A Foundation for Representing and Querying Moving Objects*

Ralf Hartmut Güting†, Michael H. Böhlen‡, Martin Erwig†, Christian S. Jensen‡,
Nikos A. Lorentzos§, Markus Schneider†, and Michalis Vazirgiannis¶

September 7, 1998

Abstract

Spatio-temporal databases deal with geometries changing over time. The goal of our work is to provide a DBMS data model and query language capable of handling such time-dependent geometries, including those changing continuously which describe *moving objects*. Two fundamental abstractions are *moving point* and *moving region*, describing objects for which only the time-dependent position, or position and extent, are of interest, respectively. We propose to represent such time-dependent geometries as attribute data types with suitable operations, that is, to provide an abstract data type extension to a DBMS data model and query language.

This paper presents a design of such a system of abstract data types. 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 non-temporal and related temporal types. Satisfying these goals leads to a simple and expressive system of abstract data types that may be integrated into a query language to yield a powerful language for querying spatio-temporal data, including moving objects. The paper formally defines the types and operations, offers detailed insight into the considerations that went into the design, and exemplifies the use of the abstract data types using SQL. The paper offers a precise and conceptually clean foundation for implementing a spatio-temporal DBMS extension.

1 Introduction

A common characteristic of concrete, physical objects is that they have a position and an extent in space at any point in time. This applies to countries, land parcels, rivers, taxis, forest harvesting equipment, fishing boats, air planes, glaciers, lakes, forests, birds, polar bears, and persons, to name but a few types of objects.

*This work was partially supported by the CHOROCHRONOS project, funded by the EU under the Training and Mobility of Researchers Programme, Contract No. ERB FMRX-CT96-0056.

†Praktische Informatik IV, FernUniversität Hagen, D-58084 Hagen, Germany, {gueting, erwig, markus.schneider}@fernuni-hagen.de

‡Dept. of Computer Science, Aalborg University, DK-9220, Aalborg Øst, Denmark, {boehlen, csj}@cs.auc.dk

§Informatics Laboratory, Agricultural University of Athens, Iera Odos 75, 11855 Athens, Greece, lorentzos@auadec.aua.ariadne-t.gr

¶Dept. of Informatics, Athens University of Economics and Business, Patision 76, 10434 Athens, Greece, mvazirg@aueb.gr

A wide and increasing range of database applications manage such space and time referenced objects, termed spatio-temporal objects. In these database applications, the current as well as the past and anticipated future positions and extents of the objects are frequently of interest. This brings about the need for capturing these aspects of the objects in the database.

As an example, forest management involves the management of spatio-temporal objects. Forest harvesting machines have Global Positioning System (GPS) devices attached. A harvesting machine cuts down a pine tree while holding on to the tree; it then strips off the branches while simultaneously cutting the tree into logs of specified lengths, placing also the logs in different piles so that similar logs go into the same pile. During this process, the machine measures the amount and properties of the harvested wood (e.g., volumes, diameters, lengths) and transmits this information together with the positions of the piles to head quarters. Together with the orders for wood, this information along with the present locations of the harvesting machines then is used for scheduling the pickup of already harvested wood as well as further harvesting.

Two types of spatio-temporal objects may be distinguished, namely discretely moving objects and continuously moving objects. For the former type of object, e.g., land parcels, it is relatively easy to keep track in the database of the objects' changing positions and extents. This may be accomplished by more or less frequent database updates, and solutions exist for capturing and querying discretely changing spatial positions and extents. For example, this may be accomplished by using separate spatial and temporal columns in relational tables.

Objects that change position or extent continuously, termed moving objects for short, are pervasive, but in contrast to the discretely changing objects, they are much more difficult to accommodate in the database. Supporting these kinds of moving objects is exactly the challenge addressed by this paper. It is not feasible to capture these with separate spatial and temporal values, and the database cannot be updated for each change to the objects' spatial aspect. Another tack must be adopted.

The paper defines a complete framework of abstract data types for moving objects. The proposed framework is intended to serve as a precise and conceptually clean foundation for the representation and querying of spatio-temporal data. While proposals exist for spatial and temporal types, no framework has previously been proposed for spatio-temporal types that include support for moving objects. (Section 6 positions the paper's contribution with respect to related research.)

The framework takes as its outset a set of basic types that includes standard data types such as integer and Boolean; spatial data types, including point and region; and the temporal type instant. The next step is to introduce type constructors that may be applied to the basic types, thus creating new types. For example, the type constructor "moving" that maps an argument type to the type that is a mapping from time to the argument type is included. This leads to types such as moving point, which is a function from instant to point. For example, a harvesting machine's position may be modeled as a moving point.

The framework emphasizes three properties, namely closure, simplicity, and expressiveness. For example, closure dictates that types exist for the domains and ranges of types that are functions between types.

It is important to note that in a design of abstract data types like the one of this paper, the definitions of the structure of entities (e.g., values of spatial data types) and of the semantics of operations can be given at different levels of abstraction. For example, the trajectory of a moving point can be described either as a curve or as a polygonal line in two-dimensional space. In the first case, a curve is defined as a (certain kind of) infinite set of points in the plane *without fixing any finite representation.* In the second case, the definition uses a finite representation of a polygonal line, which in turn defines the infinite point set making up the trajectory of the

moving point. In [EGSV97] the difference between these two levels of modeling is discussed at some depth, and the terms *abstract* and *discrete* modeling have been introduced for them. Basically, the advantage of the abstract level is that it is conceptually clean and simple, because one does not have to express semantics in terms of the finite representations. One is also free to select later different kinds of finite representations, e.g., polygonal lines, or descriptions based on splines. On the other hand, this additional step of fixing a finite representation is still needed. The advantage of discrete modeling is that it is closer to implementation.

The design of this paper is an abstract model in this sense. However, care has been taken to define all data types and operations in such a way that an instantiation with finite representations (e.g., set of polygons for region) is possible without problems.

The proposed abstract data types may be used as column types in conventional relational DBMSs, or they may be integrated in object-oriented or object-relational DBMS's. It is also possible for a user or a third-party developer to implement abstract data types based on this paper's definitions in an extensible DBMS, e.g., a so-called Universal Server.

The paper is structured as follows. Abstract data types consist of data types and operations that encapsulate the data types, i.e., they form an *algebra*. Section 2 discusses the embedding of such an algebra into a query language. Section 3 proceeds to present the data types in the framework, and Section 4 defines the appropriate sets of operations to go with the data types. Section 5 explores the expressiveness of the resulting language within two application areas. Section 6 covers related research. Section 7 concludes the paper and identifies promising directions for future research pointed to by the paper.

2 Preliminaries: Language Embedding

In order to illustrate the use of the framework of abstract data types in queries, these must be embedded in a query language. A range of languages would suffice for this purpose, including theoretical and practical languages as well as relational, object-relational, and object-oriented languages.

We do not care into which language our design, which can be viewed as an application-specific sublanguage, is embedded. In the examples of this paper we show an embedding into a relational model and an SQL-like language with which most readers should be familiar.

To achieve a smooth interplay between the embedding language and an embedded system of abstract data types, a few interface facilities and notations are needed, expressible in one form or another in most object-oriented or object-relational query languages. In order to not be bound to any particular SQL standard, we briefly explain our notations for these facilities.

Assignments. The construct `LET <name> = <query>` assigns the result of `query` to a new object called `name` which can then be used in further steps of a query.

Multistep queries. A query can be written as a list of assignments, separated by semicolon, followed by one or more query expressions. The latter are displayed as the result of the query.

Example 2.1 We assume an example relation

`employee(name:`*string*`, salary:`*int*`, permanent:`*bool*`)`

Here is a multistep query.

```
LET big = 10000;
LET well_paid = SELECT name, salary FROM employee WHERE salary > big;
SELECT SUM(salary) FROM well_paid
```

The result of the last SELECT statement is displayed. □

Conversions between sets of objects and atomic values. In relational terms, this means that a relation with a single tuple and a single attribute can be converted into a typed atomic value and vice-versa. We use the notations ELEMENT(<query>) and SET(<attrname>, <value>) for this.

Example 2.2 The expression

```
ELEMENT(SELECT salary FROM employee WHERE name = "John Smith") > 100000
```

is a good predicate, because the ELEMENT construct returns a value of type *integer*. Conversely, the expression

```
SET(name, "John Smith")
```

returns a relation with an attribute name and a single tuple having John Smith as the value of that attribute. □

Defining derived attributes. We assume that arbitrary ADT operations over new or old data types may occur anywhere in a WHERE clause as long as in the end a predicate is constructed, and they can be used in a SELECT clause to produce new attributes, with the notation

```
<new attrname> AS <expression>
```

Example 2.3 Here a new attribute thousands is derived.

```
SELECT name, thousands AS salary div 1000 FROM employee
```

We generally typeset ADT operators in bold face, so **div** is assumed to be an ADT operator here. □

Defining operations. We allow for the definition of new operations derived from existing ones, in the form LET <name> = <functional expression>.

Example 2.4 This example shows how functional expressions are written and may be used.

```
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 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, a name for the aggregate function can be defined by LET <name> = AGGREGATE(<operator>, <neutral element>).

Example 2.5 We can sum all salaries by

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

We can determine whether all employees have permanent positions by:

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

Whereas we have explained all these facilities in terms of simple standard data types, they are much more useful in the handling of complex non-standard data types. – A deeper study about interfacing ADT algebras with embedding languages can be found in [GS95].

3 Spatio-Temporal Data Types

In this and the next section we define a system of data types and operations, or an *algebra*, suitable for representing and querying geometries changing over time, and in particular, moving objects. Defining an algebra consists of two steps. In a first step we design a type system by introducing some basic types as well as some type constructors. For each type in the type system, its semantics is given by defining a *carrier set*. In the second step we design a collection of operations over the types of the type system. For each operation, its signature is defined, describing the syntax of the operation, i.e., the correct argument and result types, and its semantics is given by defining a function on the carrier sets of the argument types.

In this section we define the type system; operations are given in Section 4. We first discuss requirements, then define basic types and type constructors, and finally justify some of the particular choices made in the design.

3.1 Requirements and Scope

A fundamental requirement to this design is that the types enable us to record in the database the spatio-temporal aspects of the objects. We focus on capturing spatio-temporal aspects that change continuously as this is the most challenging and allows us to also capture spatio-temporal aspects that only change in discrete steps.

The outset for the spatio-temporal type system is the standard database types traditionally built into relational database management systems, termed *base types* and represented here by the integers, reals, strings, and Booleans; the *spatial types* of point, line, and region; and the *time type* of instants.

Another fundamental requirement to the type system is that it be orthogonal. With this requirement in mind, it is possible to construct various functional types over the types just mentioned as follows.

- Functions from the domain of points in 2D space to some other domain (e.g., a temperature distribution over a country),
- functions from the domain of instants to some other domain, called *temporal types*, and
- functions from the domain of point-instant pairs to some domain.

Not all of these types are equally interesting in the context of this paper that focusses on moving objects. In keeping with this focus, we will restrict our attention to consider explicitly only (i) base types, spatial types and time types, and, primarily, (ii) temporal types, of which spatio-temporal types (= spatial temporal types) are a special case. The temporal types are also termed "moving," and since temporal base types have been covered extensively in the research literature on temporal databases, emphasis is given to spatio-temporal types.

A third requirement is that the type system be closed. This means that the domains and ranges of all the "functional" types as well as the ranges of the inverse functions must be present in the type system. This requirement dictates the presence of a time type of sets of time intervals, for example.

3.2 The Type System

We define the type system as a signature. Any (many-sorted) signature consists of sorts and operators, where the sorts control the applicability of operators (see e.g. [LEW96]). A signature

generates a set of terms. Here the sorts are called *kinds* and describe certain subsets of types, and instead of operators we have type constructors. The terms generated by the signature describe exactly the types available in our type system. For more background on this technique for defining type systems and algebras see [Güt93].

Table 1 shows the signature defining our type system.

Type constructor	Signature	
int, *real*, *string*, *bool*		$\rightarrow BASE$
point, *points*, *line*, *region*		$\rightarrow SPATIAL$
instant		$\rightarrow TIME$
moving, *intime*	$BASE \cup SPATIAL$	$\rightarrow TEMPORAL$
range	$BASE \cup TIME$	$\rightarrow RANGE$

Table 1: Signature describing the type system

Terms, and therefore types, generated by this signature are e.g., *int*, *region*, *moving*(*point*), *range*(*int*), etc. The *range* type constructor is applicable to all the types in the kind *BASE* and all types in kind *TIME*, hence all types that can be constructed by it are *range*(*int*), *range*(*real*), *range*(*string*), *range*(*bool*), and *range*(*instant*).

One can see that quite a few types are around. Although the focus of interest are the spatio-temporal data types, especially *moving*(*point*) and *moving*(*region*), in order to obtain a closed system of operations it is necessary to include the related spatial, time, and base types into the design.

So far we have just introduced some names for types. In the sequel we describe their semantics first informally, and then formally by defining carrier sets.

We start with the constant types (type constructors with no arguments), and then discuss (proper) type constructors. Throughout the paper, type constructors, including constant types, are typeset in *italics*.

3.2.1 Base Types

The base types are *int*, *real*, *string*, and *bool*. All base types have the usual interpretation, except that each domain is extended by the value $\perp$ (undefined).

Definition 3.1 The carrier sets for the types *int*, *real*, *string*, and *bool*, are defined as:

$A_{int} \triangleq \mathbb{Z} \cup \{\perp\}$,
$A_{real} \triangleq \mathbb{R} \cup \{\perp\}$,
$A_{string} \triangleq V^* \cup \{\perp\}$, where V is a finite alphabet,
$A_{bool} \triangleq \{FALSE, TRUE\} \cup \{\perp\}$. □

We sometimes need to talk about the carrier set without the undefined value. As a shorthand for this we define $\bar{A}_\alpha \triangleq A_\alpha \setminus \{\perp\}$.

3.2.2 Spatial Types

Basic conceptual entities that have been identified in spatial database research are point, line, and region [Güt94]. A point is a suitable representation for an object for which only the position, not the extent, is of interest. A region is the abstraction of an object for which the position

and the extent are relevant. A line is (in most cases) an abstraction for ways of moving through space, or for connections through space (roads, rivers, electricity networks, gas lines, etc.).

The focus of our interest are point and region entities changing over time, as these are most relevant for applications involving moving objects. Lines (connection structures) changing over time do exist, but are not as prominent as the other two. However, as we will see below, to obtain a closed system it is necessary to have data types for lines and even for collections of points available.

The spatial data types used in our design are called *point*, *points*, *line*, and *region*. They are illustrated in Figure 1.

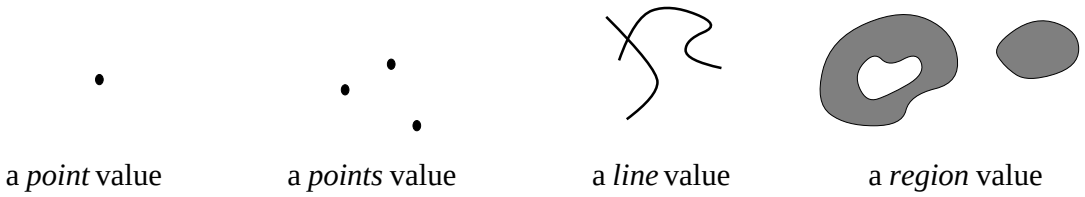

Figure 1: The spatial data types

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 faces each of which may have holes. It is allowed that a face lies within a hole of another face. Each of the three finite set types may be empty. More precise conditions on the structures are given next.

Formal definitions are based on the point set paradigm and on point set topology. The point set paradigm expresses that space is composed of infinitely many points and that spatial objects are distinguished subsets of space which are viewed as entities. Point set topology provides concepts specifying the notions of continuity and closeness and allows one to identify special topological structures of a point set like its interior, closure, boundary, and exterior. We assume that the reader is familiar with the fundamental concepts of point set topology like those just mentioned and others such as topological space, open and closed sets, etc., and refer to textbooks such as [Gaa64].

Point and point set types are still quite simple:

Definition 3.2 The carrier sets for the types *point* and *points* are:

$A_{point} \triangleq \mathbb{R}^2 \cup \{\perp\}$,

$A_{points} \triangleq \{P \subseteq \mathbb{R}^2 \mid P \text{ is finite}\}$ □

For the definition of lines, we need the concept of a (continuous) curve. In the following definition, a total order on $\mathbb{R}^2$ is assumed, namely lexicographic order on the coordinates (first x, then y), and $<$ refers to that order.

Definition 3.3 A *curve* is a mapping $f : [0,1] \to \mathbb{R}^2$ such that

1. f is continuous
2. $\forall a, b \in (0,1) : a \neq b \Rightarrow f(a) \neq f(b)$
3. $\forall a \in \{0,1\}, \forall b \in (0,1) : f(a) \neq f(b)$

4. $f(0) < f(1) \vee (f(0) = f(1) \wedge \forall a \in (0,1) : f(0) < f(a))$

The points $f(0)$ and $f(1)$ are called the *end points* of f. The corresponding point set in the range of f is denoted by $rng(f)$. □

The definition allows loops ($f(0) = f(1)$) but forbids equality of different interior points and equality of an interior with an end point. The last condition ensures uniqueness of representation, e.g. in a closed curve, $f(0)$ must be the leftmost point.

The curves that we want to deal with must be simple in the sense that the intersection of two curves yields only a finite number of proper intersection points (disregarding common parts). This is ensured by the following definitions.

Definition 3.4 Let $Q \subseteq \mathbb{R}^2$ and $p \in Q$. p is called *isolated in* $Q :\Leftrightarrow \exists \epsilon \in \mathbb{R}, \epsilon > 0 : Sp(p, \epsilon) \cap (Q \setminus \{p\}) = \emptyset$.

Here $Sp(p, \epsilon)$ denotes an open sphere around p with radius ϵ. The set of all isolated points in Q is denoted as $isolated(Q)$. □

Definition 3.5 Let C be the set of all curves w.r.t. Def. 3.3. A class of curves $C' \subset C$ is called *simple* $:\Leftrightarrow \forall c_1, c_2 \in C' : isolated(rng(c_1) \cap rng(c_2))$ is finite. □

The *line* data type is to represent any finite union of curves from some class of simple curves. When the abstract design of data types given in this paper is implemented by some discrete design (as explained in the introduction), 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 graph structure embedded into the plane. Given a set of points of such a graph, there are many different sets of curves resulting in this point set. We obtain a unique representation by enforcing that all curves are disjoint except for the end points.

Definition 3.6 Let S be a class of curves. A *C-complex over* S is a finite set of curves $C \subseteq S$ such that $\forall c_1, c_2 \in C, \forall a, b \in (0,1) : c_1(a) \neq c_2(b)$.

The set of points of this C-complex, denoted $points(C)$, is $\bigcup_{c \in C} rng(c)$.

The set of all C-complexes over S is denoted by $CC(S)$. □

Definition 3.7 Let S be a simple class of curves. The carrier set of the *line* data type is:

$A_{line} \triangleq \{Q \subseteq \mathbb{R}^2 \mid \exists C \in CC(S) : points(C) = Q\}$ □

For the definition of regions, we need the concept of a regular closed set. A set $Q \subseteq \mathbb{R}^2$ is called *regular closed* if the closure of its interior coincides with the set itself, i.e., $Q = closure(interior(Q))$. The reason for this regularization process is that regions should not have geometric anomalies like isolated or dangling line or point features and missing lines and points in the form of cuts and punctures. These are avoided by regularity.

A *region* can be viewed as a finite set of *faces*. Each face is a non-empty, regular closed set. Any two faces of a region are disjoint except for finitely many "touching points" at the boundary.

Definition 3.8 Let Q, R be two regular closed sets. Q and R are *quasi-disjoint* $:\Leftrightarrow Q \cap R$ is finite. □

Definition 3.9 Let S be a class of curves. An *R-complex over* S is a finite set R of regular closed sets, such that:

1. Any two distinct elements of R are quasi-disjoint.
2. $\forall r \in R, \exists c \in CC(S) : \partial r = points(c)$

Here ∂r denotes the boundary of r. Each element of the R-complex is called a *face*. The union of all points of all faces is denoted $points(R)$. The set of all R-complexes over S is denoted $RC(S)$. □

An R-complex captures the intuition of a set of disjoint faces. In addition, it ensures that boundaries of faces are simple in the same sense that lines are simple. For example, the intersection of two regions will also produce only finitely many isolated intersection points. Note that the boundary of a face has outer as well as possibly inner parts, i.e., the face may have holes.

Definition 3.10 Let S be a simple class of curves. The carrier set of the *region* data type is defined as:

$A_{region} \stackrel{\triangle}{=} \{Q \subseteq \mathbb{R}^2 \mid \exists R \in RC(S) : Q = points(R)\}$ □

We require that the same class S of curves is used in the definition of the *line* and the *region* data type.

We extend the shorthand $\bar{A}$ to the spatial data types, and in fact to all types whose carrier set contains sets of values. For these types α we define $\bar{A}_\alpha \stackrel{\triangle}{=} A_\alpha \setminus \{\emptyset\}$.

3.2.3 Time Type

Type *instant* represents a point in time or is undefined. Time is considered to be linear and continuous, i.e., isomorphic to the real numbers.

Definition 3.11 The carrier set for *instant* is:

$A_{instant} \stackrel{\triangle}{=} \mathbb{R} \cup \{\perp\}$ □

3.2.4 Temporal Types

From the standard base types and base space types, we want to derive corresponding temporal types. The type constructor *moving* is used for this purpose. The *moving* type constructor yields for any given type α a mapping from time to α. More precisely, this means:

Definition 3.12 Let α be a data type to which the *moving* type constructor is applicable, with carrier set A_α. Then the carrier set for $moving(\alpha)$, is defined as follows:

$A_{moving(\alpha)} \stackrel{\triangle}{=} \{f \mid f : \bar{A}_{instant} \to \bar{A}_\alpha \text{ is a partial function} \wedge \Gamma(f) \text{ is finite}\}$ □

Hence, each value f from the carrier set of $moving(\alpha)$ is a function describing the development over time of a value from the carrier set of α. The condition "$\Gamma(f)$ is finite" says that f consists of only a finite number of continuous components. This is made precise in Appendix A where a generalized notion of continuity is defined. This condition is needed to ensure (i) that projections

of moving objects have only a finite number of components, (ii) for the **decompose** operation defined below, and (iii) as a precondition to make 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. The type constructor *intime* serves this purpose. The resulting family of types is used, for example, to represent the result of operations (e.g., **atinstant**) that for values of a temporal type yield the value at a specified instant (a time-slice of the value). The *intime* type constructor converts a given type α into a type that associates instants of time with values of α:

Definition 3.13 Let α be a data type to which the *intime* type constructor is applicable, with carrier set A_α. Then the carrier set for $intime(\alpha)$, is defined as follows:

$$A_{intime(\alpha)} \stackrel{\triangle}{=} A_{instant} \times A_\alpha$$ □

3.2.5 Range Types (Sets of Intervals)

For all temporal types we would like to have operations that correspond to projections into the domain and the range of the functions. 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. Such types are obtained through a *range* type constructor.

Definition 3.14 Let α be a data type to which the *range* type constructor is applicable (and hence on which a total order $<$ exists). An *α-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 α-intervals are *adjacent*, if they are disjoint and their union is an α-interval. An *α-range* is a finite set of disjoint, non-adjacent intervals. For an α-range R, $points(R)$ denotes the union of all its intervals.

An α-interval X is left-closed if $\inf(X) \in X$, it is right- closed if $\sup(X) \in X$, and it is closed if it is both left-closed and right-closed. Dually, it is (left-/right-) open iff it is not (left-/right-) closed. The closure of an α-interval X is defined as $closure(X) \stackrel{\triangle}{=} X \cup \{\inf(X)\} \cup \{\sup(X)\}$. □

Definition 3.15 Let α be any data type to which the *range* type constructor is applicable. Then the carrier set for $range(\alpha)$ is:

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

Because we are particularly interested in ranges over the time domain we introduce a special name for this type: $periods = range(instant)$.

3.3 Rationale for this Design

At this point, one may wonder why the design of data types was done in this particular way, and ask questions like the following:

1. What are all the data types for? For example, if we are interested in moving points and regions, why do we bother about lines? Why are there separate *point* and *points* data types? Why do we need "moving reals" or sets of intervals of integers?

2. Why do some types have the undefined value included, others not?

3. Why are the formal definitions done in this peculiar way? For example, why do we have the finiteness conditions in some places although generally we are talking about infinite point sets?

Although a full understanding of these issues will be possible only after reading Section 4, let us already point out the following design principles. The main concerns are closure and consistency.

1. *Closure and consistency between collections of non-temporal and temporal types.*

   Closure means that for a certain collection of non-temporal types we want to have corresponding temporal types, and for each temporal type, we want to have the corresponding non-temporal type available. The collection of types here are those in $BASE \cup SPATIAL$ to which we will also refer as *kernel types* in the sequel.

   Consistency means that given a temporal type such as *moving*(*region*), its value at each instant is a value of the corresponding non-temporal type *region*. Moreover, for each non-temporal type for which a temporal version exists, the temporal type can represent the development over time of any of its values. This principle would be violated, for example, if a moving region could consist only of a single connected volume (in the 2D+time space), whereas a region might have several disjoint components.

2. *Closure and consistency of operations on non-temporal types under temporal "lifting".*

   In most cases, an operation that is of interest in the non-temporal case is relevant in the temporal case as well. For example, we are interested in the distance between two points, but also in the time-dependent distance between two moving points. To keep this simple and uniform, in Section 4 we first define an algebra over the kernel types, the *kernel algebra*. We then require that for each operation in the kernel algebra any of its time-dependent variants exists as well, and its semantics is consistent with that of the kernel operation.

3. *For all temporal data types, time-slice, projection into the (time) domain, and projections into the range must be available.*

   For example, we must have types to represent the trajectory of a *moving*(*point*), or the set of values assumed by a *moving*(*real*).

4. *The generic point vs. point set (or element vs. set) view must be supported by all kernel data types.*

   There is a major organizing principle in designing generic operations in Section 4, which is to consider interaction between single values and sets of values of some domain. This means, for each domain, we need a type for single values and one for sets of values. For example, *int* and *range*(*int*) are these related types for integers, whereas in 2D *point* is the single value type, and *points*, *line*, and *region* are the set types.

5. *All set types must be closed under set operations.*

   By set operations we mean union, intersection, and difference of the underlying point sets. For example, there must be types to represent the result of intersecting two regions.

6. *We want to have a semantically rich, non-degenerate type system for the spatial data types.*

   We want a type system that corresponds to the users conceptual entities and therefore distinguishes between point, line, and region features in 2D space. In contrast, a collapsed type system would represent any spatial value by a single type "geometry", or a partially collapsed type system may have a type to represent point and line features together. An analogon for the better known standard data types would be to have a single data type "number" instead of distinguishing integers and reals. Here a tradeoff is involved. In a collapsed type system, of course, closure is much easier to achieve. This approach is, by the way, taken in constraint databases, where the uniform representation of a set of constraints corresponds to having a single type "geometry". On the other hand, the advantages of a non-collapsed type system are (i) better expressiveness; e.g., in a relation schema we can distinguish point or line features, (ii) better type checking in expressions (queries) – the classical reason for having expressive type systems, and (iii) more efficient data structures and algorithms. For example, geometric algorithms are almost always designed to deal with features of a certain kind (e.g. regions) rather than a mixed collection of anything. Knowledge of the type allows one to choose the most compact and efficient representations.

7. *For the spatial data types, predicates as well as certain operations with numeric results are needed.*

   It is obvious that predicates are indispensable in any spatial algebra. Operations with numeric results are, e.g., distance (*real*-valued), or the number of components (*int*-valued).

Figure 2 shows how (almost) the entire type system can be motivated through these principles, starting from the types of interest *mpoint* and *mregion*.

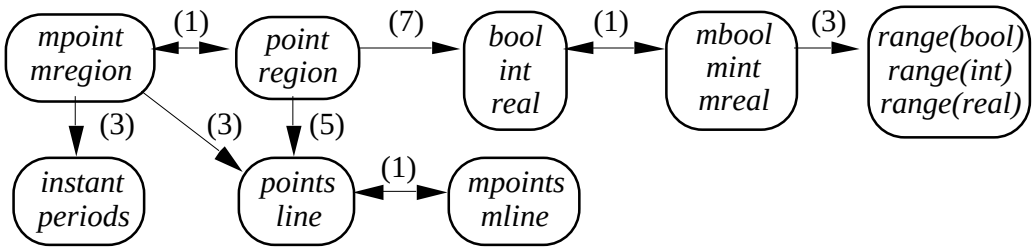

Figure 2: Type system motivated through *mpoint* and *mregion*

Here types *mpoint* and *mregion* are assumed as given. Types *point* and *region* follow by principle (1). Types *points* and *line* are needed for two reasons: (i) because by (5) we need types to represent the intersection of regions (which yields point, line and region features; we will introduce separate operations to return these in different types), and (ii) because according to (3) we need types for the projection of a moving point into the plane, which can consist of point and line features (point if the moving point is still for a while, or does not move at all). Types *mpoints* and *mline* follow by principle (1) again. Note that the application of (3) to *mpoints* and *mline* leads to types *points*, *line*, and *region* which are there already (projection of *mpoints* has *points* and *line* parts, of *mline* has *line* and *region* parts; again we will have separate operations to get these parts).

According to (7) we need types *bool*, *int*, and *real*, and then according to (1) also *mbool*, *mint*, and *mreal*. For these we need projection types according to (3) which induces the three *range* types. Finally, projection into the time domain (3) requires the types *instant* and *periods*.

What is missing in this picture are the *string* type and the class of *intime* types. The *string* type is indeed not needed as far as *mpoint* and *mregion* are concerned, since there are no operations on spatial data types or their induced types that yield *string* values. However, *string* is probably the most fundamental type of all in a DBMS. If we had no string attributes in tables, the system would be rather poor. So we assume this type is needed; then by principle (1) *mstring* should be there as well.

The *intime* types are in fact not as fundamental for this design as the rest. They are just handy to have for certain operations (e.g. finding the closest distance between two moving points including the time as well as the distance). Note however, that they do not violate the 7 principles, as *intime* values are just pairs of values of existing types, and we will provide operations to do projections into these two components. Hence especially principle (3) holds also for them.

A few issues remain to be discussed. Why are there separate *point* and *points* types? We have seen that starting from *mpoint*, *points* arise by closure. Could one entirely omit *point* and *mpoint* and just live with sets of (moving) points? First, this would violate principle (4) since we would have no type for single values in 2D. Second, *point* and *mpoint* are the main target of the whole design; these are the natural abstractions for the position of a (moving) entity, and we should not by closure arguments lose our main target!

Another option one could pursue is to merge *points* and *line* into a single data type containing point and line features. This means the projection of a moving point would not yield two results of different types (and so require two different operations in our design), but only of this mixed type. However, this would violate principle (6), as it would lead to a partially collapsed type system for spatial types. By closure, one would obtain the temporal version of this mixed type which is conceptually quite ugly.

Why is the undefined value $\perp$ included in some types, and why not in all? It was included to have closure for the operation **atinstant** (time-slice). The temporal values that we deal with are generally partial functions. We need to be able to ask for the value at any instant of time. The result value, even if the function happens to be undefined at this instant, should be usable within subexpressions of the query. For example, such an expression may be evaluated for each tuple of a relation, where sometimes the temporal attribute has a defined value at this instant, sometimes not. The undefined value is included in all types representing single values; it is not needed in the set types. This is because all set types have the empty set as a value. If we ask for the value of a set type at an instant when it was not defined, then we can return the empty set. Indeed, an alternative view of the "moving" types is to view them as complete functions that can assume the values $\perp$ and $\emptyset$. However, when we talk about the definition time (operation **deftime**), we refer to those times when the value is different from $\perp$ or $\emptyset$.

Finally, why is there the emphasis on finite collections in the definition of spatial data types even though we are dealing with infinite point sets anyway? This is in order to be realistic and implementable. As mentioned in the Introduction, this abstract design is intended as a basis for a discrete design where we have to give finite representations for all the types. For each component of a spatial type, e.g. a face of a region, one can easily come up with a finite representation (e.g. a polygon with holes). Such a polygon certainly represents an infinite number of points of the plane. However, it is crucial to keep the number of components finite, as each of them will be some component of a data structure in the end.

4 Operations

4.1 Overview

The design of the operations adheres to three principles, of which some aspects have already been touched in Section 3.3.

1. *Design operations as generic as possible.*
2. *Achieve consistency between operations on non-temporal and temporal types.*
3. *Capture the interesting phenomena.*

The first principle is crucial, as our type system is quite large. To avoid a proliferation of operations, it is mandatory to find a unifying view of collections of types, and hence to focus on properties shared by many types.

The basic approach to achieve generality is to relate each type to either a one-dimensional or a two-dimensional space and to consider all values either as 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 (structured into disjoint intervals). Similarly, *point* describes single elements of two-dimensional space, whereas *points*, *line*, and *region* describe (different kinds of) subsets of the two-dimensional space.

Second, in order to achieve consistency of operations on non-temporal and temporal types, we proceed in two steps. In the first, we define operations on non-temporal types. In a second step, we systematically extend operations defined in the first step to the temporal variants of the respective types. This is called *lifting*.

Third, in order 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, we also need to have operations based on order relationships, topology, metric spaces, etc. There is no clear recipe to achieve closure of "interesting phenomena"; nevertheless, that should not keep us from having concepts and operations available like distance, size of a region, relationships of boundaries, and the like.

Section 4 is structured as follows. Section 4.2 develops an algebra over non-temporal types, based on the generic point and point set (value vs. subset of space) view of these types. The classes of operations considered are:

1. *Predicates.* These are operations that return Boolean values.
2. *Set operations.* These are the basic set operations such as the union of sets.
3. *Aggregate operations.* These compute some single value from a set in the respective space.
4. *Numeric operations.* These are closely related to the aggregate functions and compute some numeric value from a set, e.g., the perimeter of a region value.
5. *Distance and direction operations.* This type of function allows for the computation of, e.g., the minimum distance between two sets of points.
6. *Specific base type operations.* Some operations on base types outside the generic view are needed.

Table 2 gives an overview, listing just the names of operations. The corresponding section (Section 4.2 in this case) explains these operations in detail and gives their signature. The semantics of all operations is defined formally in the Appendix.

Class	Operations
Predicates	**isempty** $=, \neq$, **intersects**, **inside** $<, \leq, \geq, >$, **before** **touches**, **attached**, **overlaps**, **on_border**, **in_interior**
Set Operations	**intersection**, **union**, **minus** **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**

Table 2: Classes of Operations on Non-Temporal Types

Also the *kernel algebra* is defined as the set of operations of this section, restricted to the types in $BASE \cup SPATIAL$.

Section 4.3 defines operations on temporal types. There we are interested in the following classes of operations, which are summarized in Table 3:

1. *Projection to domain and range.* For each temporal type, an operation to project into the time domain or the range is available.
2. *Interaction with values from domain and range.* This allows one, for example, to restrict a temporal value to certain times or certain range values.
3. *The* **when** *operation.* A powerful operation to evaluate a generic predicate on a temporal value.
4. *Lifting.* For all operations of the kernel algebra corresponding operations on temporal types are introduced.
5. *Operations related to rate of change.* This includes operations like **speed**, **derivative**.

Class	Operations
Projection to Domain/Range	**deftime**, **rangevalues**, **locations**, **trajectory** **routes**, **traversed**, **inst**, **val**
Interaction with Domain/Range	**atinstant**, **atperiods**, **initial**, **final**, **present** **at**, **atmin**, **atmax**, **passes**
When	**when**
Lifting	(all new operations inferred)
Rate of Change	**derivative**, **speed**, **turn**, **velocity**

Table 3: Classes of Operations on Temporal Types

Some operations are needed that are based on our data types, but require a manipulation of a set of objects in the database (e.g., a relation). Such operations are treated in Section 4.4 and shown in Table 4.

Class	Operations
Operations on Sets of Objects	**decompose**

Table 4: Operations on Sets of Database Objects

4.2 Operations on Non-Temporal Types

As motivated above we take the view that we are dealing with single values and sets of these values in one-dimensional and two-dimensional spaces. The types can then be classified according to Table 5.

	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*

Table 5: Classification of Non-Temporal Types

Table 5 shows that we are dealing with five different one-dimensional spaces called Integer, Boolean, etc. and one two-dimensional space called 2D. The two types belonging to space Integer, for example, are *int* and *range*(*int*). One-dimensional spaces are further classified as being discrete or continuous. The distinction between 1D and 2D spaces is relevant because only the 1D spaces have a 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 accordingly as point types or point set types.

Example 4.1 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)
```

□

4.2.1 Notations for Signatures

Let us briefly introduce notations for signatures that are partly based on Table 5.

In defining operation signatures and semantics (e.g., in Tables 6 and 8 and in the Appendix), π and σ are type variables, ranging over all point and all point set types of Table 5, respectively. If several type variables occur in a signature (e.g., for binary operations), then they are always assumed to range over types of the same space. Hence in a signature $\pi \times \sigma \to \alpha$ we can, for example, select the one-dimensional space Integer and instantiate π to *int* and σ to *range*(*int*). Or we can select the two-dimensional space 2D where we can instantiate π to *point* and σ to either *points*, *line*, or *region*.

A signature $\sigma_1 \times \sigma_2 \to \alpha$ means that the type variables σ_1 and σ_2 can be instantiated independently; nevertheless, they have to range over the same space. In contrast, a signature

$\sigma \times \sigma \to \alpha$ says that both arguments have to be of the same type. The notation $\alpha \otimes \beta \to \gamma$ is used if any order of the two argument types is valid, hence it is an abbreviation for signatures $\alpha \times \beta \to \gamma$ and $\beta \times \alpha \to \gamma$.

Some operations are restricted to certain classes of spaces. The classes of interest are:

1D Integer, Boolean, String, Real, Time

2D 2D

1Dcont Real, Time

1Dnum Integer, Real, Time

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 $\alpha \to \beta$ [1D] is valid for all one-dimensional spaces.

A single operation may have several functionalities (signatures), as shown in Table 6 for **isempty**. Sometimes for a generic operation there exist more appropriate names for arguments of more specific types. For example, there is a **size** operation for any point set type; however, for type *periods* it makes more sense to call this size **duration**. In such a 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 π type, and $U, V, \ldots$ generic sets of values (point sets) of a σ type. For clarity, single values and sets in one-dimensional spaces are denoted $x, y, \ldots$ and $X, Y, \ldots$, respectively.

The default syntax for using operations in queries is the prefix notation $op(arg_1, \ldots, arg_n)$. An exception are the comparison operators $=$, $<$, etc. and the Boolean operators **and** and **or**, for which it is customary to have infix notation. For two operators, **when** and **decompose**, a special syntax is defined explicitly below.

4.2.2 Predicates

To achieve some completeness, the design of predicates is based on the following strategy.

First, we consider unary and binary predicates. For the latter, 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. Order means total order here, which is available only in one-dimensional spaces. Topology means considering boundaries and interiors of point sets.

On this abstract level, there are not many unary predicates one can think of. For a single point, we can ask whether it is undefined, and for a point set, we can ask whether it is empty. The generic predicate **isempty** is used for this purpose.

Operation	Signature
isempty	$\pi \to bool$
	$\sigma \to bool$

Table 6: Unary Predicates

The design space for binary predicates according to the two "dimensions" mentioned above is shown in Table 7.

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

Table 7: Analysis of Binary Predicates

The idea of Table 7 is to systematically evaluate the possible interactions between single values and sets and, based on that, to introduce (names for) operations. For example, we find that a check whether boundaries intersect is important, and then introduce **touches** as a name for that. Note that operations in the middle column are available in one-dimensional (ordered) spaces *in addition* to those in the other columns.

This design should be complete with respect to the first and the last column, as all possible interactions between the involved points and point sets have been considered. In contrast, for the comparison of one-dimensional point sets we have just offered a single predicate, **before**. A deeper study of configurations of sets of intervals (e.g., temporal elements) is beyond the scope of this paper.

Nevertheless, for comparing single intervals, Allen's interval predicates [All83] provide a yardstick for the temporal aspects of expressiveness and completeness. It has been shown that all Allen's interval predicates can be expressed in terms of inequality predicates on the interval start and end points [AH85]. These can be extracted with the **min** and **max** functions introduced below, respectively. So, with the standard ordering predicates, the set of operations has the expressive power of Allen's interval predicates.

Apart from completeness, another important issue is redundancy. It is obvious that some operations in Table 7 can be expressed by others if we have Boolean operators available (which is the case, see Section 4.2.7). There are actually two somewhat conflicting goals in this design. One is to obtain a concise set of operations. In the extreme this would mean a minimal set of operators, as would be the goal of a more theoretical study. The other goal is language design; it should be easy to express natural concepts. This allows for some redundancy. For example, every programming language offers all the six comparison operators $=, \neq, <, \leq, \geq, >$ even though only two of them $(=, <)$ are strictly needed. We strive for a good balance between these goals.

As a result, we obtain the signature in Table 8.

We have not offered any predicates related to distance or direction (e.g. "north"). However, such predicates can be obtained via numeric evaluations (see Section 4.2.6).

Example 4.2 "What are the neighbor countries of Belgium?"

```
SELECT C.name
FROM country B, country C
WHERE B.name = "Belgium" and touches(B.area, C.area)
```

□

4.2.3 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.

Operation	Signature	
$=, \neq$	$\pi \times \pi$	$\rightarrow$ *bool*
	$\sigma_1 \times \sigma_2$	$\rightarrow$ *bool*
intersects	$\sigma_1 \times \sigma_2$	$\rightarrow$ *bool*
inside	$\sigma_1 \times \sigma_2$	$\rightarrow$ *bool*
	$\pi \times \sigma$	$\rightarrow$ *bool*
$<, \leq, \geq, >$	$\pi \times \pi$	$\rightarrow$ *bool* [1D]
before	$\sigma_1 \times \sigma_2$	$\rightarrow$ *bool* [1D]
	$\pi \otimes \sigma$	$\rightarrow$ *bool* [1D]
touches, **attached**, **overlaps**	$\sigma_1 \times \sigma_2$	$\rightarrow$ *bool*
on_border, **in_interior**	$\pi \times \sigma$	$\rightarrow$ *bool*

Table 8: Binary Predicates

Singleton sets or empty sets that result from this use are interpreted as point values. This is possible because all domains include the undefined value ($\perp$), whose meaning we identify with the empty set. Permitting set operations on point types is especially useful in the context of temporal types, as we shall see later.

There is no union operation on two single points, because the result could be two points, which cannot be represented as a value of point type.

Defining set operations on a combination of one- and two-dimensional point sets is more involved. This is because we are using arbitrary closed or open sets (see the definition of the *range* constructor) in the one-dimensional space, whereas only closed point sets (*points*, *line*, and *region*) exist in the two-dimensional case.

The restriction to closed point sets in the two-dimensional case is a natural and common one. Regions lacking part of their boundary (Figure 3) are rather strange entities. Similarly, regions lacking isolated interior points or curves appear unnatural. The same holds for curves lacking single interior points.

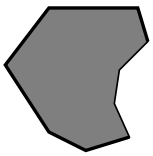

Figure 3: The Closure Operation ρ Completes an Open Set with the Points on the Boundary

Because our two-dimensional types are closed, it is necessary to apply a closure operation after applying the set operations on such entities. The closure operation adds all points on the boundary of an open set to make it closed (see Figure 3).

For example, if R_1 and R_2 are point sets of region values, then R_1 **minus** $R_2 \triangleq \rho(R_1 \setminus R_2)$. Hence any points on the boundary of R_2 subtracted from R_1 will again belong to R_1 **minus** R_2.

This seems to suggest that for uniformity, in the 1D space we should also work only with closed sets. However, there are also arguments to the contrary. Consider subsets of the domain of temporal types, the *periods* data type. For example, consider the simple stepwise constant function over time $u(t)$ of Figure 4. From time 1 to time 5, the value is 2, and from 5 to 9, it is 4. But what is the value at time 5? Since u is a *function* of time, there can only be a single value at time 5. Therefore, to be precise, we have to say that the value is 2 in the half-open time interval $[1, 5)$, and 4 in the closed interval $[5, 9]$. If we ask for the time period when the

value is 2, we get a half-open interval. Thus, as far as mappings are concerned, there is no way to avoid open intervals in their domains.

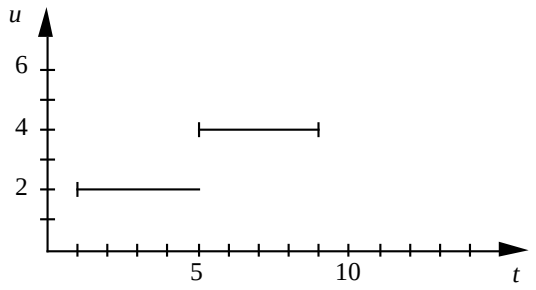

Figure 4: Stepwise Constant Functions in the Time Domain Require Open Boundaries

The same argument would force us to use open sets in 2D space if we considered mappings defined over 2D space. But they do not occur in our context. This justifies a different treatment of one- and two-dimensional point sets.

Whereas in all the one-dimensional spaces there is only a single point set type, in 2D space there are three. This requires an analysis of which argument type combinations make sense (return interesting results), and what the result types are.

Generally, if we apply set operations to values of different types, we get results that are a mixture of 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 [Til80]. For example, the regularized intersection removes all lower-dimensional pieces from the result of the corresponding intersection result. We will also adopt regularization in our framework as the semantics of the three "standard" set operations **union**, **minus**, **intersection** in 2D.

The behavior of the three set operations on different argument type combinations can be described as follows.

- Union of arguments of equal types has the usual semantics. For unions on different types, the argument values don't really merge; regularization throws away the lower-dimensional pieces and the result is the higher-dimensional argument. This result is not interesting as we know it already. Hence we will define union only for equal types.

- Difference always results in the type of the first argument. Closure has to be applied to the result. Among the argument combinations, only those return new results where the dimension of the second argument is equal or higher to that of the first. In the other cases, by closure, the first argument value is returned unchanged. We will allow difference on all type combinations even though some of them are not relevant.

- Intersection produces results of all dimensions smaller or equal to the dimension of the lowest-dimensional argument. For example, the intersection of two *region* values may yield a mixture of zero-, one-, and two-dimensional point sets, i.e., points, lines, and proper regions. The intersection of a *line* value with a *region* value may result in points and lines. We will define the **intersection** operator for all type combinations with regularized semantics, i.e., it returns the highest-dimensional result. To make the other kinds of results available, we introduce specialized operators. For example, **common_border** is introduced to return the line parts of region/region intersection, and **touch_points** returns resulting (isolated) points.

As a result we obtain the signature shown in Table 9.

Operation	Signature		
intersection, **minus**	$\pi \times \pi$	$\rightarrow \pi$	
intersection	$\pi \otimes \sigma$	$\rightarrow \pi$	
minus	$\pi \times \sigma$	$\rightarrow \pi$	
	$\sigma \times \pi$	$\rightarrow \sigma$	
union	$\pi \otimes \sigma$	$\rightarrow \sigma$	
intersection, **minus**, **union**	$\sigma \times \sigma$	$\rightarrow \sigma$	[1D]
intersection	$\sigma_1 \times \sigma_2$	$\rightarrow min(\sigma_1, \sigma_2)$	[2D]
minus	$\sigma_1 \times \sigma_2$	$\rightarrow \sigma_1$	[2D]
union	$\sigma \times \sigma$	$\rightarrow \sigma$	[2D]
crossings	*line* $\times$ *line*	$\rightarrow$ *points*	
touch_points	*region* $\otimes$ *line*	$\rightarrow$ *points*	
	region $\times$ *region*	$\rightarrow$ *points*	
common_border	*region* $\times$ *region*	$\rightarrow$ *line*	

Table 9: Set Operations

Here signatures are divided into five groups, the first two concerning point/point and point vs. point-set interaction. The last three groups deal with point-set/point-set interaction in one- and two-dimensional spaces; the last group introduces specialized intersection operations to obtain lower-dimensional results. The notation $min(\sigma_1, \sigma_2)$ refers to taking the minimum in an assumed "dimensional" order *points* < *line* < *region*.

The formal definition of the semantics of all set operations along the principles explained above can be found in the Appendix.

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

Example 4.3 "Determine the total land area of Europe."

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

This makes use of the facility for constructing aggregate functions described in Section 2. `TheEmptyRegion` is some empty region constant defined in the database. □

4.2.4 Aggregation

Aggregation reduces sets of points to points (Table 10).

Operation	Signature		
min, **max**	σ	$\rightarrow \pi$	[1D]
avg	σ	$\rightarrow \pi$	[1Dnum]
avg[**center**]	σ	$\rightarrow \pi$	[2D]
single	σ	$\rightarrow \pi$	

Table 10: Aggregate Operations

In one-dimensional space, where total orders are available, closed sets have minimum and maximum values, and functions (**min** and **max**) are provided that extract these. For open and

half-open intervals, we choose to let these functions return infimum and supremum values, i.e., the maximum and minimum of their closure. This is preferable over returning undefined values.

In all domains that have addition (e.g., 1Dnum), we can compute the average (**avg**). In 2D, the average is based 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. For example, some operations have to return set types although often the result is expected to be a single value. The operation **single** does this conversion.

Example 4.4 The query "Find the point where highway A1 crosses the river Rhine!" can be expressed as:

```
LET RhineA1 = ELEMENT(
  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)
```

The result can be used as a *point* value in further queries, whereas **crossings** returns a *points* value. □

4.2.5 Numeric Properties of Sets

For sets of points some well known numeric properties exist (Table 11).

Operation	Signature	
no_components	σ	$\rightarrow$ *int*
size	σ	$\rightarrow$ *real* [cont]
perimeter	*region*	$\rightarrow$ *real* [cont]
size[**duration**]	*periods*	$\rightarrow$ *real*
size[**length**]	*line*	$\rightarrow$ *real*
size[**area**]	*region*	$\rightarrow$ *real*

Table 11: Numeric Operations

For example, the number of 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 *range*(*real*), *periods*, *line*, and *region*). For 1D types, the size is the sum of the lengths of component intervals, for *line* it is the length, and for *region* it is the area. For the region type, we are additionally interested in the size of the boundary, called **perimeter**.

The second part of Table 11 introduces some alias names for the specific types.

Example 4.5 "List for each country its total size and the number of disjoint land areas."

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

□

Example 4.6 "How long is the common border of France and Germany?"

```
LET France = ELEMENT(SELECT area FROM country WHERE name = "France");
LET Germany = ELEMENT(SELECT area FROM country WHERE name = "Germany");
length(common_border(France, Germany))
```

□

4.2.6 Distance and Direction

A distance measure exists for all continuous types. The **distance** function determines the minimum distance between the closest pair of points where the first element is from the first argument and the second element is from the second argument. The distance between two points is the absolute value of the difference in one-dimensional space and the Euclidean distance in two-dimensional space. The time domain inherits arithmetics from the domain of real numbers, to which it is isomorphic.

Operation	Signature	
distance	$\pi \times \pi$	$\rightarrow$ *real* [cont]
	$\pi \otimes \sigma$	$\rightarrow$ *real* [cont]
	$\sigma \times \sigma$	$\rightarrow$ *real* [cont]
direction	*point* $\times$ *point*	$\rightarrow$ *real*

Table 12: Distance and Direction Operations

The direction between points is sometimes of interest. A **direction** function is thus included that 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 direction operation returns the undefined value $\perp$.

Example 4.7 "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
```

In this way we can express direction relationships such as north, south, etc. via numeric relationships. □

4.2.7 Specific Operations for Base Types

Some operations on base types are needed that are not related to the point/point set view. We mention them because they have to be included in the scope of operations to be lifted, i.e., the kernel algebra.

Operation	Signature	
and, **or**	*bool* $\times$ *bool*	$\rightarrow$ *bool*
not	*bool*	$\rightarrow$ *bool*

Table 13: Boolean Operations

4.2.8 Scope of the Kernel Algebra

The *kernel algebra* is defined to consist of the types in $BASE \cup SPATIAL$ together with all operations defined in Section 4.2, restricted to these types.

4.3 Operations on Temporal Types

Values of temporal types (i.e., types *moving*(α)) are partial functions of the form

$$f : A_{instant} \rightarrow \bar{A}_{\alpha}$$

In the following subsections we discuss operations for projection into domain and range, interaction with values from domain and range, the **when** operation, lifting, and operations related to rate of change.

4.3.1 Projection to Domain and Range

For values of all *moving* types – which are functions –, operations are provided that yield the domain and range of these functions. The domain function **deftime** returns the times for which a function is defined.

In 1D space, operation **rangevalues** 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 of lines; these can be obtained by operations **locations** and **trajectory** respectively.

For values of *intime* types, the two trivial projection operations **inst** and **val** are offered, yielding the two components.

Operation	Signature	
deftime	*moving*(α)	$\rightarrow$ *periods*
rangevalues	*moving*(α)	$\rightarrow$ *range*(α) [1D]
locations	*moving*(*point*)	$\rightarrow$ *points*
	moving(*points*)	$\rightarrow$ *points*
trajectory	*moving*(*point*)	$\rightarrow$ *line*
	moving(*points*)	$\rightarrow$ *line*
routes	*moving*(*line*)	$\rightarrow$ *line*
traversed	*moving*(*line*)	$\rightarrow$ *region*
	moving(*region*)	$\rightarrow$ *region*
inst	*intime*(α)	$\rightarrow$ *instant*
val	*intime*(α)	$\rightarrow \alpha$

Table 14: Operations for Projection of Temporal Values into Domain and Range

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 *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. That these projections can be represented as finite collections of intervals, faces, etc. and hence correspond to our data types is due to the continuity condition required for types *moving*(α) (see Section 3.2.4).

The design is complete in that all projection values in domain and range can be obtained. This was one of the major principles in the design of the type system, as discussed in Section 3.3.

Example 4.8 For illustration of operations on temporal types we use the example relations:

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

Attributes `airline` and `no` of the relation `flight` identify a flight. In addition, the relation records the names of the departure and destination cities and the route taken for each flight. The last attribute is of type *moving*(*point*). We have chosen to not record the scheduled departure and arrival times. The actual departure and arrival times may be derived from the route, as will be illustrated shortly. We assume that a flight's route is defined only for the times the plane is in flight and not when it is on the ground.

The relation `weather` records weather events such as high pressure areas, storms, or temperature maps. Some of these events are given names to identify them. The attribute `kind` gives the type of weather event, such as, "snow-cloud" or "tornado," and the `area` attribute provides the evolving extent of each weather event.

Relation `site` contains positions of certain well-known sites such as the Eiffel tower, Big Ben, etc. □

Example 4.9 With the operations of this subsection we can formulate queries:

"What distance traverses flight LH 257 over France?"

```
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 4.10 "At what time and distance passes flight 257 the Eiffel tower?"

We assume a **closest** operator exists with signature $mpoint \times point \rightarrow intime(point)$, which returns time and position when a moving point is closest to a given fixed point in the plane. We will later show how such an operator can be defined in terms of others.

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

□

4.3.2 Interaction With Points and Point Sets in Domain and Range

In this subsection we systematically study operations that relate the functional values of *moving* types with values either in their (time) domain or their range. For example, a moving point moves through the 2D plane; does it pass a given point or region in this plane? Does a moving real ever assume the given value 3.5? Besides comparison, one can also restrict the moving entity to the given domain or range values, e.g., get the part of the moving point when it was within the region, or determine the value of the moving real at time t or within time interval $[t_1, t_2]$.

In Table 15, the first group of operations concerns interaction with time domain values, the second interaction with range values. Operations **atinstant** and **atperiods** restrict a moving entity to a given instant, resulting in a pair (instant, value), or to a given set of time intervals, respectively. The **atinstant** operation is similar to the timeslice operator found in most temporal

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

Table 15: Interaction of Temporal Values With Values in Domain and Range

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**), this time to values in the range. For 1D space, restriction by either a point or a point-set value returns a value of the given moving type. For example, we can reduce 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 α and β with respect to the order *point* < *points* < *line* < *region*. For example, the restriction of a *moving*(*region*) by a *point* will result in a *moving*(*point*). This is analogous to the definition of result types for **intersection** in 2D in Section 4.2.3.

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** allows one to check whether the moving value ever assumed (one of) the value(s) given as a second argument.

All of these operations are of interest from a language design point of view. Some of them are derived, however, so they can be expressed by other operations in the design. For example, we have

```
present(f, t) = not(isempty(val(atinstant(f, t))))
```

Example 4.11 "When and where did flight 257 enter the territory of France?"

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

□

Example 4.12 "For which periods of time 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))
```

□

4.3.3 The Elusive when Operation

We now consider (speculate about) an extremely powerful yet conceptually quite simple operation called **when**, whose signature is shown in Table 16.

Operation	Signature	Syntax
when	$moving(\alpha) \times (\alpha \rightarrow bool) \quad \rightarrow moving(\alpha)$	$arg_1\ op[arg_2]$

Table 16: The **when** Operation

The idea is that we can restrict a time dependent value to the periods when its range value fulfils some property specified as a predicate. If we had such an operator, we could express a query such as "Restrict a moving region `mr` to the times when its area was greater 1000" as:

```
mr when[FUN (r:region) area(r) > 1000]
```

Here the result would be of type *mregion* again.

Whereas such an operation would be very powerful and desirable, it is questionable whether such a definition makes any sense. This is because the operator has to call for evaluation of the parameter predicate infinitely many times, since our moving entities are functions over a continuous domain. Looping over an infinite domain is inherently impossible. So for the moment this operation seems impossible to implement.

4.3.4 Lifting Operations to Time-Dependent Operations

Section 4.2 systematically defines operations on non-temporal types, the *kernel algebra*. This section uniformly lifts these operations to apply to the corresponding *moving* (temporal) types.

Consider an operation to be lifted. The idea is to allow any argument of the operation to be made temporal and to return a temporal type. More specifically, the lifted version of an operation with signature $\alpha_1 \times \ldots \times \alpha_k \rightarrow \beta$ has signatures

$$\alpha'_1 \times \ldots \times \alpha'_k \rightarrow moving(\beta)$$

with $\alpha'_i \in \{\alpha_i, moving(\alpha_i)\}$. So, each of the argument types may change into a time-dependent type which will transform the result type into a time-dependent type as well. The operations that result from lifting are given the same name as the operation they originate from. For example, the **intersection** operation with signature

$$region \times point \rightarrow point$$

is lifted to the signatures

$mregion \times point \rightarrow mpoint$,
$region \times mpoint \rightarrow mpoint$, and
$mregion \times mpoint \rightarrow mpoint$.

This lifting of operations generalizes existing operations that did not appear to be of great utility to operations that are quite useful. 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 4.2.3 took care to define the set operations for all argument types, including single points.

After lifting we find that some operations introduced earlier from a systematic point of view can now be expressed as derived operations. For example, the **at** operation can be explained in terms of lifted **intersection**:

```
at(mx, y) = intersection(mx, y)
```

In fact, the purpose of **at** is restriction, and it is not too surprising that this is related to intersection.

The fact that now all operations of the kernel algebra are available also as time-dependent operations results in a very powerful query language. Here are some examples.

Example 4.13 We can formulate pretty 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)))
```

Here predicate **on_border** yields a result of type *mbool*. Operation **at** reduces the definition time of this *mbool* to the times when it has value *TRUE*. – We can also check whether `mp` was *always* on the border of `r`:

```
min(rangevalues(on_border(mp, r)))
```

Note that this assumes an order $FALSE < TRUE$ on the domain *bool*. The **range** operator, applied to the *mbool* value resulting from the **on_border** operation yields a set of intervals $\{[TRUE, TRUE]\}$ iff **on_border** was true at all times. In this case, the whole expression evaluates to true. □

Example 4.14 "Determine the periods of time when snow storm 'Lizzy' consisted of exactly three separate areas."

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

Again, this works because 'Lizzy' is of type *mregion*, hence the lifted versions of **no_components** and of equality apply. □

Example 4.15 We are now able to define the **closest** operator of Example 4.10 within a query:

```
LET closest = FUN (mp:mpoint, p:point)
  atinstant(mp, inst(initial(atmin(distance(mp, p)))))
```

This depends on the lifted **distance** operator. We reduce the resulting *mreal* to the times when it is minimal, take the first such (instant, value) pair and then the instant from this pair. Finally, the original moving point is taken at this instant. □

Lifting is the key to achieving the goal of consistency and closure between non-temporal and temporal operations, as explained earlier in Section 3.3 (design principle 2).

4.3.5 The Elusive when Revisited

After lifting the operations of the kernel algebra, it turns out that we have another way of expressing the query of Section 4.3.3 "Restrict a moving region `mr` to the times when its size was greater 1000". Using **when** this was written:

```
mr when[FUN (r:region) area(r) > 1000]
```

Using the lifted versions of **area** and >, this is equivalent to:

```
atperiods(mr, deftime(at(area(mr) > 1000, TRUE)))
```

Why is it suddenly possible to realize the effect of the apparently unimplementable **when**? The reason is that we do not try to evaluate the parameter expression

```
area(r) > 1000
```

on infinitely many instances of parameter `r`, but instead evaluate its "lifted version"

```
area(mr) > 1000
```

on the original argument `mr` of **when**.

This is in fact a general technique for translating **when** queries. It is applicable for all parameter expressions of **when** that are formed using only operations of the kernel algebra. The translation is:

```
x when[FUN(y:α) p(y)] = atperiods(x, deftime(at(p(y)θ, TRUE)))
```

The substitution $\theta = \{y/x\}$, applied to `p(y)`, replaces each occurrence of `y` with the original moving object `x` (of type *moving*(α)). So, based on lifting and rewriting, we have in fact obtained an effective implementation of the **when** operator.

4.3.6 Rate of Change

An important property of any time-dependent value is its rate of change, i.e., its **derivative**. To determine to which of our data types this concept is applicable, consider the definition of the derivative, given next.

$$f'(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

This definition, and thus the notion of derivation, is applicable to any temporal type *moving*(α) with a range type α that (i) supports a difference operation, and (ii) supports division by a value of type *real*.

Type *real* clearly qualifies as a range type. For type *point*, at least three operations may assume the rule of difference in the definition, namely the Euclidean distance, the direction between two points, and the vector difference (viewing points as 2D vectors). This leads to three different derivative operations, which we call **speed**, **turn**, and **velocity**, respectively.

Operation	Signature
derivative	*mreal* → *mreal*
speed, **turn**	*mpoint* → *mreal*
velocity	*mpoint* → *mpoint*

Table 17: Derivative Operations

Note that one can get the acceleration of a moving point `mp` as a number by

```
derivative(speed(mp))
```

and as a vector, or moving point, by **velocity**(**velocity**(mp)).

The notion of derivation does not apply to the discrete data types *int*, *string*, and *bool* because there is no division available (for *string* and *bool* a difference operation is also absent).

An interesting question is whether one can define the derivative of a moving region. One possibility is to define difference on regions based on set difference of the underlying point sets. Since the definition of difference requires a neutral element and "negative values", one could view a region as a pair (R, S), consisting of a "positive" point set R, and a "negative" point set S. The empty region, playing the role of the neutral element, is represented as $(\emptyset, \emptyset)$, and the difference $(\emptyset, \emptyset) \setminus (Q, \emptyset)$ is $(\emptyset, Q)$. So far, so good. However, there seems to be no obvious definition of a division operation. Also, it is hard to imagine what the derivative of a moving region defined in this way means. Therefore we have not introduced a derivative on moving regions.

Example 4.16 Nevertheless, one can still observe, for example, the growth rate of a moving region: "At what time did snow storm Lizzy expand most?"

inst(**initial**(**atmax**(**derivative**(**area**(Lizzy))))) □

Example 4.17 "Show on a map the parts of the route of flight 257 when the plane's speed exceeds 800 kmh."

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

Of course, the background of the map still has to be produced by a different tool or query. □

4.4 Operations on Sets of Objects

All operations defined in Sections 4.2 and 4.3 apply to "atomic" data types only, i.e., attribute data types with respect to a DBMS data model. All data types of our design, as described in Section 3, and including the temporal ones, are atomic in this sense.

However, sometimes in the design of data types for new applications there are operations of interest that cannot be formulated in terms of the atomic data types alone, but need to manipulate a set of database objects (with attributes of the new data types) as a whole. A striking example in spatial databases is the computation of a Voronoi diagram. Given a set of points P in the plane, the Voronoi diagram is a partition of the plane into regions such that for each point $p \in P$ there is a region $V(p)$ which consists of all points of the plane that are closer to p than to any other point in P. Hence the region $V(p)$ describes some kind of "neighborhood" of p.

Although for each point a region is computed, it is clearly impossible to formulate this as an operation with signature *point* $\rightarrow$ *region*, since the region $V(p)$ depends not only on p but also on all or some of the other points in P. This operation can, however, be defined as a manipulation of a set of database objects with an attribute of type *point* (e.g., a relation with a *point* attribute). The result will be the same set of objects extended with an attribute of type *region*, containing the Voronoi region.

Operations on sets of objects are essentially motivated by the need to manipulate *sets of values of atomic data types*. However, each of these values is associated with (an attribute of) a database object, and it is important to maintain the connection between the value and the object it belongs to. For this reason it makes sense to formulate these operations as manipulations of sets of database objects.

Such data type related operations on sets of objects have been introduced earlier, for example, in the geo-relational algebra [Güt88], in [SV89], and in the ROSE algebra [GS95]. The geo-relational algebra has a Voronoi operator as described above.

In the design of this paper we need only a single set operator called **decompose** which has been defined similarly in the context of the ROSE algebra. This one, however, is crucial, as it allows us to access the internal structure of many of our data type values.

Operation	Signature	Syntax
decompose	$set(\omega_1) \times (\omega_1 \to \sigma) \times ident \to set(\omega_2)$ $set(\omega_1) \times (\omega_1 \to moving(\alpha)) \times ident \to set(\omega_2)$	$arg_1\ op[arg_2, arg_3]$

Table 18: Operations on Sets of Database Objects

The purpose of **decompose** is to make the components of values of point set types accessible within a query. "Components" refers to connected components; all our point set types are defined to have a structure that consists of a finite number of connected components. For any *range* type a component is a single interval, for the types *points*, *line*, and *region*, a component is a single point, a maximal connected subgraph, and a face, respectively.

Decomposition basically transforms a value of some point set type σ into a set of values of the same type σ such that each value in the result set contains a single component. For example, a value of the *periods* type consists of a number of intervals. Such a value is decomposed into a set of *periods* values, with each value being a single interval.

Similarly, **decompose** makes available the connected components of temporal data types. Here a component is a maximal continuous part of the function value.

As a manipulation of a set of database objects, this is treated as follows. The first argument of **decompose** is a set of database objects (e.g., a set of tuples in the relational model). The second argument is a function (e.g., an attribute name) that maps an object (e.g., a tuple) into a value of some point set type. The third argument is an identifier, used as a name for a new attribute. The result set of objects is produced as follows: For each object u with an attribute value that has k components, **decompose** returns k copies of u, each of which is extended by one of the k component values (under the new attribute).

Example 4.18 Consider the relation `country(name:`*string*`, area:`*region*`)` introduced earlier. The query

```
country decompose[area, part]
```

returns a relation with schema `(name:`*string*`, area:`*region*`, part:`*region*`)` □

In the signature of Table 18, ω_1 denotes an object (tuple) type and σ one of our point set types, hence $\omega_1 \to \sigma$ is an attribute of type σ. Similarly, $\omega_1 \to moving(\alpha)$ is an attribute of type $moving(\alpha)$. The resulting set of objects has a different object (tuple) type ω_2 due to the extension by the new attribute *ident* of type σ or $moving(\alpha)$, respectively.

These operations can be used as follows.

Example 4.19 "Determine the area of the smallest connected region of any country."

```
LET country2 = country decompose[region, part];
LET country3 = SELECT partsize AS area(part) FROM country2;
SELECT MIN(partsize) FROM country3
```

□

Example 4.20 Let us assume that flight 257 alternates between being over land areas of Europe and over sea. We would like to see a list of time periods, ordered by duration, when flight 257 was over land.

```
LET land257 =
  SET(route, at(route257, Europe)) decompose[route, piece];
SELECT start AS min(deftime(piece)), end AS max(deftime(piece)),
  duration AS duration(deftime(piece))
FROM land257
ORDER BY duration
```

Here the **at** operation restricts flight 257 to the parts above Europe (the area of which has been computed earlier, in Example 4.3). The SET constructor transforms this into a relation with one tuple and a single attribute `route` containing this value. To this relation, **decompose** is applied which puts each component of the moving point into a separate tuple. The relation `land257` created in this way is then processed in the next part of the query. □

5 Application Examples

To illustrate the query language resulting from our design, in this section we consider two rather different example applications. The first, related to multimedia presentations, has relatively simple spatio-temporal data that change only in discrete steps. The second, forest fire analysis, allows us to show some more advanced examples on moving objects, and moving regions in particular.

5.1 Multimedia Scenario

Multimedia presentations are good examples of spatio-temporal contexts. Here we have multimedia objects that are presented for some time occupying space on the screen (we assume that they are rectangles) and then they disappear. A crucial part of a multimedia scenario is the set of spatio-temporal relationships/constraints that define the spatial and/or temporal order of media object presentations [VTS98]. The ability to query the spatio-temporal configuration of a multimedia presentation would be an important aid to multimedia application designers.

A sample scenario for a news clip might be described as follows.

> The news clip starts with presentation of image A, located at point (50, 50) relative to the application origin. At the same time a background music E starts. 10 seconds later a video clip B starts. It appears to the right side (18cm) and below the upper side of A (12 cm). Just after the end of B, a video clip C starts that shows the highlights of a fashion show. It appears 7 cm below (and left aligned to) the position of B. 3 seconds after the start of C, a text logo D (e.g. the designer's logo) appears inside C, 8 cm above the bottom side of C, aligned to the right side. D will remain for 4 seconds on the screen. Meanwhile, at the 10th second of the news clip, the TV channel logo (F) appears at the bottom-left corner of the application window. F disappears after 3 seconds. The application ends when music background E ends.

The spatial and temporal configuration of the scenario is illustrated in Figure 5.

We assume that the following relational schema is used to store information about objects that participate in the presentation as moving regions:

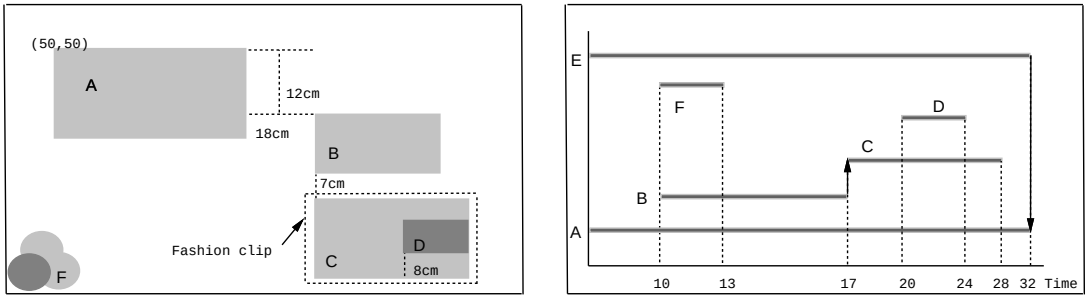

Figure 5: Spatial and Temporal Layout of a News Clip

`object(name:`*string*`, actor:`*mregion*`)`

In this case actors are boxes (trivial moving regions) that result from the presentation of an object for some time. We can then formulate the following queries.

Example 5.1 "What is the screen layout at the 5th second of the application?"

```
SELECT val(atinstant(actor, 5)) FROM object WHERE present(actor, 5)
```

□

Example 5.2 "What is the temporal layout of the application between the 10th and the 18th second of the application?"

```
SELECT name, intersection(deftime(actor), [10,18])
FROM object
WHERE intersects(deftime(actor), [10,18])
```

□

Example 5.3 "Which objects overlap spatially object A during its presentation?"

```
SELECT Y.name
FROM object X, object Y,
WHERE intersects(X.actor, Y.actor) and X.name = "A" and Y.name =/= "A"
```

□

Example 5.4 "Find the objects that spatially overlap B before its presentation."

```
SELECT X.name
FROM object X, object Y,
WHERE Y.name = "B" and before(deftime(X.actor), deftime(Y.actor))
  and intersects(traversed(X.actor), traversed(Y.actor))
```

□

5.2 Forest Fire Control Management

In a number of countries like the USA, Canada, and others, fire is one of the main agents of forest damage. Forest fires are often caused by the carelessness of people abandoning campfires in and around forests. Another essential reason is self-ignition through lightning strikes, long drought, or underground fire sources like coal seams. 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 like cover type, elevation, slope, distance to roads,

and distance to human settlements. Specialized geographical information systems enriched by a temporal component and by corresponding analysis tools could be appropriate systems to support these tasks.

In a very simplified manner this application 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(fightername:string, location:mpoint)
```

The relation `forest` records the location and the development of different forests growing and shrinking over time through clearing, cultivation, and destruction processes, for example. The relation `forest_fire` documents the evolution of different fires from their ignition up to their extinction. The relation `fire_fighter` describes the motion of fire fighters being on duty from their start at the fire station up to their return. The following sample queries illustrate enhanced spatio-temporal database functionality.

Example 5.5 "When and where did the fire called 'The Big Fire' reach what 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 maximum. The area operator is used in its lifted version. □

Example 5.6 "Determine the total size of the forest areas destroyed by the fire called 'The Big Fire'."

```
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
```

Here the **intersects** predicate of the join condition is a lifted predicate. 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 5.7 "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. These are eliminated in the second subquery. □

Example 5.8 "Which fire fighters had been enclosed by a fire but could always escape from the flames?"

```
LET times =
  SELECT fightername, maxtime_on_duty AS inst(final(location)),
    maxtime_in_fire AS inst(final(intersection(location, extent)))
  FROM forest_fire, fire_fighter
  WHERE ever(inside(location, extent));
LET times2 =
  SELECT fightername, maxtime_on_duty, last_in_fire AS MAX(maxtime_in_fire)
  FROM times
  GROUPBY fightername, maxtime_on_duty;
SELECT fightername
FROM times2
WHERE maxtime_on_duty > last_in_fire
```

The first subquery finds all fire fighter/forest fire pairs for which holds that the fire fighter was at some time inside the forest fire. We record his/her name, the time when he/she was on duty the last time, and the time when he/she left the fire. Since each fire_fighter may have been in several fires, and so have several values of `maxtime_in_fire`, in the next subquery we group by `fightername` and determine the last time the person was in any fire. Finally, the person survived, if he/she was still on duty after this time. □

Example 5.9 " How long was fire fighter Th. Miller enclosed by the fire called 'The Big Fire' and which distance did he 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 5.5, and that we know that Th. Miller was in this fire (otherwise time and distance will be returned as zero). □

Example 5.10 "How many times was each forest victim of fires?"

```
SELECT forestname, fireno AS COUNT(*)
FROM forest, forest_fire
WHERE ever(intersects(territory, extent))
GROUP BY forestname
```

□

Example 5.11 "Determine the times and locations when 'TheBigFire' started."

We assume that a fire can start at different times with different initial regions which may merge into one or even stay separate. The task is to determine these initial regions. This is a fairly complex problem and one may wonder whether it can be expressed with the given operations at all. We will show that it is possible.

The crucial point is that with **no_components** we have a tool to find the transitions when a new region (face) was added to the moving region describing the fire. We will find the times of these transitions and then go back to the moving region itself to determine the new face starting at this time.

```
LET number_history =
  SET(number, no_components(TheBigFire)) decompose[number, no];
LET history =
  SELECT period AS deftime(no), value AS single(no)
  FROM number_history;
LET pairs =
  SELECT interval1 AS X.period, interval2 AS Y.period
  FROM history X, history Y
  WHERE max(X.period) = min(Y.period) and X.value < Y.value;
SELECT starttime AS min(interval2),
  region AS minus(val(initial(atperiods(TheBigFire, interval2))),
    val(final(atperiods(TheBigFire, interval1))))
FROM pairs
```

In the first step, the lifted version of **no_components** produces a moving integer describing how many components 'TheBigFire' had at different times. We put this into a single attribute/single tuple relation and then apply **decompose**. For a moving integer each change of value produces another component, hence after **decompose** there is one tuple for each value with its associated time interval.

In the second step, relation `history` is computed which has for each of these components its time interval and value. In the third step, a self-join of `history` is performed to find pairs of adjacent time intervals where the number of components increased. In the final step, we compute the transition times (when the number of components increased) as well as the new fire regions. These can be obtained by subtracting the final region of the earlier time interval from the initial region of the later time interval.

Since this observes only changes in the number of components, but not yet the change from 0 to 1, we still have to get the very first time and region of the fire. However, these are very easy to determine by **inst(initial(TheBigFire))** and **val(initial(TheBigFire))**. □

Note that the capability of observing structural changes via **no_components**, as demonstrated in the previous example, is important for many applications. For example, one can find transitions when states merged or split (e.g. reunification), when disjoint parts of a highway network were connected, etc.

6 Related Work

The core of this paper's contribution is a framework of data types for capturing the time-varying spatial extents of moving objects. We cover in turn the relation to spatial and temporal databases, then consider a variety of related spatio-temporal proposals. Finally, attention is devoted to the relation to the data types available in object-relational database systems.

The traditional database management systems, offering a fixed set of types for use in columns of tables, are generally inadequate for managing spatial, let alone spatio-temporal, data. The restriction to the use of standard data types such as integers, reals, and strings in columns forces

a decomposition of spatial values into simple components, thus distributing the representation of even a single polygon over many rows. This renders even simple queries such as "find regions adjacent to a given region" difficult to formulate; and they are hopelessly inefficient to process because the decomposed spatial values must be reconstructed.

Observations such as these have led to an abstract data type view of spatial entities with suitable operations, used as attribute types in relational or other systems. Spatial types and operations have been used in many proposals for spatial query languages, e.g., [Güt88, SH91, Ege94]; and they have been implemented in prototype systems, e.g. [OM88, RFS88, Güt89]. Dedicated designs of spatial algebras with formal semantics are given in [SV89, GNT91, GS95].

Perhaps in part because of the pervasiveness of time and their simpler structures, time types are already supported by existing database systems, and the SQL standard offers types such as `DATE`, `TIME`, and `TIMESTAMP` [MS93]. In the research domain, semantic foundations for interpreting time values [Sno95, Ch. 5] and efficient formats [Sno95, Ch. 25] for storing time values have been proposed [DS94], as has extensible, multi-cultural support, including support for multiple languages, character sets, time zones, and calendars. Most proposals adopt bounded, discrete, and totally ordered types for the representation of time.

The temporal database community has also explored the use of temporal types, but mainly with a focus on temporal base types and at an abstract data-model level. Thus, a number of temporal data models (e.g., TERM [KL83], HRDM [Cli82, CC87], and Gadia's temporal data model [Gad88]) offer types that are functions from time to types corresponding to the base types in this paper. The related time-sequences data model [TCG+93, Ch. 11] allows attribute values that are basically sequences of time-value pairs.

Next, temporal data models may be generalized to be spatio-temporal. The idea is simple: Temporal data models provide built-in support for capturing one or more temporal aspects of the database entities. It is conceptually straightforward to also associate the database entities with spatial values. Concrete proposals include a variant of Gadia's temporal model [TCG+93, Ch. 2], a derivative of this model [CG94], and STSQL [BJS98]. Essentially, these proposals introduce functions from the product of time and space to base domains, and they provide languages for querying the resulting databases. These proposals are orthogonal to the specifics of types and simply abstractly assume types of arbitrary subsets of space and time; no frameworks of spatio-temporal types are defined. Over the past decade, Lorentzos has studied the inclusion of a generic interval column data type in multiple papers (see, e.g., [LM97] for further references). This type may be used for representing time intervals as well as lengths, widths, heights, etc.

From the other side, it is also possible to generalize spatial data models to become spatio-temporal. The data model by Worboys [Wor94] represents this approach. Here, spatial objects are associated with two temporal aspects, and a set of operators for querying is provided. However, this model does not provide an expressive type system, but basically offers only a single type, termed ST-complex, with a limited set of operations. In addition, two papers exist that consider spatio-temporal data as a sequence of spatial snapshots and in this context address implementation issues related to the representation of discrete changes of spatial regions over time [Käm94, RYG94].

Reference [SWCD97] presents a model for moving objects along with a query language. This model represents the positions of objects as continuous functions of time. However, the model captures just the *current* and anticipated, near future positions, in the form of motion vectors. The main issue addressed is how often motion vectors need to be updated to guarantee some bound on the error in predicted positions. This model does not describe complete trajectories of moving objects, as is done in this paper, nor does it offer a comprehensive set of types and operations. Moving regions are not addressed.

Reference [NNS97] proposes a model for moving multimedia objects. There, discrete snapshots of the trajectories of the objects composing a scene are captured using a graph notation. Objects are at the nodes, loops are labeled by trajectories, and non-loop edges are labeled by spatio-temporal relationships. A set of operations for the retrieval of moving objects is provided, and the model encompasses topological and directional relationships for space and time. In contrast to this paper's contribution, this quite different model is limited in its non-continuous representation of objects motion and its assumption that object shapes are non-changing.

Work in constraint databases is applicable to to spatio-temporal settings, as arbitrary shapes in multidimensional spaces can be described. Papers that explicitly address spatio-temporal examples and models include [GRS98, CR97]. However, this kind of work essentially assumes the single type "set of constraints," and is not concerned with types in the traditional sense. Operations for querying are basically those of relational algebra on infinite point sets. Recent work recognizes the need to include other operations, e.g., distance [GRS98].

The Informix Dynamic Server with Universal Data Option offers type extensibility [Inf97a]. So-called DataBlade modules may be used with the system, thus offering new types and associated functions that may be used in columns of database tables. Users may design their own DataBlades or may use some of the already available DataBlades. Of relevance for this paper, the Informix Geodetic DataBlade Module [Inf97b] offers types for time instants and intervals as well as spatial types for points, line segments, strings, rings, polygons, boxes, circles, ellipses, and coordinate pairs. An underlying ellipsoidal representation of the Earth is assumed for the spatial types; separate altitude range values may be associated with spatial values, to obtain three-dimensional support.

Informix does not offer any integrated spatio-temporal data types. Limited spatio-temporal data support may be obtained only by associating separate time and spatial values. The framework put forward in this paper provides a foundation allowing Informix or a third-party developer to develop a DataBlade that extends Informix with expressive and truly spatio-temporal data types.

Since 1996, the Oracle DBMS has offered a so-called spatial data option, also termed a Spatial Cartridge, that allows the user to better manage spatial data [Ora97]. Current support encompasses geometric forms such as points and point clusters, lines and line strings, and polygons and complex polygons with holes. However, this is just a layer on top of the DBMS (the so-called layered architecture, see, e.g., [Güt94]), hence there are no such data types in the DBMS itself. Future versions of Oracle will allow for the use of the geometric forms as types of columns of tables. However, neither spatio-temporal types nor layers are available in Oracle.

The support offered by Oracle resembles the support offered by DB2's Spatial Extender [Dav98], which offers spatial types such as point, line, and polygon, along with "multi-" versions of these, as well as associated functions, yielding several spatial ADT's. Unlike in Oracle, the Spatial Extender allows these types in columns of tables. But like Oracle, spatio-temporal types are absent.

7 Conclusions

The contribution of this paper is 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, whose flexibility, expressivity, and ease of use is so far unmatched in the literature. – Some unique aspects of our framework are the following:

- The emphasis on genericity, closure, and consistency. This is to some extent due to the fact that we had to deal with many more data types than any previous proposal (e.g., an algebra for just spatial types like [GS95]). Without this emphasis, such a design would become unmanageable.

- The abstract level of modeling. This design includes 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. Previous designs have been given in terms of polygons or simplicial complexes, for example.

- Continuous functions. This is also to our knowledge the first model that deals systematically and coherently with continuous functions as values of attribute data types.

- Lifting. The idea of defining a kernel algebra over non-temporal types that is then lifted uniformly to operations over temporal types seems to be a new and important concept to achieve consistency between non-temporal and temporal operations.

Complete, precise definitions of signatures for all operations and of the semantics of types and operations have been provided. The usability of the design as a query language has been demonstrated by example applications and queries. – Future work suggested by this paper includes:

- Checking the bounds of expressivity. On the one hand this could mean a comparison with theoretical models such as constraint databases. On the other hand one should consider example applications at more depth and see whether their needs are fulfilled by this design. Whereas we are convinced that this is a conceptually clean and very powerful core design, a study of applications might still reveal that certain kinds of operations are missing.

- Design a discrete model. As mentioned earlier, the abstract model of this paper has to be instantiated by selecting discrete representations. The issues arising at this step are discussed in some detail in [EGSV97].

- Given a discrete model, design appropriate data structures for the types and algorithms for the operations.

- Implement these data structures and algorithms in the form of a DBMS extension package for some extensible DBMS interface (e.g., as a data blade).

References

[AH85] J. F. Allen and J. Hayes. A Common-Sense Theory of Time. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985.

[All83] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 16(11):832–843, 1983.

[BJS98] 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*, pages 226–234, Atlanta, Georgia, February 1998.

[CC87] J. Clifford and A. Croker. The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, February 1987.

[CG94] T. S. Cheng and S. K. Gadia. A Pattern Matching Language for Spatio-Temporal Databases. In *Proceedings of the ACM Conference on Information and Knowledge Management*, pages 288–295, November 1994.

[Cli82] J. Clifford. A Model for Historical Databases. In *Proceedings of Workshop on Logical Bases for Data Bases*, December 1982.

[CR97] J. Chomicki and P. Revesz. Constraint-Based Interoperability of Spatio-Temporal Databases. In *Proceedings of the 5th International Symposium on Large Spatial Databases*, pages 142–161, Berlin, 1997.

[Dav98] J. R. Davis. IBM's DB2 Spatial Extender: Managing Geo-Spatial Information Within The DBMS. Technical report, IBM Corporation, May 1998.

[DS94] C. E. Dyreson and R. T. Snodgrass. Efficient Timestamp Input and Output. *Software – Practice and Experience*, 24(1):89–109, 1994.

[Ege94] M. Egenhofer. Spatial SQL: A Query and Presentation Language. *IEEE Transactions on Knowledge and Data Engineering*, 6:86–95, 1994.

[EGSV97] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. Technical Report Informatik 215, FernUniversität Hagen, 1997. Revised Version, May 1998. See also extended abstract [EGSV98].

[EGSV98] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Abstract and Discrete Modeling of Spatio-Temporal Data Types. In *Proceedings of the 6th ACM Symposium on Geographic information Systems*, Washington, D.C., November 1998.

[Gaa64] S. Gaal. *Point Set Topology.* Academic Press, 1964.

[Gad88] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Transactions on Database Systems*, 13(4):418–448, 1988.

[GNT91] M. Gargano, E. Nardelli, and M. Talamo. Abstract Data Types for the Logical Modeling of Complex Objects. *Information Systems*, 16(5), 1991.

[GRS98] S. Grumbach, P. Rigaux, and L. Segoufin. The Dedale System for Complex Spatial Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 213–224, 1998.

[GS95] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):100–143, 1995.

[Güt88] R. H. Güting. Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 506–527, Venice, March 1988.

[Güt89] R. H. Güting. Gral: An Extensible Relational Database System for Geometric Applications. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 33–44, Amsterdam, 1989.

[Güt93] 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*, pages 277–286, Washington, 1993.

[Güt94] R. H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3:357–399, 1994.

[Inf97a] *Extending Informix Universal Server: Data Types*. Informix Press, March 1997.

[Inf97b] *Informix Geodetic DataBlade Module: User's Guide*. Informix Press, June 1997.

[KL83] M. R. Klopprogge and P. C. Lockemann. Modelling Information Preserving Databases: Consequences of the Concept of Time. In *Proceedings of the International Conference on Very Large Data Bases*, pages 399–416, 1983.

[Käm94] T. Kämpke. Storing and Retrieving Changes in a Sequence of Polygons. *International Journal of Geographical Information Systems*, 8(6):493–513, 1994.

[LEW96] J. Loeckx, H. D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. John Wiley & Sons, Inc. and B.G. Teubner Publishers, 1996.

[LM97] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):480–499, 1997.

[MS93] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers, 1993.

[NNS97] M. Nabil, A. Ngu, and J. Shepherd. Modeling Moving Objects in Multimedia Database. In *Proceedings of the 5th Conference on Database Systems for Advanced Applications*, 1997.

[OM88] J. Orenstein and F. Manola. PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Transactions on Software Engineering*, 14:611–629, 1988.

[Ora97] Oracle8: Spatial Cartridge. An Oracle Technical White Paper, Oracle Corporation, June 1997.

[RFS88] N. Roussopoulos, C. Faloutsos, and T. Sellis. An Efficient Pictorial Database System for PSQL. *IEEE Transactions on Software Engineering*, 14:639–650, 1988.

[RYG94] H. Raafat, Z. Yang, and D. Gauthier. Relational Spatial Topologies for Historical Geographic Information. *International Journal of Geographical Information Systems*, 8(2):163–173, 1994.

[SH91] P. Svensson and Z. Huang. Geo-SAL: A Query Language for Spatial Data Analysis. In *Proceedings of the 2nd International Symposium on Large Spatial Databases*, pages 119–140, 1991.

[Sno95] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Boston, 1995.

[SV89] M. Scholl and A. Voisard. Thematic Map Modeling. In *First International Symposium on Spatial Databases (SSD'89)*, pages 167–190, 1989.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the International Conference on Data Engineering*, pages 422–432, 1997.

[TCG+93] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1993.

[Til80] R. B. Tilove. Set Membership Classification: A Unified Approach to Geometric Intersection Problems. *IEEE Transactions on Computers C-29*, pages 874–883, 1980.

[VTS98] M. Vazirgiannis, Y. Theodoridis, and T. Sellis. Spatio-Temporal Composition and Indexing for Large Multimedia Applications. *ACM/Springer-Verlag Multimedia Systems Journal*, 1998.

[Wor94] F. Worboys. A unified model for spatial and temporal information. *The Computer Journal*, 37(1):25–34, 1994.

A Definition of Continuity

We are interested in a generalized definition of continuity that is valid for all our temporal data types (i.e., types *moving*(α)) whereas the well-known classical definition refers only to real-valued functions. The definition should capture "discrete changes". A discrete change occurs when, for example, a new point appears in a *points* value, a curve in a *line* value suddenly turns by 90 degrees, or a *region* value suddenly ("from one instance to the next") is displaced to a new position. Intuitively, discontinuity means that the value changes in a single step without traversing all the intermediate stages.

We start by slightly modifying the basic definition of continuity. Since we are interested in temporal functions, the definition is given for them directly, rather than in more abstract terms.

Definition A.1 Let $f : \bar{A}_{instant} \to \bar{A}_\alpha$, and $t \in \bar{A}_{instant}$. f is ψ-continuous in t iff

$$\forall \gamma > 0 \; \exists \epsilon > 0 \text{ such that } \forall \delta < \epsilon : \psi(f(t \pm \delta), f(t)) < \gamma$$

where $\gamma, \epsilon, \delta \in \mathbb{R}$, and ψ is a function $\psi : \bar{A}_\alpha \times \bar{A}_\alpha \to \mathbb{R}$. □

Continuity is hence determined by the function ψ which expresses a measure of "dissimilarity" of its two arguments. It should be zero iff the two values are equal, and it should approach zero when the two values get more and more similar. The definition then says that for any chosen threshold γ, we can find an ϵ-environment of t where dissimilarity is bounded by γ.

Definition A.2 For any type α to which the *moving* type constructor is applicable, the dissimilarity function ψ is defined as follows:

$$\begin{aligned}
\alpha \in \{int, string, bool\}: \quad \psi(x,y) &= \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise} \end{cases} \\
\alpha = real: \quad \psi(x,y) &= |x - y| \\
\alpha = point: \quad \psi(p_1,p_2) &= d(p_1,p_2) \\
\alpha = points: \quad \psi(P_1,P_2) &= \sum_{p \in P_1} d(p,P_2) + \sum_{p \in P_2} d(p,P_1) \\
\alpha = line: \quad \psi(L_1,L_2) &= \sum_{c \in sc(L_1)} \int_0^1 d(c(u),L_2)\,du + \sum_{c \in sc(L_2)} \int_0^1 d(c(u),L_1)\,du \\
\alpha = region: \quad \psi(R_1,R_2) &= size(R_1 \setminus R_2) + size(R_2 \setminus R_1)
\end{aligned}$$

Here $d(p_1,p_2)$ denotes the Euclidean distance between two points, $d(p,P)$ the distance from p to the closest point in P. Similarly, for a line L, $d(p,L)$ denotes the distance from p to the closest point in L. For a line L, $sc(L)$ denotes the set of simple curves of which L is composed; this notation is defined in Appendix B. Finally, $size(R)$ denotes the area of a region R. □

This means that there are no continuous changes for the three discrete types; whenever the value changes, a discontinuity occurs. For points, dissimilarity is the sum over the distances from each point of one set to the closest point in the other set. For lines, the idea is the same; one just needs to integrate over the simple curves. For regions, dissimilarity is the area of the symmetric difference. Note that the definition fulfils the requirements stated for ψ above.

Based on this, the values of our types $moving(\alpha)$ can be partitioned along the time domain into maximal continuous pieces. For a value $\mu \in A_{moving(\alpha)}$ we denote by $\Gamma(\mu)$ its set of maximal continuous components.

B Definition of Semantics of Operations

For the definition of the semantics of operations we generally assume their strict evaluation, i.e., for each function operation *op* we demand:

$$f_{op}(\ldots, \perp, \ldots) = \perp$$

In the semantics definitions we use the following variable naming conventions: x, y (u, v) range over arbitrary values (2D-points), and X, Y (U, V) range over corresponding sets of values. b (B) ranges over values (sets of values) of base types, and predicates are denoted by p. We use μ to range over moving objects and t (T) to range over instant values (periods).

Predicates

Operation	Signature	Semantics
isempty	$\pi \rightarrow bool$	$u = \perp$
	$\sigma \rightarrow bool$	$U = \emptyset$

Table 19: Semantics of Unary Predicates

Operation	Signature		Semantics
$=, \neq$	$\pi \times \pi$	$\rightarrow bool$	$u = v, u \neq v$
	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$U = V, U \neq V$
intersects	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$U \cap V \neq \emptyset$
inside	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$U \subseteq V$
	$\pi \times \sigma$	$\rightarrow bool$	$u \in V$
$<, \leq, \geq, >$	$\pi \times \pi$	$\rightarrow bool$ [1D]	$x < y$ etc.
before	$\sigma_1 \times \sigma_2$	$\rightarrow bool$ [1D]	$\forall x \in X, \forall y \in Y : x \leq y$
	$\pi \times \sigma$	$\rightarrow bool$ [1D]	$\forall y \in Y : x \leq y$
	$\sigma \times \pi$	$\rightarrow bool$ [1D]	$\forall x \in X : x \leq y$
touches	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$\partial U \cap \partial V \neq \emptyset$
attached	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$\partial U \cap V^o \neq \emptyset$
overlaps	$\sigma_1 \times \sigma_2$	$\rightarrow bool$	$U^o \cap V^o \neq \emptyset$
on_border	$\pi \times \sigma$	$\rightarrow bool$	$u \in \partial U$
in_interior	$\pi \times \sigma$	$\rightarrow bool$	$u \in U^o$

Table 20: Semantics of Binary Predicates

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]	$X \cap Y, X \setminus Y, X \cup Y$
intersection	$\sigma_1 \times \sigma_2$	$\rightarrow min(\sigma_1, \sigma_2)$	[2D]	see Def. B.1
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	$line \times line$	$\rightarrow points$		see Def. B.1
touch_points	$region \otimes line$	$\rightarrow points$		
	$region \times region$	$\rightarrow points$		
common_border	$region \times region$	$\rightarrow line$		

Table 21: Semantics of Set Operations

Definition B.1 The semantics of intersection operations is defined as follows. Let P, L, and R, possibly indexed, denote arguments of type *points*, *line*, and *region*, respectively. Let Q be an argument of any of the three types. For commutative operations we give the definition only for one order of the arguments as it is identical for the other order. Definitions are ordered by argument combinations.

$f_{\textbf{intersection}}(P, Q) \triangleq P \cap Q$

$f_{\textbf{crossings}}(L_1, L_2) \triangleq \{p \in L_1 \cap L_2 \mid p \text{ is isolated in } L_1 \cap L_2\}$
$f_{\textbf{intersection}}(L_1, L_2) \triangleq (L_1 \cap L_2) \setminus f_{\textbf{crossings}}(L_1, L_2)$
$f_{\textbf{touch\_points}}(L, R) \triangleq \{p \in L \cap R \mid p \text{ is isolated in } L \cap R\}$
$f_{\textbf{intersection}}(L, R) \triangleq (L \cap R) \setminus f_{\textbf{touch\_points}}(L, R)$
$f_{\textbf{intersection}}(R_1, R_2) \triangleq \rho((R_1 \cap R_2)^o)$
$f_{\textbf{common\_border}}(R_1, R_2) \triangleq f_{\textbf{intersection}}(\partial R_1, \partial R_2)$
$f_{\textbf{touch\_points}}(R_1, R_2) \triangleq f_{\textbf{crossings}}(\partial R_1, \partial R_2)$ □

Recall that $\rho(Q), Q^o$, and ∂Q denote closure, interior, and boundary of Q, respectively.

Aggregation

Let $sc(U)$ denote the set of simple curves from which line U is built, that is,

$$sc(U) = C \in CC(S) : points(C) = U$$

We define the x- and y-projections of a curve: $c_x(u) = x$ and $c_y(u) = y$ iff $c(u) = (x, y)$. Then the length of a curve c is defined as:

$$\|c\| = \int_0^1 \sqrt{c'_x(u)^2 + c'_y(u)^2}\, du$$

where, e.g., c'_x is the derivative of c_x, that is, $\frac{d\, c_x(u)}{du}$. The length of a line U is given by the sum of lengths of its curves:

$$\|U\| = \sum_{c \in sc(U)} \|c\|$$

The average of a curve c is defined as a point vector:

$$\overrightarrow{c'} = \int_0^1 \overrightarrow{c(u)}\, du$$

Below, an unbound variable n always denotes the cardinality of the (finite) point set in scope.

Let S be an element of $range(\alpha)$ with $\alpha \in BASE \cup TIME$. We define a function *intvls* which maps a set S into its corresponding set of intervals. Let $intvls(S) = \{T_1, \ldots, T_n\}, n \in \mathbb{N}$, such that

1. $S = \bigcup_{i=1}^{n} T_i$
2. $\forall i \in \{1, \ldots, n\}\, \forall x, y \in T_i\, \forall z \in \bar{A}_\alpha : x < z < y \Rightarrow z \in T_i$
3. $\forall 1 \leq i < j \leq n : T_i$ and T_j are not adjacent

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

Table 22: Semantics of Aggregate Operations

Operation	Signature		Semantics
no_components	σ	$\rightarrow \int[1D]$	$\lvert intvls(U)\rvert$
no_components	*points*	$\rightarrow \int$	$\lvert U\rvert$
no_components	*line*	$\rightarrow \int$	$\lvert blocks(U)\rvert$ (see Def. B.2)
no_components	*region*	$\rightarrow \int$	$\lvert faces(U)\rvert$ (see Def. B.2)
perimeter	*region*	$\rightarrow$ *real*	$f_{\textbf{length}}(\partial U)$
size[**duration**]	*periods*	$\rightarrow$ *real*	$\sum_{T \in intvls(U)} \sup(T) - \inf(T)$
size[**length**]	*line*	$\rightarrow$ *real*	$\lVert U\rVert$
size[**area**]	*region*	$\rightarrow$ *real*	$\int_R dA$

Table 23: Semantics of Numeric Operations

Numeric Operations

Definition B.2 The definition of **no_components** for an argument L of type *line* is as follows. A non-empty set $A \subseteq L$ is called *connected* iff $\forall x, y \in A\ \exists c \in C : rng(c) \subseteq A$ and $c(0) = x$ and $c(1) = y$. A is called *maximally connected* iff A is connected and $\forall x \in A\, \forall y \in L \setminus A\, \nexists c \in C : rng(c) \subseteq A$ and $c(0) = x$ and $c(1) = y$.

Let $blocks(L) = \{L_1, \ldots, L_n\}$ such that

1. $L = \bigcup_{i=1}^{n} L_i$
2. $\forall 1 \leq i \leq n : L_i$ is maximally connected

We define: $f_{\textbf{no\_components}}(L) = \lvert blocks(L)\rvert = n$

For an argument R of type *region* the definition of **no_components** is as follows. Let $faces(R) = \{R_1, \ldots, R_n\}$ such that

1. $\forall 1 \leq i \leq n : R_i$ is regular closed
2. $R = \bigcup_{i=1}^{n} R_i$
3. $\forall 1 \leq i < j \leq n : R_i$ and R_j are quasi-disjoint
4. $\forall 1 \leq i \leq n\ \nexists X, Y \subseteq R : X$ and Y are quasi-disjoint, regular closed sets and $R_i = X \cup Y$

We define: $f_{\textbf{no\_components}}(R) = \lvert faces(R)\rvert = n$ □

Distance and Direction

Definition B.3 The definition of the **direction** operation is as follows. Let u and v be *point* values. Then

$$f_{\mathbf{direction}}(u,v) = \begin{cases} \perp & \text{if } u = v \\ \arctan \frac{v.y-u.y}{v.x-u.x} & \text{if } (u.x < v.x) \text{ and } (u.y \leq v.y) \\ 90 & \text{if } (u.x = v.x) \text{ and } (u.y < v.y) \\ 180 + \arctan \frac{v.y-u.y}{v.x-u.x} & \text{if } u.x > v.x \\ 270 & \text{if } (u.x = v.x) \text{ and } (u.y > v.y) \\ 360 + \arctan \frac{v.y-u.y}{v.x-u.x} & \text{if } (u.x < v.x) \text{ and } (u.y > v.y) \end{cases}$$

□

Operation	Signature		Semantics
distance	$real \times real$	$\rightarrow real$	$\lvert u - v \rvert$
	$instant \times instant$	$\rightarrow real$	$\lvert u - v \rvert$
	$point \times point$	$\rightarrow real$	$dist(u,v) = \sqrt{(u.x - v.x)^2 + (u.y - v.y)^2}$
	$\pi \otimes \sigma$	$\rightarrow real$ [1Dcont]	$\min\{\lvert u - v \rvert \mid v \in V\}$
	$\pi \otimes \sigma$	$\rightarrow real$ [2D]	$\min\{dist(u,v) \mid v \in V\}$
	$\sigma \times \sigma$	$\rightarrow real$ [1Dcont]	$\min\{\lvert u - v \rvert \mid u \in U, v \in V\}$
	$\sigma \times \sigma$	$\rightarrow real$ [2D]	$\min\{dist(u,v) \mid u \in U, v \in V\}$
direction	$point \times point$	$\rightarrow real$	see Def. B.3

Table 24: Distance and Direction Operations

Boolean Operations

The obvious definitions of boolean operators. Note that they are interpreted strictly.

Projections of Temporal Values

Operation	Signature		Semantics
deftime	$moving(\alpha)$	$\rightarrow periods$	$dom(\mu)$
rangevalues	$moving(\alpha)$	$\rightarrow range(\alpha)$ [1D]	$intvls(rng(\mu))$
locations	$moving(point)$	$\rightarrow points$	$isolated(rng(\mu))$
	$moving(points)$	$\rightarrow points$	$isolated(\bigcup rng(\mu))$
trajectory	$moving(point)$	$\rightarrow line$	$rng(\mu) \setminus f_{\mathbf{locations}}(\mu)$
	$moving(points)$	$\rightarrow line$	$\bigcup rng(\mu) \setminus f_{\mathbf{locations}}(\mu)$
traversed	$moving(line)$	$\rightarrow region$	$\rho((\bigcup rng(\mu))^\circ)$
	$moving(region)$	$\rightarrow region$	$\bigcup rng(\mu)$
routes	$moving(line)$	$\rightarrow line$	$\rho(\bigcup rng(\mu)) \setminus f_{\mathbf{traversed}}(\mu)$
inst	$intime(\alpha)$	$\rightarrow instant$	t where $u = (t,v)$
val	$intime(\alpha)$	$\rightarrow \alpha$	v where $u = (t,v)$

Table 25: Operations for Projection of Temporal Values into Domain and Range

Domain/Range Interactions

For a partial function $f : A \to B$ we write $f(x) = \perp$ whenever f is undefined for $x \in A$. To adjust the undefined value $\perp$ for values of type *points*, *line*, and *region* to $\emptyset$, we use the function:

$$x \uparrow \alpha = \begin{cases} \emptyset & \text{if } x = \perp \ \wedge \alpha \in \{points, line, region\} \\ x & \text{otherwise} \end{cases}$$

The *domain* of f is given by $dom(f) = \{x \in A \mid f(x) \neq \perp\}$. Similarly, the *range* of f is defined by $rng(f) = \{y \in B \mid \exists x \in A : f(x) = y\}$.

Operation	Signature			Semantics
atinstant	$moving(\alpha) \times instant$	$\to intime(\alpha)$		$(t, \mu(t) \uparrow \alpha)$
atperiods	$moving(\alpha) \times periods$	$\to moving(\alpha)$		$\{(t,y) \in \mu \mid \exists i \in T : t \in i\}$
initial	$moving(\alpha)$	$\to intime(\alpha)$		$\lim_{t \to \mathbf{inf}(dom(\mu))} \mu(t)$
final	$moving(\alpha)$	$\to intime(\alpha)$		$\lim_{t \to \mathbf{sup}(dom(\mu))} \mu(t)$
present	$moving(\alpha) \times instant$	$\to bool$		$\mu(t) \neq \perp$
present	$moving(\alpha) \times periods$	$\to bool$		$f_{\mathbf{atperiods}}(\mu, T) \neq \emptyset$
at	$moving(\alpha) \times \alpha$	$\to moving(\alpha)$	[1D]	$\{(t,y) \in \mu \mid y = b\}$
at	$moving(\alpha) \times range(\alpha)$	$\to moving(\alpha)$	[1D]	$\{(t,y) \in \mu \mid y \in B\}$
at	$moving(\alpha) \times point$	$\to mpoint$	[2D]	$\{(t,y) \in \mu \mid y = u\}$
at	$moving(\alpha) \times \beta$	$\to moving(min(\alpha,\beta))$	[2D]	$\{(t,y) \in \mu \mid y \in U\}$
atmin	$moving(\alpha)$	$\to moving(\alpha)$	[1D]	$\{(t,y) \in \mu \mid y = \min(rng(\mu))\}$
atmax	$moving(\alpha)$	$\to moving(\alpha)$	[1D]	$\{(t,y) \in \mu \mid y = \max(rng(\mu))\}$
passes	$moving(\alpha) \times \beta$	$\to bool$		$f_{\mathbf{at}}(\mu, x) \neq \emptyset$

Table 26: Semantics of Domain/Range Interactions

When

Operation	Signature	Semantics
when	$moving(\alpha) \times (\alpha \to bool) \to moving(\alpha)$	$\{(t,y) \in \mu \mid p(y)\}$

Table 27: Semantics of the **when** Operator

Lifting

An operation $op : \alpha_1 \times \ldots \times \alpha_k \to \beta$ can be lifted with respect to any combination of argument types. Such a combination can be conveniently described by a set of indices $L \subseteq \{1, \ldots, k\}$ for the lifted parameters, and we define:

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

Thus, the signature of any lifted version of op can be written as $op : \alpha_1^L \times \ldots \times \alpha_k^L \to moving(\beta)$. 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:

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

Now we can define the functions f_{op}^L pointwise by:

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

Derivative Operations

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

Table 28: Semantics of Derivative Operations

Decompose

Operation	Signature		Semantics
decompose	$set(\omega_1) \times (\omega_1 \rightarrow \sigma) \times ident$	$\rightarrow set(\omega_2)$	see Def. B.4
	$set(\omega_1) \times (\omega_1 \rightarrow moving(\alpha)) \times ident$	$\rightarrow set(\omega_2)$	see Def. B.4

Table 29: Operations on Sets of Database Objects

Definition B.4 Let $\Gamma(\mu)$ be the function which determines the maximal continuous components of a moving object μ. Let $\alpha \in BASE \cup TIME, \beta \in BASE \cup SPATIAL$, and $S \in range(\alpha) \cup moving(\alpha)$. Let $comp(S)$ compute the maximally connected components of S as follows:

$$comp(S) = \begin{cases} intvls(S) & \text{if } S \in range(\alpha) \\ S & \text{if } S \in points \\ blocks(S) & \text{if } S \in line \\ faces(S) & \text{if } S \in region \\ \Gamma(S) & \text{if } S \in moving(\beta) \end{cases}$$

Let $O = \{o_1, \ldots, o_n\}$ be a set of database objects (tuples in the relational model), and let $attr(o)$ be a function yielding a $range(\alpha)$ or a $moving(\beta)$ value as an attribute value of a database object $o \in O$. Moreover, we introduce an "object extension" function $\oplus$ which at the instance level adds a data type value to a database object. Hence, $o \oplus v$ is a database object o extended by a value v. At the type level, the given object type ω_1 is extended to an object type ω_2 by appending an attribute of type γ. That is ω_2 results from $\omega_1 \oplus (name, \gamma)$ where $name$ is the name of the new attribute of type $ident$ and $\gamma \in \{range(\alpha), moving(\beta)\}$. We are now able to give the semantics of the **decompose** operator.

$$f_{\mathbf{decompose}}(O, attr, name) = \{o \oplus v | o \in O, v \in comp(attr(o))\}$$

$$\tau_{\mathbf{decompose}}(set(\omega_1), \omega_1 \rightarrow \gamma, name) = set(\omega_1 \oplus (name, \gamma))$$

Here τ is a type mapping function defining the result type of a **decompose** application. □