data. Hence, the algebra is not many-sorted and enables the uniform treatment of any of the above types of data. Regarding the case of spatial data, it has been identified that a *map* matches the geometric representation of a relation that contains such data. The model is also close to human intuition. By definition, for example, a line or a surface consists of an infinite number of 2D points, a line is treated as a degenerate surface and a point is treated as either a degenerate line or as a degenerate surface. Due to this, it is estimated that the model is also user-friendly. Finally, it is very general. This is not only because it can be applied to the previously mentioned types of data. It can also handle relations with many attributes of some *time* data type [25] and investigation results have shown that such relations may also contain $n$-dimensional spatial objects. Relevant research concerns implementation issues and the definition of an SQL extension.

### 4.5.4 Spatio-temporal Statement Modifiers

Data types and operators are generally embedded in some host language, which makes them available for use during data management. The characteristics of this

language in large part determines the difficulty in migrating existing applications to a new, spatio-temporal DBMS (STDBMS). The concept of a *statement modifier* extended host language [2,31,30], largely orthogonal to the specific abstract data types offered, enables he migration of legacy applications.

This section defines technical requirements to an STDBMS that provide a foundation for making it economically feasible to migrate legacy applications to an STDBMS. It proceeds to present the design of the core of a spatio-temporal, multi-dimensional extension to SQL–92, called STSQL, that satisfies the requirements. This is achieved by offering *upward compatible*, *dimensional upward compatible*, *reducible*, and *non-reducible* query language statements.

A planning and scheduling system (termed Ecoplan), which is used for forest management [28], serves to exemplify the new concepts. A `stands` table captures regions that are homogeneous with respect to soil fertility, wood specie, and average age. An `estates` table records the IDs of estates and their owners. An estate is a legal entity covering a geographical region, possibly including one or more forests. A `plans` table captures harvest plans, with each stand being associated with one or more plans (and vise versa), an estimated harvest volume for each stand, and an optimal harvest time (a so-called ripe year) of the stand.

**Migration Requirements.** Let $M = (DS, QL)$ be a data model with a data structure and a query language component. For query $s \in QL$ and database $db \in DS$, we define $\langle\!\langle s(db) \rangle\!\rangle_M$ as the result of applying $s$ to $db$ in data model $M$. We use the superscripts "$^s$" and "$^d$" to indicate snapshot and dimensional entities, respectively. The dimensional slice operator, $\tau_p^{M^d,M^s}$, takes a dimensional database $db^d$ and returns a snapshot database $db^s$ containing all tuples that are defined at point $p$.

**Definition 5. (UC)** Model $M_1$ is *upward compatible* with model $M_2$ iff

- $\forall db_2 \in DS_2\ (db_2 \in DS_1)$,
- $\forall s_2 \in QL_2\ (s_2 \in QL_1)$, and
- $\forall db_2 \in DS_2\ (\forall s_2 \in QL_2\ (\langle\!\langle s_2(db_2) \rangle\!\rangle_{M_2} = \langle\!\langle s_2(db_2) \rangle\!\rangle_{M_1}))$.

The conditions imply that all existing databases and query expressions in the old DBMS, captured by $M_2$, are also legal in the new DBMS, captured by $M_1$ and that all existing queries compute the same results in the new and old DBMSs.

**Definition 6. (DUC)** Model $M^d$ is *dimensional upward compatible* with model $M^s$ iff

- $M^d$ is upward compatible with $M^s$ and
- $\forall db^s \in DS^s\ (\forall\, \mathcal{U}\ (\forall q^s \in QL^s\ (\langle\!\langle q^s(\mathcal{U}(db^s)) \rangle\!\rangle_{M^s} = \langle\!\langle q^s(\mathcal{U}(\mathcal{D}(db^s))) \rangle\!\rangle_{M^d})))$.

DUC ensures that legacy applications remain operational even if the database is rendered dimensional. Intuitively, a query $q^s$ must return the same result on an associated snapshot database $db^s$ as on the dimensional counterpart of the database, $\mathcal{D}(db^s)$ (operator $\mathcal{D}$ adds dimensions to its argument). A sequence of

modification statements, $\mathcal{U}$, may not affect this. To satisfy DUC, legacy queries ignore spatial dimensions and are evaluated only on tuples with time periods that overlap *now*.

To illustrate the compatibility requirements, consider the following statements issued in an STDBMS satisfying UC and DUC with respect to SQL-92.

```
> SELECT * FROM plans;
> ALTER TABLE plans ADD harvest1 PERIOD AS VALID;
> SELECT * FROM plans;
```

The first statement is syntactically an SQL–92 query and is issued on the legacy table, `plans`. Due to UC, it returns the same result as in the old DBMS. The next statement alters `plans`, adding a valid-time dimension to indicate harvest periods of stands. The last statement, now on a dimensional table, yields the same result as the first due to DUC.

To generalize the relational model to a dimensional relational model, we adopt the view that a dimensional table is a collection of snapshot tables, each of which has an associated multi-dimensional point and contains all snapshot tuples with an associated multi-dimensional region that contains the point.

**Definition 7. (SR)** Data model $M^d$ is *snapshot reducible with respect to* data model $M^s$ iff

$$\forall q^s \in QL^s(\exists q^d \in QL^d(\forall db^d \in DS^d(\forall p(\tau_p^{M^d,M^s}(q^d(db^d)) = q^s(\tau_p^{M^d,M^s}(db^d))))))$$

In addition, it is desirable that $q^d$ be *syntactically similar snapshot reducible* to $q^s$ [4].

**Definition 8. (SSSR)** Data model $M^d$ is a *syntactically similar snapshot-reducible extension* of model $M^s$ iff

- data model $M^d$ is snapshot reducible with respect to data model $M^s$, and
- there exist two (possibly empty) strings, $S_1$ and $S_2$, such that each query $q^d$ in $QL^d$ that is snapshot reducible with respect to a query $q^s$ in $QL^s$ is syntactically identical to $S_1 q^s S_2$.

The SSSR requirement enables the SQL–92 programmer to easily formulate spatio-temporal queries.

```
> ALTER TABLE estates ADD es_area 2D_REGION AS SPACE;
> ALTER TABLE stands ADD st_area 2D_REGION AS SPACE;
> REDUCIBLE (es_area) AS area SELECT * FROM estates;
> REDUCIBLE (es_area, st_area) AS area
  SELECT es_ID, st_ID FROM estates, stands;
```

The first two statements render `estates` and `stands` dimensional. The two queries have an SQL–92 core. The prepended string, `REDUCIBLE`, is a *statement modifier* that determines the handling of the dimension attributes in the queries. The presence of `REDUCIBLE` implies that, conceptually, the queries are computed

point-by-point. More specifically, for each point in space, the legacy SQL statement following the statement modifier is evaluated on the snapshot database corresponding to that point. The results for each point in space are integrated into a single dimensional table.

Many useful dimensional queries cannot be specified as reducible generalizations of snapshot queries, and there is a need for queries where no built-in processing of the dimension attributes. The modifier `NONREDUCIBLE` specifies that dimension attributes are to be considered as regular attributes. Together with the predicates and functions offered by the dimensional data types, this yields full control over the dimension attributes.

```
> NONREDUCIBLE (es_area, st_area)
    SELECT s.st_ID, s.st_area FROM stands s WHERE  NOT EXISTS (
      SELECT * FROM estates e WHERE e.es_area CONTAINS s.st_area);
```

This query retrieves each stand for which no single estate exists that covers the stand's area. We consider the regions of the stands as being non-decomposable and constrain them with a spatial predicate. This contrasts the `REDUCIBLE` queries from before, where regions are decomposed into their constituent points.

**STSQL Design.** The first step in the design of STSQL is to introduce new, dimensional abstract data types. Time values are anchored time periods while spatial values are unions of regions. The corresponding data types are `PERIOD`, `1D_REGION`, `2D_REGION` and `3D_REGION`, respectively [3]. The second step is to make tables *dimensional*, by enabling the designation of certain time or space valued attributes as dimensional. In STSQL, a dimension attribute is specified as either a `VALID`, a `TRANSACTION`, or a `SPACE` attribute.

STSQL permits a table to have any number of dimension attributes. This generality is useful for many purposes. Several `VALID`-type attributes may record different temporal aspects of a tuple. For example, the `plans` table has a `VALID` attribute `harvest1` recording when a stand is supposed to be harvested. We add another `VALID` attribute `harvest2` that records an alternative harvest period. The two resulting harvest attributes reflect different (possible) worlds. It is equally easy to envision uses of multiple space dimension attributes: the multiple-worlds argument applies equally well to space, and tuples may have several different kinds of spatial aspects [6].

When formulating queries on dimensional tables, it is advantageous to proceed in several steps. All dimensions are initially ignored, and the core STSQL query, typically an SQL-92 query, is formulated. Next, the query's statement modifier is specified. For each dimension of each table in the query, it must be stated how the dimension is used in the query. Specifically, each dimensions that should be evaluated with reducible semantics is identified. Each occurrence of `REDUCIBLE` requires the participation of exactly one dimension from each table. Following this, each dimensions to be given `NONREDUCIBLE` semantics is identified. This semantics is chosen if we want to formulate user-defined predicates

(e.g., `CONTAINS`) on the dimension attribute or if we want to override the DUC-consistent semantics, which are given to dimension attributes not mentioned in the statement modifier.

We conclude by formulating the following query: *for each stand that is ripe in 2000, determine its harvest periods.* The `stands` and the `plans` tables are joined using a reducible join over the valid times, to associate stands with relevant plans only. Next, we are only interested in the current data on `stands`. Assuming that a transaction-time dimension `st_tt` has been added to `stands`, we want to consider only tuples that overlap *now*. This is the semantics provided by DUC, so no modifier is specified for `st_tt`. The location of a stand is not relevant and, thus, must be disregarded. This is again DUC semantics, so no modifier for `st_area` is specified. Finally, we want to retrieve (and handle) the harvest periods as regular attributes. This is achieved by specifying a non-reducible modifier for these dimensions.

```
> REDUCIBLE (st_vt,pl_vt) AS vt AND NONREDUCIBLE (harvest1, harvest2)
    SELECT st.st_ID, harvest1, harvest2
    FROM   stands st, plans pl
    WHERE  pl.st_ID = st.st_ID AND pl.ripe = 2000;
```

**Conclusion and Future Research.** We formulated requirements to a new dimensional DBMS aiming at addressing legacy-related concerns. The objectives are to make it possible for legacy database applications using a conventional SQL-92-based DBMS to be migrated to a dimensional DBMS without without affecting the legacy applications, while also reusing programmer expertise. A spatio-temporal extension to SQL–92, termed STSQL, that provides built-in data management support for spatio-temporal data has been designed to meet the above requirements.

Several directions for further explorations may be identified. First, we have only described the initial design of the core of STSQL, and further formalizations of the language are in order. Next, we have chosen one possible semantics for DUC statements. Other semantics are possible; further studies are needed to identify these and explore their utility.

## References

1. J.K. Berry. Fundamental Operations in Computer-Assisted Map Analysis. *Int. Journal of Geographic Information Systems*, 1(2):119–136, 1987.
2. M.H. Böhlen and C. S. Jensen. Seamless Integration of Time into SQL. Technical Report R-96-49, Department of Computer Science, Aalborg University.
3. M.H. Böhlen, C.S. Jensen, and B. Skjellaug. Spatio-Temporal Database Support for Legacy Applications. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pp. 226–234. Atlanta, Georgia, February 1998.
4. M.H. Böhlen, C.S. Jensen, and R.T. Snodgrass. Evaluating the Completeness of TSQL2. In *Recent Advances in Temporal Databases, International Workshop on Temporal Databases*, pp. 153–172. Springer, Berlin, Zürich, Switzerland, September 1995.

5. E.P.F. Chan and R. Zhu. QL/G - A Query Language for Geometric Data Bases. In *Proc. 1st International Conference on GIS, Urban Regional and Environmental Planning*, pp. 271–286. Samos, Greece, 1996.
6. H. Couclelis. People Manipulate Objects (but Cultivate Fields): Beyond the Raster-Vector Debate in GIS. In *Lecture Notes in Computer Science*, Vol. 639, pp. 65–77, Springer-Verlag, 1992.
7. M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):265–291, 1999.
8. M. Erwig and B. Meyer. Heterogeneous Visual Languages – Integrating Visual and Textual Programming. In *11th IEEE Symp. on Visual Languages*, pp. 318–325, 1995.
9. M. Erwig and M. Schneider. Partition and Conquer. In *3rd Int. Conf. on Spatial Information Theory*, LNCS 1329, pp. 389–408, 1997.
10. M. Erwig and M. Schneider. Developments in Spatio-Temporal Query Languages. In *IEEE Int. Workshop on Spatio-Temporal Data Models and Languages*, pp. 441–449, 1999.
11. M. Erwig and M. Schneider. Spatio-Temporal Predicates. Technical Report 262, FernUniversität Hagen, 1999.
12. M. Erwig and M. Schneider. The Honeycomb Model of Spatio-Temporal Partitions. In *Int. Workshop on Spatio-Temporal Database Management*, LNCS 1678, pp. 39–59, 1999.
13. M. Erwig and M. Schneider. Visual Specifications of Spatio-Temporal Developments. In *15th IEEE Symp. on Visual Languages*, pp. 187–188, 1999.
14. M. Erwig and M. Schneider. Query-By-Trace: Visual Predicate Specification in Spatio-Temporal Databases. In *5th IFIP Conf. on Visual Databases*, 2000. To appear.
15. M. Erwig, M. Schneider, and R.H. Güting. Temporal Objects for Spatio-Temporal Data Models and a Comparison of Their Representations. In *Int. Workshop on Advances in Database Technologies*, LNCS 1552, pp. 454–465, 1998.
16. L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, 2000.
17. S. Gaal. *Point Set Topology*. Academic Press, 1964.
18. R.H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 277–286. Washington, 1993.
19. R.H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3:357–399, 1994.
20. R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M.Vazirgiannis. A Foundation for Representing and Querying Moving Objects. Technical Report Informatik 238, FernUniversität Hagen, 1998. Available at `http://www.fernuni-hagen.de/inf/pi4/papers/Foundation.ps.gz`.
21. R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M.Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1), 2000.
22. R.H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):100–143, 1995.
23. Z. Huang, P. Svensson, and H. Hauska. Solving Spatial Analysis Problems with GeoSAL, a Spatial Query Language. In *6th Int. Working Conf. on Scientific and Statistical Database Management*, 1992.

24. J. Loeckx, H. D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. John Wiley & Sons, Inc. and B.G. Teubner Publishers, 1996.
25. N.A. Lorentzos and Y.G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.
26. N.A. Lorentzos, N. Tryfona, and J.R. Rios Viqueira. Relational Algebra for Spatial Data Management. In *Proc. International Workshop Integrated Spatial Databases: Digital Images and GIS*. Portland, Maine, June 1999.
27. N.A. Lorentzos, J.R. Rios Viqueira, and N. Tryfona. Quantum-Based Spatial Data Model. Technical Report, Informatics Laboratory, Agricultural University of Athens, 2000.
28. G. Misund, B. Johansen, G. Hasle, and J. Haukland. Integration of geographical information technology and constraint reasoning — A promising approach to forest management. Technical Report STF33A 95009, SINTEF Applied Mathematics, Oslo, Norway, June 1995.
29. M. Scholl and A. Voisard. Thematic Map Modeling. In *1st Int. Symp. on Large Spatial Databases*, LNCS 409, pp. 167–190, 1989.
30. R.T. Snodgrass, M.H. Böhlen, C.S. Jensen, and A. Steiner. Adding Transaction Time to SQL/Temporal. ANSI X3H2-96-152r, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-143, May 1996.
31. R.T. Snodgrass, M.H. Böhlen, C.S. Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. ANSI X3H2-96-151r1, ISO–ANSI SQL/Temporal Change Proposal, ISO/IEC JTC1/SC21/WG3 DBL MCI-142, May 1996.
32. R.B. Tilove. Set Membership Classification: A Unified Approach to Geometric Intersection Problems. *IEEE Transactions on Computers C-29*, pp. 874–883, 1980.
33. C.D. Tomlin. *Geographic Information Systems and Cartographic Modeling*. Prentice Hall, 1990.
34. J.W. van Roessel. Conceptual Folding and Unfolding of Spatial Data for Spatial Queries. In V.B. Robinson and H. Tom, eds., *Towards SQL Database Extensions for Geographic Information Systems*, pp. 133–148, National Institute of Standards and Technology, Gaithersburg, Maryland, 1993. Report NISTIR 5258.
35. J.W. van Roessel. An Integrated Point-Attribute Model for Four Types of Areal Gis Features. In R.G. Healey T.C. Waugh, ed., *Proc. 6th International Symposium on Spatial Data Handling (SDH94)*, pp. 127–144. Edinburgh, Scotland, UK, 1994.