A Data Model and Data Structures for Moving Objects Databases[*]

Luca Forlizzi[†], Ralf Hartmut Güting[‡],
Enrico Nardelli[†], and Markus Schneider[‡]

Abstract

We consider spatio-temporal databases supporting spatial objects with continuously changing position and extent, termed *moving objects databases*. We formally define a data model for such databases that includes complex evolving spatial structures such as line networks or multi-component regions with holes. The data model is given as a collection of data types and operations which can be plugged as attribute types into any DBMS data model (e.g. relational, or object-oriented) to obtain a complete model and query language. A particular novel concept is the *sliced representation* which represents a temporal development as a set of *units*, where unit types for spatial and other data types represent certain "simple" functions of time. We also show how the model can be mapped into concrete physical data structures in a DBMS environment.

1 Introduction

A wide and increasing range of database applications has to deal with spatial objects whose position and/or extent changes over time. This applies on the one hand to objects usually represented in maps such as countries, rivers, roads, pollution areas, land parcels and so forth. On the other hand it includes physical objects moving around such as taxis, air planes, oil tankers, criminals, polar bears, hurricanes, or flood areas, to name but a few examples. The management of the first class of objects is the more traditional task of spatio-temporal databases. The goal of our research is to support representation and querying not only of the first, but in particular of the more dynamic second class of objects; to emphasize this we speak of *moving objects databases*.

In previous work, we have proposed a *data type oriented* approach for modeling and querying such data [EGSV99, EGSV98]. The idea is to represent the temporal development of spatial entities in certain data types such as *moving point* or *moving region*. Values of such types are functions that associate with each instant in time a point or a region value. Suitable operations are provided on these types to support querying. Such data types can be embedded as attribute types into object-relational or other data models; they can be implemented and provided as extension packages (e.g. data blades) for suitable extensible DBMS environments.

[*]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.

[†]Dipartimento di Matematica Pura ed Applicata, Universita Degli Studi di L'Aquila, L'Aquila, Italy, {forlizzi, nardelli}@univaq.it

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

Following this approach, two questions arise. First, exactly which types and operations should be offered? Second, at what level of abstraction should these types and operations be described?

By "level of abstraction" we mean the following. A moving point can be defined either as a continuous function from time into the 2D plane, or as a polyline in the three-dimensional (2D + time) space. A region can be defined as a connected subset of the plane with non-empty interior, or as a polygon with polygonal holes. The essential difference is that in the first case we define the domains of data types just in terms of infinite sets, whereas in the second case we describe certain finite representations for the types.

In [EGSV98] we have discussed the issue at some depth and introduced the terms *abstract model* for the first and *discrete model* for the second level of abstraction. Both levels have their respective advantages. An abstract model is relatively clean and simple; it allows one to focus on the essential concepts and not get bogged down by representation details. However, it has no straightforward implementation. A discrete model fixes representations and is generally far more complex. It makes particular choices and thereby restricts the range of values of the abstract model that can be represented. For example, a moving point could be represented not only by a 3D polyline but also by higher order polynomial splines. Both cases (and many more) are included within the abstract model. On the other hand, once such a finite representation has been selected, it can be translated directly to data structures.

In [EGSV98] we came to the conclusion that both levels of modeling are needed and that one should first design an abstract model of spatio-temporal data types and then continue by defining a corresponding discrete model. Such an abstract model has been developed in [GBE$^+$98]. The main concerns in that design have been orthogonality in the type system, genericity and consistency of operations, and closure and consistency between structure and operations of related non-temporal and temporal types. Semantics of all types and operations have been defined formally.

The purpose of this paper is to continue this work by defining a discrete data model implementing the abstract model of [GBE$^+$98]. This means that for all data types of the abstract model we introduce corresponding "discrete" types whose domains are defined in terms of finite representations. We define precisely which constraints apply so that a finite representation does indeed describe a value of the abstract model. For example, a region will be described by a set of line segments, but not every set of line segments describes a valid region value.

The discrete model is a high-level specification of data structures for a spatio-temporal DBMS. In the last part of the paper we show how the discrete model can be mapped to real data structures that can be used to implement attribute data types in a DBMS. Hence the paper offers a good basis for the implementation of a "moving objects data blade."

Earlier work on spatio-temporal databases has generally been restricted to accommodate discrete changes of spatial values. Worboys [Wor94] has proposed such a model which represents spatio-temporal entities as the cross-product of a spatial and a temporal description, using simplicial complexes for the spatial part and sets of rectangles (for two time dimensions) for the temporal part. Other such models are [CG94] or [PD95]. Some papers in the GIS literature, e.g. [Käm94], study implementation issues such as efficient storage schemes for sequences of region snapshots.

More recently, research has addressed the more dynamic applications that we (and others) call "moving objects databases". Wolfson and colleagues [Wol98, WCD$^+$98] consider the management of collections of moving points in the plane. However, their model describes only the current and the expected position of a point in the near future, as represented by a motion vector. The main issue is to determine how often updates of motion vectors are needed to balance the cost of updates against imprecision in the knowledge of positions. Their model does not describe complete trajectories of moving objects, and it also does not address more complex spatial structures such as regions. Chomicki and Revesz [CR99] study a framework where spatio-temporal objects can be described as collections of *atomic geometric objects*, and each such atomic object is essentially given as a spatial object of some dimension d together with a continuous function describing the development of the spatial object over time. For the continuous functions, affine mappings (allowing translation, rotation, and scaling) and subclasses thereof are considered. They establish some basic results, e.g., rectangles with linear translation and scaling are closed under set operations whereas polygons with linear translation and scaling are only closed under union.

The CHOROCHRONOS project, in which we participate, has addressed some issues related to moving objects databases. Conceptual modeling is discussed in [TH97], indexing in [TSPM98]. Reference [PJ99] addresses the uncertainty in capturing moving point trajectories.

The constraint database approach can also be used to describe spatial as well as spatio-temporal data. Papers that explicitly address spatio-temporal examples and models are [GRS98, CR97].

However, except for [GBE$^+$98] to our knowledge there does not exist in the literature a comprehensive design of spatio-temporal types and operations, let alone a corresponding discrete data model as it is given in this paper. Our own earlier work [EGSV99, EGSV98] discusses the idea and some basic issues related to spatio-temporal data types, but does not yet define a discrete data model.

The paper is structured as follows. In Section 2 the abstract model as the basis for our design is briefly reviewed. Section 3 defines the discrete data types, first for non-temporal, and then for temporal types. Section 4 describes data structures for the discrete types. Two example algorithms illustrating the use of the model and the data structures are given in Section 5. Section 6 offers conclusions.

2 Review of the Abstract Model

The abstract model of [GBE$^+$98] offers the data types, or actually the *type system* shown in Table 1.

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

Table 1: Signature describing the abstract type system

The type system is described by a signature. A signature in general has *sorts* and *operators* and defines a set of terms. In this case the sorts are called *kinds* and the operators are *type constructors*.[1] The terms generated by the signature are the available

[1] We write signatures by giving first the argument and result sorts, and then the operators with this

data types. Some data types defined by this signature are _*int*_, _*region*_, _*range*_(_*instant*_), or _*moving*_(_*point*_).

The meaning of the data types, informally, is the following. The constant types _*int*_, _*real*_, _*string*_, _*bool*_ are as usual, except that the domains are extended by a special value "undefined". A value of type _*point*_ is a point in the real (2D) plane, a _*points*_ value a finite set of points. A _*line*_ value is a finite set of continuous curves in the plane. A _*region*_ value is a finite set of disjoint *faces* where each face is a connected subset of the plane with non-empty interior. Faces may have holes and lie within holes of other faces. Types _*line*_ and _*region*_ are illustrated in Figures 2 and 3, respectively.

Type _*instant*_ offers a time domain isomorphic to the real numbers. The _*range*_ type constructor produces types whose values are finite sets of pairwise disjoint intervals over the argument domain. The _*intime*_ constructor yields types associating a time instant with a value of the argument domain.

The most important type constructor is _*moving*_. Given an argument type α in BASE or SPATIAL, it constructs a type whose values are functions from time (the domain of _*instant*_) into the domain of α. Functions may be partial and must consist of only a finite number of continuous components (which is made precise in [GBE$^+$98]). For example, a _*moving*_(_*region*_) value is a function from time into _*region*_ values.

Over the types so defined, the abstract model offers a large set of operations. It defines first generic operations over the non-temporal types (all types except those constructed by _*moving*_ or _*intime*_). These operations include predicates (e.g. **inside** or $\leq$), set operations (e.g. **union**), aggregate operations, operations with numeric result (e.g. **size** of a region), and distance and direction operations.

In a second step, by a mechanism called temporal *lifting*, all operations defined in the first step over non-temporal types are uniformly and consistently made applicable to the corresponding temporal ("moving") types. For example, the operation **inside**, applicable e.g. to a _*point*_ and a _*region*_ argument and returning _*bool*_, is by lifting also applicable to a _*moving*_(_*point*_) vs. a _*region*_, or a _*point*_ vs. a _*moving*_(_*region*_), or a _*moving*_(_*point*_) vs. a _*moving*_(_*region*_); in all these cases it returns a _*moving*_(_*bool*_).

Third, special operations are offered for temporal types _*moving*_(α) whose values are functions. They can all be projected into domain (time) and range. Their intersection with values or sets of values from domain or range can be formed (e.g. **atinstant** restricts the function to a certain time instant). The rate of change (**derivative**, **speed**) can also be observed.

An example now shall briefly demonstrate how these data types can be embedded into any DBMS data model as attribute types and how pertaining operations can be used in queries. For example, we can integrate them into the relational model and have a relation

```
planes (airline: string, id: string, flight: mpoint)
```

where _*mpoint*_ is used as a synonym for _*moving*_(_*point*_) and included into the relation schema as an *abstract data type*. The term `flight` denotes a spatio-temporal attribute whose values record the locations of planes over time.

For posing queries we introduce the signatures of some operations. We only formulate special instances of them as far as they are needed for our examples. Corresponding generic signature specifications can be found in [GBE$^+$98].

functionality. As a convention, kinds are denoted by capitals and type constructors in italic underlined. Operations on data types are written in bold face.

Operation	Signature	
trajectory	$\underline{moving}(\underline{point})$	$\rightarrow \underline{line}$
length	$\underline{line}$	$\rightarrow \underline{real}$
distance	$\underline{moving}(\underline{point}) \times \underline{moving}(\underline{point})$	$\rightarrow \underline{moving}(\underline{real})$
atmin	$\underline{moving}(\underline{real})$	$\rightarrow \underline{moving}(\underline{real})$
initial	$\underline{moving}(\underline{real})$	$\rightarrow \underline{intime}(\underline{real})$
val	$\underline{intime}(\underline{real})$	$\rightarrow \underline{real}$

The projection of moving points into the plane may consist of points and lines. The operation **trajectory** computes the line parts of such a projection. The operation **length** determines the length of a $\underline{line}$ value. The distance between two moving points is calculated by **distance**. Operation **atmin** here restricts a moving real to all times with the same minimal $\underline{real}$ value. The first ($\underline{instant}$, $\underline{real}$) pair of a moving real is returned by the operation **initial**. Operation **val** is here applied to a ($\underline{instant}$, $\underline{real}$) pair and projects onto the second component.

We can now ask a query "Give me all flights of Lufthansa longer than 5000 kms":

```
SELECT airline, id
FROM planes
WHERE airline = ''Lufthansa'' AND length(trajectory(flight)) > 5000
```

This query just employs projection into space. An example of a genuine spatio-temporal query, which cannot be answered with the aid of projections, is: "Find all pairs of planes that during their flight came closer to each other than 500 meters!":

```
SELECT p.airline, p.id, q.airline, q.id
FROM planes p, planes q
WHERE val(initial(atmin(distance(p.flight, q.flight)))) < 0.5
```

This query represents an instance of a *spatio-temporal join*. Note that the **distance** operation is here used in its temporally lifted version.

Many further illustrating query examples from different application scenarios (e.g., multimedia presentations, forest fire control management) can be found in [GBE$^+$98]. These applications demonstrate that a very flexible and powerful query language results from this design.

In the following development of a discrete model, we focus on defining finitely and efficiently representable domains for the data types. Of course, the discrete model also includes operations. Almost all operations of the abstract model will also be available in the discrete model.[2] Of course, the next step is to develop algorithms for implementing these operations on the discrete representations. This is, however, beyond the scope of this paper, except for two relatively simple example algorithms in Section 5.

3 Data Types

3.1 Overview

In Section 3 we define data types that can represent values of corresponding types of the abstract model. Of course, the discrete types can in general only represent a subset

[2] A few operations, especially **derivative**, cannot be transferred, as they are not closed in the chosen discrete representation.

of the values of the corresponding abstract type.

All type constructors of the abstract model will have direct counterparts in the discrete model except for the *moving* constructor. This is, because it is impossible to introduce at the discrete level a type constructor that automatically transforms types into corresponding temporal types. The type system for the discrete model therefore looks quite the same as the abstract type system up to the *intime* constructor, but then introduces a number of new type constructors to implement the *moving* constructor, as shown in Table 2.

	$\to$ BASE	*int*, *real*, *string*, *bool*
	$\to$ SPATIAL	*point*, *points*, *line*, *region*
	$\to$ TIME	*instant*
BASE $\cup$ TIME	$\to$ RANGE	*range*
BASE $\cup$ SPATIAL	$\to$ TEMPORAL	*intime*
BASE $\cup$ SPATIAL	$\to$ UNIT	*const*
	$\to$ UNIT	*ureal*, *upoint*, *upoints*, *uline*, *uregion*
UNIT	$\to$ MAPPING	*mapping*

Table 2: Signature describing the discrete type system

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

The interesting part of the model is how temporal ("moving") types are represented. In this paper we describe the *sliced representation*. The basic idea is to decompose the temporal development of a value into fragments called "slices" such that within the slice this development can be described by some kind of "simple" function. This is illustrated in Figure 1.

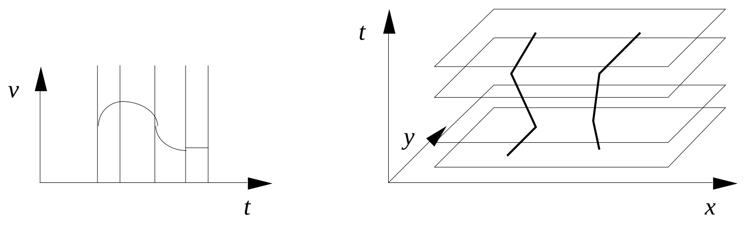

Figure 1: Sliced representation of moving *real* and moving *points* value

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

In summary, we obtain the correspondence between abstract and discrete temporal types shown in Table 3.

Abstract Type	Discrete Type
$\underline{moving}(\underline{int})$	$\underline{mapping}(\underline{const}(\underline{int}))$
$\underline{moving}(\underline{string})$	$\underline{mapping}(\underline{const}(\underline{string}))$
$\underline{moving}(\underline{bool})$	$\underline{mapping}(\underline{const}(\underline{bool}))$
$\underline{moving}(\underline{real})$	$\underline{mapping}(\underline{ureal})$
$\underline{moving}(\underline{point})$	$\underline{mapping}(\underline{upoint})$
$\underline{moving}(\underline{points})$	$\underline{mapping}(\underline{upoints})$
$\underline{moving}(\underline{line})$	$\underline{mapping}(\underline{uline})$
$\underline{moving}(\underline{region})$	$\underline{mapping}(\underline{uregion})$

Table 3: Correspondence between abstract and discrete temporal types

In Table 3 we have omitted the representations $\underline{mapping}(\underline{const}(\underline{real}))$, etc. which can be used to represent discretely changing real values and so forth, but are not so interesting for us.

In the remainder of Section 3 we formally define the data types of the discrete model. That means, for each type we define its *domain* of values in terms of some finite representation. From an algebraic point of view, we define for each *sort* (type) a *carrier set*. For a type α we denote its carrier set as D_α.

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

The following Section 3.2 contains definitions for all non-temporal types and for the temporal types in the sliced representation. For the spatial temporal data types $\underline{moving}(\underline{points})$, $\underline{moving}(\underline{line})$, and $\underline{moving}(\underline{region})$ one can also define direct three-dimensional representations in terms of polyhedra etc.; these representations will be treated elsewhere.

3.2 Definition of Discrete Data Types

3.2.1 Base Types and Time Type

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

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

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

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

3.2.2 Spatial Data Types

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

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

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

A value of type *<u>points</u>* is simply a set of points.

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

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

The definition of discrete representations for the types *<u>line</u>* and *<u>region</u>* is based on linear approximations. A value of type *<u>line</u>* is essentially just a finite set of line segments in the plane. Figure 2 shows the correspondence between the abstract type

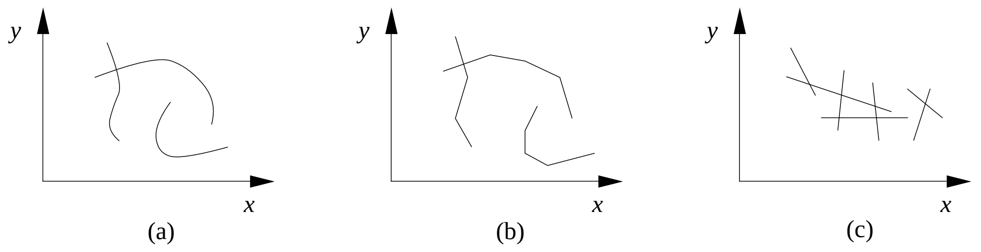

Figure 2: (a) *<u>line</u>* value of the abstract model (b) *<u>line</u>* value of the discrete model (c) any set of line segments is also a *<u>line</u>* value

for *<u>line</u>* and the discrete type. The abstract type is a set of curves in the plane which was viewed in [GBE+98] as a planar graph whose nodes are intersections of curves and whose edges are intersection-free pieces of curves. The discrete *<u>line</u>* type represents curves by polylines. However, one can assume a less structured view and consider the same shape to be just a collection of line segments. At the same time, any collection of line segments in the plane defines a valid collection of curves (or planar graph) of the abstract model (see Figure 2 (c)). Hence, modeling *<u>line</u>* as a set of line segments is no less expressive than the polyline view. It has the advantage that computing the projection of a (discrete representation) moving point into the plane can be done very efficiently as it is not necessary to compute the polyline or graph structure. Hence we prefer to use this unstructured view. Let

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

be the set of all line segments.

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

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

A <u>*region*</u> value at the discrete level is essentially a collection of polygons with polygonal holes (Figure 3). Formal definitions are based on the notions of *cycles* and

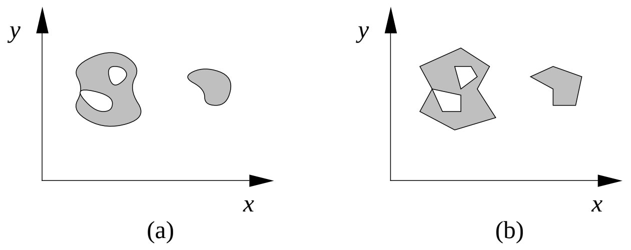

Figure 3: (a) <u>*region*</u> value of the abstract model (b) <u>*region*</u> value of the discrete model

faces. These definitions are similar to those of the ROSE algebra [GS95]. We need to reconsider such definitions here for two reasons: (i) They have to be modified a bit because here we have no "realm-based" [GS95] environment any more, and (ii) we are going to extend them to the "moving" case in the following sections.

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

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

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

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{Face = \{(c, H) \mid c \in Cycle, H \subset Cycle, \text{ such that}} \\
(i) & \forall h \in H : edge\text{-}inside(h, c) \\
(ii) & \forall h_1, h_2 \in H : h_1 \neq h_2 \Rightarrow edge\text{-}disjoint(h_1, h_2) \\
(iii) & \text{any cycle that can be formed from the segments of } c \text{ or } H \\
 & \text{is either } c \text{ or one of the cycles of } H
\end{array}
$$

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

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

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

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

The semantics of a region value should be clear: A cycle c represents all points of the plane enclosed by it as well as the points on the boundary. Given $\sigma(c)$, we have for a face $\sigma((c, H)) = \mathit{closure}(\sigma(c) \setminus \bigcup_{h \in H} \sigma(h))$, that is, hole areas are subtracted from the outer cycle area, but then the resulting point set is closed again in the abstract domain. The area of a region is then obviously the union of the area of its faces.

3.2.3 Sets of Intervals

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

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

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

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

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

where the two sets LC and RC are defined as

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

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

Whether two intervals $u = (s_u, e_u, lc_u, rc_u)$ and $v = (s_v, e_v, lc_v, rc_v) \in \mathit{Interval}(S)$

are *disjoint* or *adjacent* is defined as follows:

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

The last condition for *r-adjacent* is important for discrete domains such as <u>*int*</u>. Representations of finite sets of intervals over S can now be defined as

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

The conditions ensure that a set of intervals has a unique and minimal representation. The <u>*range*</u> type constructor can then be defined as:

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

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

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

3.2.4 Sliced Representation for Moving Objects

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

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

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

$$\textit{Unit}(S) = \textit{Interval}(\textit{Instant}) \times S$$

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

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

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

The $\underline{mapping}$ type constructor is defined for any type $\alpha \in \mathit{UNIT}$ as:

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

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

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

$$\iota_\alpha : S_\alpha \times \mathit{Instant} \to D_\alpha$$

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

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

3.2.5 Temporal Units for Base Types

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

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

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

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

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

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

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

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

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

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

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

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

and evaluation at time t is defined by:

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

3.2.6 Temporal Units for Spatial Data Types

In this subsection we specialize the concept of unit to moving instances of spatial data types.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Moving Lines. We now introduce the unit type for *line* called *uline*. Here we restrict movements of segments so that in the time interval associated to a value of *uline* each segment maintains its direction in the 2-dimensional space. That is, segments which rotate during their movement are not admitted. See in Figure 4 an example of a valid *uline* value. This constraint derives from the need of keeping a balance between ease of

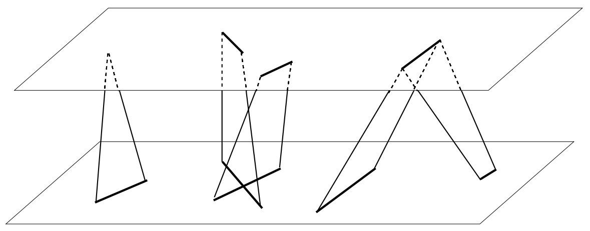

Figure 4: An instance of *uline*

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

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

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

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

The carrier set for <u>*uline*</u> can now be defined as:

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

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

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

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

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

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

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

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

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

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

A final remark on the design decisions for the discrete type for moving lines is the following. Assume we choose instance u_1 (resp., u_2) of <u>*uline*</u> as the discrete representation at the initial (resp., final) time t_1 (t_2) of a unit for the (continuously) moving line l. Then the constraint that segments making up the discrete representation of l cannot rotate during the unit does not restrict too much the fidelity of the discrete representation. Indeed, since members of *MSeg* in a unit can be triangles, this leaves the possibility of choosing among many possible mappings between endpoints of their segments in t_1 and those in t_2, as long as the non-rotation constraint is satisfied. In Figure 5 an example of a discrete representation of a continuously moving line by means of an instance of *uline* is shown. If this approach causes a too rough approximation

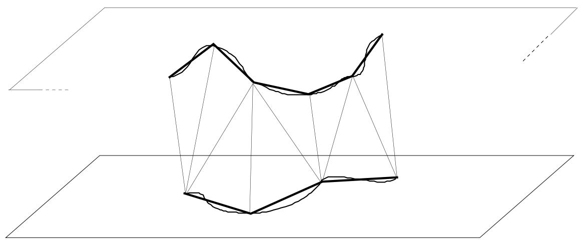

Figure 5: A discrete representation of a moving line

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

Moving Regions. We now introduce the moving counterpart for *region*, namely the *uregion* data type. We adopt the same restriction used for moving lines, i.e., that rotation of segments in the 3-dimensional space is not admitted. We therefore base the definition of *uregion* on the same set of all pairs of lines in a 3D space that are coplanar, namely *MSeg*, with additional constraints ensuring that throughout the whole unit we always obtain a valid instance of the *region* data type. Figure 6 shows an example of a valid *uregion* value. (It also shows the degeneracies that can occur in the end points of a unit interval.)

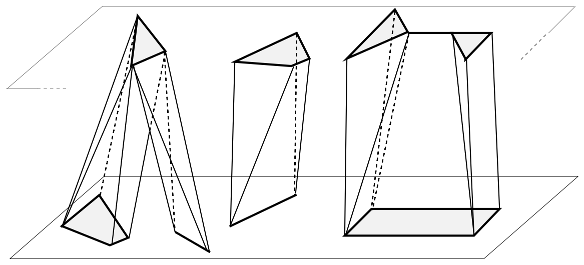

Figure 6: An instance of *uregion*.

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

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

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

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

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

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

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

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

4 Data Structures

The discrete model developed in Section 3 offers a precise basis for the implementation of data structures for a spatio-temporal database system; it is in fact a high-level specification of such data structures. In this section we can therefore, relatively briefly, explain how these definitions translate into data structures. Two general issues need to be considered in that step.

First, some requirements arise from the fact that the data structures implementing the data types are to be used within a database system, and in particular to represent attribute data types within some given data model implementation. This means that values are placed under control of the DBMS into memory which in turn implies that (i) one should not use pointers, and (ii) representations should consist of a small number of memory blocks that can be moved efficiently between secondary and main memory.

One way to fulfill these requirements is to implement each data type by a fixed number of records and arrays; arrays are used to represent the varying size components of a data type value and are allocated to the required size. All pointers are expressed as array indices.

The Secondo extensible DBMS [DG99, GDF+99], under which we are implementing this model, offers a specific concept for the implementation of attribute data types. Such a type has to be represented by a record (called the "root record") which may have one or more components that are (references to) so-called "database arrays". Database arrays are basically arrays with any desired field size and number of fields; additionally they are automatically either represented "inline" in a tuple representation, or outside in a separate list of pages, depending on their size [DG98]. The root record is always represented within the tuple. In our subsequent design of data structures we will apply this concept. Hence each data type will be represented by a record and possibly some (database) arrays. In other DBMS environments one can store the arrays using the facilities offered there for large object management.

Second, many of the data types of Section 3 are set-valued. Sets will be represented in arrays. We always define a unique order on the set domains and store elements in the array in that order. In this way we can enforce that two set values are equal iff their array representations are equal, which makes efficient comparisons possible.

4.1 Non-Temporal Data Types

For the simple types of Section 3.2.1, the implementation is straightforward: they are represented as a record consisting of the given programming language value[3] plus a boolean flag indicating whether the value is defined. Type *point* is represented similarly by a record with two reals and a flag.

A *points* value is represented as an array containing records with two `real` fields, representing points. Points are in lexicographic order. The root record contains the number of points and the (database) array.

The data structures for *line* and *region* values are designed somewhat similar to [GdRS95]. A *line* value is a set of line segments. This is represented as a list of *halfsegments*. The idea of halfsegments is to store each segment twice: once for the left (i.e., smaller) end point and once for the right end point. These are called the left and right halfsegment, respectively, and the relevant point in the halfsegment is called the *dominating* point. The purpose is to support plane-sweep algorithms which traverse a set of segments from left to right and have to perform an action (e.g. insertion into a sweep status structure) on encountering the left and another action on meeting the right end point of a segment. A total order is defined on halfsegments which is lexicographic order extended to treat halfsegments with the same dominating point (see [GdRS95] for a definition).

Hence the *line* value is represented as an array containing a sequence of records each of which represents a halfsegment (four reals plus a flag to indicate the dominating point); these are ordered according to the order just mentioned. The root record manages the array plus some auxiliary information such as the number of segments, total length of segments, bounding box, etc.

A *region* value can be viewed as a set of line segments with some additional structure. This set of line segments is represented by an array *halfsegments* containing the ordered sequence of halfsegment records, as for *line*. In addition, all halfsegments belonging to a cycle, and to a face, are linked together (via extra fields such as *next-in-cycle* within halfsegment records). Two more arrays *cycles* and *faces* represent the structure. The array *cycles* contains records representing cycles by a pointer[4] to the first halfsegment of the cycle and a pointer to the next cycle of the face. The latter is used to link together all cycles belonging to one face. Array *faces* contains for each face a pointer into the *cycles* array to the first cycle of the face. Some unique order is

[3]For *string* we assume an implementation as a fixed length array of characters.

[4]From now on, when we say "pointer" we always mean integer indices referring to a field of some array.

defined on cycles and faces which need not be detailed here.

The root record for *region* manages the three arrays and has additional information such as bounding box, number of faces, number of cycles, total area, perimeter, etc. Algorithms constructing region values generally compute the list of halfsegments and then call a *close* operation offered by the *region* data type, which determines the structure of faces and cycles and represents it by setting pointers.

More details on the representation strategy can be found in [GdRS95] although some details are different here.

Intervals (s, e, lc, rc) are represented by corresponding records. A value of type *range*(α) is represented as an array of interval records ordered by value (all intervals are disjoint, hence there exists a total order). A value of type *intime*(α) is represented by a corresponding record.

4.2 Unit Types

We have to distinguish units that can be represented in a fixed amount of space, called *fixed size units*, and *variable size units*. Fixed size units are *const*(*int*), *const*(*string*), *const*(*bool*)[5], *ureal*, and *upoint*. Variable size units are *upoints*, *uline*, and *uregion*.

Fixed size units can be represented simply in a record that has two component records to represent the time interval and the unit function, respectively. For example, for *ureal* the second record represents the quadruple (a, b, c, r).

For the representation of variable size units, we introduce *subarrays*. Conceptually, a subarray is just an array. Technically it consists of a reference to a (database) array together with two indices identifying a subrange within that array. The idea is that all units within a *mapping* (i.e., a sliced representation) share the same database arrays.

Variable sized units are also all represented by a record whose first component is a time interval record. In the sequel we only describe the second component.

A *upoints* unit function is stored in a subarray containing a sequence of records representing *MPoint* quadruples, in lexicographic order on the quadruples. The *upoints* unit is represented in a record whose second component record contains a subarray reference and a bounding cube[6] (the number of points can be inferred from the subarray indices).

A *uline* unit function is stored similarly in a subarray containing a sequence of records representing *MSeg* pairs which in turn are *MPoint* quadruples. Pairs are ordered lexicographically by their two component quadruples on which again lexicographic order applies. Again the *uline* unit is represented in a record whose second component consists of a subarray reference and a bounding cube.

A *uregion* unit function is basically a set of *MSeg* values (moving segments, trapeziums in 3D) with some additional constraints. We store these *MSeg* records in the same way and order in a subarray *msegments* as for *uline*. In addition, each record has two extra fields that allow for linking together all moving segments within a cycle and within a face. Furthermore, *uregion* has two additional subarrays *mcycles* and *mfaces* identifying cycles and faces, as in the *region* representation. The second component record of a *uregion* unit contains the three subarrays and a bounding cube for the unit.

For both *uline* and *uregion* one might add further summary information in the second component record, such as the (a, b, c, r) quadruples for the time-dependent length (for *uline*) or for perimeter and size (for *uregion*).

[5]We do not consider the other *const*(α) types here, as they are not so relevant in this paper.

[6]This is a bounding box in 3D.

4.3 Sliced Representation

The data structure associated with the *mapping* type constructor organizes a collections of units (slices) as a whole. Obviously this data structure is parameterized by the unit data structures. We observe that all unit data structures are records whose first component represents a time interval, and whose second component may contain one or more subarrays.

The *mapping* data structure is illustrated in Figure 7. It is basically a (database)

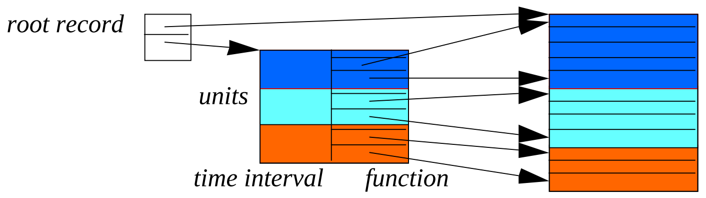

Figure 7: A *mapping* data structure containing three units, for a unit type with one subarray, such as *upoints*.

array *units* containing the unit records ordered by their time intervals. If the unit type uses k subarrays, then the *mapping* data structure has k additional database arrays. Obviously, the database arrays mentioned in the unit subarray references will be the database arrays provided in the mapping data structure. The main array *units* as well as the k additional arrays are referenced from a single root record for the *mapping* data structure. Note that the structure has the general form required for attribute data types.

5 Two Example Algorithms

In this section we briefly show two algorithms in order to illustrate the use of the data model and data structures defined in the previous sections. The first one implements the **atinstant** operation on a moving region, i.e., it determines the region value at a given instant of time. The second one implements the **inside** operation on a moving point and a moving region, hence it returns a moving boolean describing when the point was inside the region.

5.1 Algorithm *atinstant*

The moving region is represented as a value of type *mapping*(*uregion*). The idea of the algorithm is to perform binary search on the array containing the region units to determine the unit u containing the argument time instant t. Then a subalgorithm is called which evaluates each moving segment within the region unit at time t resulting in a line segment in two dimensions. These are composed to obtain the region value returned as a result.

```
algorithm atinstant $(mr, t)$
input: a moving region $mr$ as a value of type mapping(uregion), and an
   instant $t$
output: a region $r$ representing $mr$ at instant $t$
method:
    determine $u \in mr$ such that its time interval contains $t$;
    **if** $u$ exists **then return** $uregion\_atinstant(u, t)$ **else return** $\emptyset$ **endif**
**end** *atinstant*.

**algorithm** $uregion\_atinstant(u, t)$
**input**: a moving region unit $ur$ (of type uregion) and an instant $t$
**output**: a region $r$, the function value of $ur$ at instant $t$
**method**:
    let $ur = (i, F)$; $r := \emptyset$;
    **for each** mface $(c, H) \in F$ **do**
        $c' := \{\iota(s, t) | s \in c\}$;
        $H' := \emptyset$;
        **for each** $h \in H$ **do** $h' := \{\iota(s, t) | s \in h\}$; $H' := H' \cup \{h'\}$ **endfor**;
        $r := r \cup \{(c', H')\}$
    **endfor**;
    **return** $r$
**end** *uregion_atinstant*.
```

In the second algorithm the ι function defined in Section 3 is used to evaluate a moving segment at an instant of time to get a line segment.

The time complexity of this algorithm is basically $O(\log n + r)$ where n is the number of units in mr, and r is the size of the region returned (the number of segments). This is because in the first step of *atinstant* the unit can be found by binary search in $O(\log n)$ time, and the traversal of the unit data structure takes linear time. However, to construct a proper region data structure as described in Section 4.1, one has to produce the list of halfsegments in lexicographic order, and hence needs to sort the r result segments, which results in a time complexity of $O(\log n + r \log r)$. Note that if the region value is just needed for output (e.g. for display on a graphics screen) then $O(\log n + r)$ is indeed sufficient.

The above algorithm works correctly if instant t is internal to the unit time interval. For simplicity, we have ignored in this description the problem of possibly degenerated region values in the end points of the unit time interval, which requires a more complex cleanup after finding the line segments, as sketched at the end of Section 3. This problem can be avoided altogether, by the way, if we spend a little more storage space, and represent a unit with a degenerated region at one end instead by two units, one with an open time interval, and the other with a correct region representation for the single instant at the end.

Analogous implementations of the *atinstant* operation can be obtained for all other moving data types. The first algorithm *atinstant* is in fact generic; one only needs to plug in subalgorithms for other data types.

5.2 Algorithm *inside*

Here the arguments are two lists (arrays) of units, one representing a moving point, the other a moving region. The idea is to traverse the two lists in parallel, computing the refinement partition of the time axis on the way (see Figure 1).

For each time interval i in the refinement partition, an *inside* algorithm is performed on the point and region units valid at that time interval. It produces a set of boolean units representing when the point was inside the region. Note that even a linearly

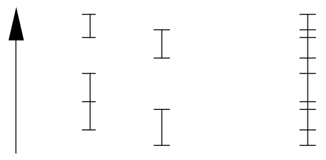

Figure 8: Two sets of time intervals on the left, their refinement partion on the right

moving point within a single <u>*upoint*</u> unit can enter and leave the region of the region unit several times.

```
**algorithm** $inside\ (mp, mr)$
**input**: a moving point $mp$ (of type $\underline{mapping}(\underline{upoint})$), and a moving region
    $mr$ (of type $\underline{mapping}(\underline{uregion})$)
**output**: a moving boolean $mb$, as a value of type $\underline{mapping}(\underline{const}(\underline{bool}))$,
    representing when $mp$ was inside $mr$
**method**:
  let $mp = \{up_1, \ldots, up_n\}$ such that the list $\langle up_1, \ldots, up_n\rangle$ is ordered by
    time intervals;
  let $mr = \{ur_1, \ldots, ur_m\}$ such that the list $\langle ur_1, \ldots, ur_m\rangle$ is ordered by
    time intervals;
  $mb := \emptyset$;
  scan the two lists $\langle up_1, \ldots, up_n\rangle$ and $\langle ur_1, \ldots, ur_m\rangle$ in parallel, determin-
    ing in each step a new refinement time interval $i$ and from each of the
    two lists either a unit $up$ or $ur$, respectively, whose time interval contains
    $i$, or *undefined*, if there is no unit in the respective list overlapping $i$:
  **for each** refinement interval $i$ **do**
    **if** both $up$ and $ur$ exist **then**
      $ub := upoint\_uregion\_inside(up, ur)$;
      $mb := concat(mb, ub)$
    **endif**
  **endfor**;
  **return** $mb$
**end** *inside*.
```

The operation *concat* on two sets of units is essentially the union, but merges adjacent intervals with the same unit value into a single unit. On the array or list representations, as given in the mapping data structure, this can be done in constant time (comparing the last unit of mb with the first unit of ub).

```
**algorithm** $upoint\_uregion\_inside(up, ur)$
**input**: a $\underline{upoint}$ unit $up$, and a $\underline{uregion}$ unit $ur$
**output**: a set of moving boolean units, as a value of type $\underline{mapping}(\underline{const}(\underline{bool}))$,
    representing when the point of $up$ was inside the region of $ur$ during their
    intersection time interval
**method**:
  let $up = (i', mpo)$ and $ur = (i'', F)$ and let $i = (s, e, lc, rc)$ be the inter-
    section time interval of $i'$ and $i''$;[7]
```

[7]For simplicity, the remainder of the algorithm assumes the intersection interval is closed. It is straightforward, but a bit lengthy, to treat the other cases.

```
    **if** the 3d bounding boxes of $mpo$ and $F$ do not intersect **then return** $\emptyset$
    **else**
      determine all intersections between $mpo$ and msegments occurring in
      (the cycles of faces of) $F$. Each intersection is represented as a
      pair $(t, action)$ where $t$ is the time instant of the intersection, and
      $action \in \{enter, leave\}$;[8]
      sort intersections by time, resulting in a list $\langle (t_1, a_1), \ldots, (t_k, a_k) \rangle$
      if there are $k$ intersections. Note that actions in the list must be
      alternating, i.e., $a_i \neq a_{i+1}$;
      let $t_0 = s$ and $t_{k+1} = e$;
      **if** $k = 0$ **then**
        **if** $mpo$ at instant $s$ is inside $F$ at instant $s$ **then**
          **return** $\{((s, e, true, true), true)\}$
        **else return** $\{((s, e, true, true), false)\}$
        **endif**
      **else**
        **if** $a_1 = leave$ **then**
          **return** $\{((t_i, t_{i+1}, true, true), true) | i \in \{0, \ldots, k\}, i \text{ is even}\}$
            $\cup \{((t_i, t_{i+1}, false, false), false) | i \in \{0, \ldots, k\}, i \text{ is odd}\}$
        **else**
          **return** $\{((t_i, t_{i+1}, true, true), true) | i \in \{0, \ldots, k\}, i \text{ is odd}\}$
            $\cup \{((t_i, t_{i+1}, false, false), false) | i \in \{0, \ldots, k\}, i \text{ is even}\}$
        **endif**
      **endif**
    **endif**
  **end** *upoint_uregion_inside*.
```

Here the moving point mpo is a line segment in 3D that may stab some of the moving segments of F, which are trapeziums in 3D. In the order of time, with each intersection the moving point alternates between entering and leaving the moving region represented in the region unit. Hence a list of boolean units is produced that alternates between $true$ and $false$. In case no intersections are found ($k = 0$), one needs to check whether at the start time of the time interval considered the point was inside the region. This can be implemented by a well-known technique in computational geometry, the "plumbline" algorithm which counts how many segments in 2D are above the point in 2D.

The first algorithm *inside* requires time $O(n + m)$, where n and m are the numbers of units in the two arguments, except for the calls to algorithm *upoint_uregion_inside*. This second algorithm requires $O(s)$ time for finding all intersections, with s the number of msegments in F. Furthermore, $O(k \log k)$ time is needed to sort the k intersections, and to return the $k + 1$ boolean units. If no intersections are found, the check whether mpo is inside F at the start time s requires $O(s)$ time. The total time for all calls to *upoint_uregion_inside* is $O(S + K \log k')$ where S is the total number of msegments in all units, K is the total number of intersections between the moving point and faces of the moving region, and k' is the largest number of intersections occurring in a single pair of units. In practical cases, k' is likely to be a small constant, and $K \log k'$ will be dominated by S, hence the total running time will be $O(n + m + S)$. If the moving point and the moving region are sufficiently far apart, so that not even the bounding

[8] The *action* can be determined if we store with each msegment (trapezium or triangle in 3D) a face normal vector indicating on which side is the interior of the region.

boxes intersect, then the running time is $O(n+m)$.

This algorithm illustrates nicely how algorithms for binary operations on moving objects can generally be reduced to simpler algorithms on pairs of units. Again, the first algorithm is generic; one only needs to plug in algorithms for specific operations on pairs of units.

6 Conclusions

We have presented and formally defined a discrete data model that implements the data types defined in the abstract model of [GBE+98]. We have also demonstrated how the discrete representations can be mapped into data structures that can be realistically used in a DBMS environment, and how algorithms can use these data structures. Hence the paper offers a precise basis for the implementation of a "spatio-temporal extension package" to be added to a suitable extensible architecture (e.g. as a data blade to Informix Universal Server).

The next step is to design (more) algorithms for the operations of [GBE+98] and to implement them on these data structures. We are currently building such an extension package and plan to integrate it into the Secondo system as well as make it a data blade for Informix.

References

[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.

[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, Germany, 1997.

[CR99] J. Chomicki and P. Revesz. A Geometric Framework for Specifying Spatiotemporal Objects. In *Proceedings of the 6th International Workshop on Temporal Representation and Reasoning (TIME)*, pages 41–46, 1999.

[DG98] S. Dieker and R.H. Güting. Efficient Handling of Tuples with Embedded Large Objects. Technical Report Informatik 236, FernUniversität Hagen, 1998. To appear in *Data and Knowledge Engineering*.

[DG99] S. Dieker and R.H. Güting. Plug and Play with Query Algebras: Secondo. A Generic DBMS Development Environment. Technical Report Informatik 249, FernUniversität Hagen, 1999.

[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*, pages 131–136, Washington, D.C., November 1998.

[EGSV99] M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):265–291, 1999. In press.

[GBE+98] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M.Vazirgiannis. A Foundation for Representing and Querying Moving Objects. Technical Report Informatik 238, FernUniversität Hagen, 1998. Available at `http://www.fernuni-hagen.de/inf/pi4/papers/Foundation.ps.gz`.

[GDF$^+$99] R.H. Güting, S. Dieker, C. Freundorfer, L. Becker, and H. Schenk. SECONDO/QP: Implementation of a Generic Query Processor. In *Proceedings of the 10th Intl. Conf. on Database and Expert Systems Applications*, pages 66–87, Florence, Italy, September 1999.

[GdRS95] R.H. Güting, T. de Ridder, and M. Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In *Proc. of the 4th Intl. Symposium on Large Spatial Databases*, pages 216–239, Portland, Maine, August 1995.

[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.

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

[PD95] D.J. Peuquet and N. Duan. An Event-Based Spatiotemporal Data Model (ESTDM) for Temporal Analysis of Geographical Data. *International Journal of Geographical Information Systems*, 9(1):7–24, 1995.

[PJ99] D. Pfoser and C.S. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. of the 6th Intl. Symposium on Spatial Databases*, pages 111–131, Hong Kong, China, 1999.

[TH97] N. Tryfona and T. Hadzilacos. Logical Data Modeling of Spatio-Temporal Applications: Definitions and a Model. In *Proc. of the Intl. Database Engineering and Applications Symposium*, 1997.

[TSPM98] Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. of the 10th Intl. Conference on Scientific and Statistical Database Management*, Capri, Italy, 1998.

[WCD$^+$98] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. of the 14th Intl. Conference on Data Engineering*, pages 588–596, Orlando, Florida, 1998.

[Wol98] O. Wolfson. Moving Objects Databases: Issues and Solutions. In *Proc. of the 10th Intl. Conference on Scientific and Statistical Database Management*, Capri, Italy, 1998.

[Wor94] M.F. Worboys. A Unified Model for Spatial and Temporal Information. *The Computer Journal*, 37(1):25–34, 1994.