Algorithms for Moving Objects Databases

José Antonio Cotelo Lema[1], Luca Forlizzi[2], Ralf Hartmut Güting[3], Enrico Nardelli[2] and Markus Schneider[4]

[1] *Laboratorio de Bases de Datos, Facultade de Informática, Universidade da Coruña, A Coruña, Spain*
[2] *Dipartimento di Matematica Pura ed Applicata, Universita Degli Studi di L'Aquila, L'Aquila, Italy*
[3] *Praktische Informatik IV, FernUniversität Hagen, D-58084 Hagen, Germany*
[4] *Database Systems Research & Development Center, University of Florida, Gainesville, FL 32611-6125, USA*
Email: gueting@fernuni-hagen.de

Whereas earlier work on spatio-temporal databases generally focused on geometries changing in discrete steps, the emerging area of *moving objects databases* supports geometries changing continuously. Two important abstractions are *moving point* and *moving region*, modeling objects for which only the time-dependent position, or also the shape and extent are relevant, respectively. Examples of the first kind of moving entity are all kinds of vehicles, aircraft, people, or animals; of the latter hurricanes, forest fires, forest growth, or oil spills in the sea. The goal is to develop data models and query languages as well as DBMS implementations supporting such entities, enabling new kinds of database applications.

In earlier work we have proposed an approach based on abstract data types. Hence, *moving point* or *moving region* are viewed as data types with suitable operations. For example, a moving point might be projected into the plane, yielding a curve, or a moving region be mapped to a function describing the development of its size, yielding a real-valued function. A careful design of a system of types and operations (an algebra) has been presented, emphasizing completeness, closure, consistency and genericity. This design was given at an abstract level, defining, for example, geometries in terms of infinite point sets. In the next step, a discrete model was presented, offering finite representations and data structures for all the types of the abstract model.

The present paper provides the final step towards implementation by studying and developing systematically algorithms for (a large subset of) the operations. Some of them are relatively straightforward; others are quite complex. Algorithms are meant to be used in a database context; we also address filtering techniques and practical issues such as large object management or numeric robustness in the context of an ongoing prototype implementation.

1. INTRODUCTION

The emerging area of *moving objects databases* has recently received a lot of interest. Whereas earlier work on spatio-temporal databases focused on geometries changing in discrete steps, here the goal is to develop data models and query languages supporting geometries evolving continuously, hence *moving objects*. Two important abstractions are *moving point*, describing entities for which only the time-dependent location needs to be managed, and *moving region*, for entities whose time-varying shape and extent is relevant. Examples of the former are cars, aircraft, ships, mobile phone users, terrorists, or polar bears, of the latter hurricanes, oil spills in the sea, forest fires, armies, or tribes of people in history. Hence a wide range of database applications managing such objects becomes feasible.

In earlier work [11, 21, 14] we have developed a *data type oriented approach* for modeling and querying such data. The idea is to consider the two major abstractions *moving point* and *moving region* as abstract data types with suitable operations that can be embedded into a DBMS data model and query language as attribute types. Operations of interest are, for example, evaluation of a moving region at a given instant of time (yielding a region), projecting a moving point into the 2D space (resulting in a 2D curve), or determining when a moving point was inside a moving region (yielding a

time-dependent boolean).

Whereas [11] describes the approach and discusses some related questions and options, reference [21] presents a careful design of a system of data types and related operations (an algebra). Here the emphasis is on completeness, closure, consistency, and genericity in the application of type constructors and the design of operations.

An important fundamental issue discussed already in [11] is which *level of abstraction* should be used in defining such an algebra. For example, a moving region could be defined either as a continuous function from time into region values, or as a polyhedral shape in the (2D + time) space. Characteristic for the first level of abstraction is that it defines types in terms of infinite point sets without regard to finite representations; for the second level, that it is always necessary to provide finite representations. In [11] the terms *abstract model* and *discrete model*, respectively, have been introduced for these two levels of abstraction.

In [21] the semantics of types and operations have been defined at the level of an abstract model. The paper [14] continues this work in defining a corresponding discrete model. Hence it provides finite representations, as well as data structures, for all the types of the abstract model.

The next task in this development towards implementation is the study of algorithms for the rather large set of operations defined in [21]. That is the purpose and focus of the present paper. Whereas [14] just provides a brief look into this issue by presenting two example algorithms at the end, in this paper we present a comprehensive, systematic study of algorithms for a quite large subset of the operations in [21]. It is still a subset, to keep the task manageable, but this subset is formed by a systematic restriction of the scope of the study. Whereas some algorithms are relatively straightforward and simple, there is still a considerable number of quite involved ones. In all cases we analyze the complexity. The data structures from [14] are also refined and extended by auxiliary fields to speed up computations.

Hence this paper offers a blueprint for implementing a "moving objects" extension package (*data blade, cartridge, extender*) for suitable extensible, e.g. object-relational, database architectures. We are also working on a corresponding prototype implementation; at the end of the paper this implementation is described and some of the practical issues going beyond just algorithms, are discussed.

Earlier research on spatio-temporal databases generally focused on the treatment of discrete changes. Examples of such models are [41, 27]. Survey articles on spatio-temporal research can be found in [1] and [26]; the latter already covers some of the recent work on moving objects.

The group of Wolfson has proposed in [32, 38, 40, 39] a concept of moving objects databases that is complementary to ours. Whereas our approach of modeling describes movement in the past[5], hence the complete history of a moving object, their focus is on capturing the *current* movement of entities, e.g. vehicles, and their anticipated loations in the near future. The basic idea is to store in a database not the actual location of an object, which would need to be updated frequently, but instead a *motion vector* describing location, speed and direction for a recent instant of time. As long as the predicted position based on the motion vector does not deviate from the actual position more than some threshold, no update to the database is necessary. An important issue is, for example, to balance the cost of updates against the cost of imprecise information. They also offer a query language based on temporal logic to formulate questions about the near future movement. – This approach is restricted to moving point objects, i.e., does not address more complex geometries such as moving regions.

The constraint database approach is suitable for modeling geometries in databases in a dimension-independent way. This can obviously also be used to describe spatio-temporal entities, e.g. in a 3D (2D + time) space. Especially relevant are the papers by Grumbach and colleagues [16, 17, 18, 19] who have implemented with the Dedale system one of the few prototypes of constraint DBMS. Whereas [16, 17] consider static geometries, [19] shows some spatio-temporal examples (although these are still restricted to stepwise constant geometries). In [18] a general concept for interpolation is developed which is important not only for moving objects, but also for digital terrain modeling.

Further work related to the constraint approach is [5, 6, 4]. The first [5] briefly addresses polygons whose vertices move linearly with time. In [6] simple geometric entities are considered whose temporal development can be described by affine mappings; for them closure properties are investigated. Reference [4] desribes moving regions as sets of parametric rectangles, that is, rectangle boundaries are linear functions of time. This work has a more theoretical focus.

As an extension to our abstract model in [21], the work in [13] introduces a concept of *spatio-temporal predicates*. The goal is to investigate temporal changes of topological relationships induced by temporal changes of spatial objects. A corresponding spatio-temporal query language incorporating these concepts is presented in [12].

Further work on modeling includes [28, 33, 37]. In [28] the modeling of position uncertainty due to a limited set of observations is addressed. [33] focuses on moving point objects and the inclusion of concepts of differential geometry (speed, acceleration) in a calculus based query language. Paper [37] considers movement in networks and some evaluation strategies.

[5] It could also be a scheduled movement within any kind of time frame.

At the implementation level, some work addresses indexing of current movement [25, 31, 2] and also of motion history [29]. There is also some interesting work on generators for test data which allow one to create sets of (descriptions of) moving objects either in a parametric way [34, 36] or in a kind of simulation approach [30]. Algorithms for creating complete moving region descriptions by interpolation from "snapshots" (observations at certain instants of time) are studied in [35].

As far as industrial solutions are concerned, the major DBMS vendors do offer extension modules for spatial or for temporal support. For example, Informix has the "Informix Spatial Datablade" and the "Informix Geodetic Datablade" for spatial support and the "Time Series Datablade" for temporal support. IBM has the "DB2 Spatial Extender"; Oracle the "Oracle Spatial Cartridge" and the "Oracle Time Series Cartridge". However, we are not aware of a module supporting spatio-temporal data, not even discretely changing spatial values. So currently the combined use of spatial and temporal fields in tables is the only way to provide applications with a very limited support for discretely changing geometries.

To our knowledge, except for [21] there does not exist in the literature a comprehensive design of spatio-temporal data types and operations, much less a systematic study of how such types and operations can be implemented. This paper is the first one to present a careful design and analysis of data structures and algorithms for an algebra on moving objects. It provides a solid basis for implementing a "moving objects datablade".

The paper is structured as follows: Section 2 briefly reviews the main concepts of [21, 14] which form the basis for this paper. Section 3 describes the employed data structures in detail including *summary fields* containing derived information to speed up computation, e.g. by filter steps. This is the basis for describing and analyzing algorithms. Sections 4 and 5 systematically investigate algorithms for the two major classes of operations studied. Some of the most involved algorithms, namely set operations on moving regions, and computations of distance functions involving moving regions, are treated in Section 6. Section 7 describes our prototype implementation and certain implementation issues. Finally, Section 8 offers conclusions.

2. REVIEW

In this section we review the material on which this paper is based, namely the abstract model for moving objects of [21] and the discrete model developed in [14]. In the last subsection we define a concept to systematically select a subset of the (signatures of) operations in [21] for study in this paper.

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

TABLE 1. Signature describing the abstract type system

2.1. The Abstract Model

Data Types. The abstract model of [21] offers the data types, or actually the *type system* shown in Table 1.

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*.[6] The terms generated by the signature are the available *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 may lie within holes of other faces.

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 *moving*(α) 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 [21]). For example, a *moving*(*region*) value is a function from time into *region* values.

To support a systematic design of operations, the paper [21] has a concept of *spaces* and within each space a notion of *point types* and *point set types*, representing single values and sets of values from the space, respectively. For example, *Integer* is a (1D) space with a point type *int* and a point set type *range*(*int*), and *2D* represents the two-dimensional space and has one point type *point* and three point set types *points*, *line*, and *region*.

[6] We write signatures by giving first the argument and result sorts, and then the operators with this functionality. As a convention, kinds are denoted by capitals and type constructors in italic underlined. Operations on data types are written in bold face.

Class	Operations
Predicates	**isempty** =, ≠, **intersects**, **inside** <, ≤, ≥, >, **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

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**
Rate of Change	**derivative**, **speed** **turn**, **velocity**

TABLE 3. Classes of Operations on Temporal Types

Operations. Over the types so defined, the abstract model offers a large set of operations. In a first step, it defines generic operations over the non-temporal types (all types except those constructed by *moving* or *intime*). These operations include predicates (e.g. **inside** or ≤), set operations (e.g. **union**), aggregate operations, operations with numeric result (e.g. **size** of a region), and distance and direction operations. Table 2 provides an overview, just listing the names of operations. The precise signatures (i.e., the possible combinations of argument and result types) and the meaning of these operations will be given in Sections 4 through 6 of this paper.

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 (see Table 3). 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. Details about these operations can be found in Section 4.

DBMS Embedding and Queries. 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

Operation	Signature	
trajectory	*moving*(*point*)	→ *line*
length	*line*	→ *real*
distance	*moving*(*point*)	
	×*moving*(*point*)	→ *moving*(*real*)
atmin	*moving*(*real*)	→ *moving*(*real*)
initial	*moving*(*real*)	→ *intime*(*real*)
val	*intime*(*real*)	→ *real*

```
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.[7]

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 [21].

The projection of a moving point 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 *line* value. The distance between two moving points is calculated by **distance**, used here in its temporally lifted version. Operation **atmin** restricts a moving real to all times with the same minimal *real* value. The first (*instant*, *real*) pair of a moving real is returned by the operation **initial**. Operation **val** is here applied to a (*instant*, *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
```

[7] It is true that air planes move in a 3D space whereas our model describes movement in the 2D plane. We ignore this for the moment as flights are a nice example to illustrate the approach, and one can easily imagine a 3D version of the *moving*(*point*) data type.

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

TABLE 4. Signature describing the discrete type system

This query represents an instance of a *spatio-temporal join*. Many further illustrating query examples from different application scenarios (e.g., multimedia presentations, forest fire control management) can be found in [21]. These applications demonstrate that a very flexible and powerful query language results from this design.

2.2. The Discrete Model

In [14] data types are defined that can represent values of corresponding types of the abstract model. Of course, the discrete types can in general only represent a subset of the values of the corresponding abstract type.

Let us once more clarify the role of the discrete model: It introduces data types whose domains are defined in terms of finite representations. Hence it is at an intermediate level in abstraction between an abstract model and concrete data structures. For example, in a discrete model we may define values of a type to be "finite sets of integers" without yet fixing a data structure for this (such as an array of integers).

All type constructors of the abstract model 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 4.

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.

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

TABLE 5. Correspondence between abstract and discrete temporal types

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

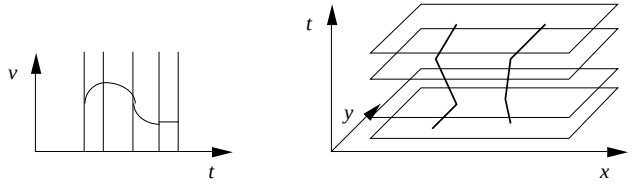

FIGURE 1. Sliced representation of moving *real* and moving *points* value

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. There exist unit types *ureal*, *upoint*, *upoints*, *uline*, and *uregion*. For values that can only change discretely, there is a trivial "simple" function, namely the constant function. It is provided by a *const* type constructor which produces units whose second component is just a constant of the argument type. This is in particular needed to represent moving *int*, *string*, and *bool* values. A *mapping* then is basically a finite set of units whose time intervals are pairwise disjoint. The data structure for this (an array of units ordered by time intervals) is discussed in Section 3.3.

In summary, we have the correspondence between abstract and discrete temporal types shown in Table 5. There we have omitted the representations *mapping*(*const*(*real*)), etc. which can be used to represent discretely changing real values and so forth, but are not so relevant here.

We introduce a number of abbreviations for the data types obtained by application of type constructors (see Table 4). All the *range* types will be

denoted by a prefix r; hence instead of *range*(*int*), *range*(*real*), etc. we can write *rint*, *rreal*, and so forth. For *range*(*instant*), the type describing a set of time intervals, we introduce a special name *periods*. All the *intime* types are abbreviated by a prefix i, hence we have *iint* for *intime*(*int*), for example. All data types representing moving objects, such as e.g. *mapping*(*const*(*int*)), *mapping*(*ureal*), *mapping*(*upoint*), or *mapping*(*uregion*), are abbreviated by a prefix m, hence we have corresponding type names *mint*, *mreal*, *mpoint*, or *mregion*, respectively.

The data structures chosen to represent values of the discrete model, and which are manipulated by the algorithms developed in this paper, are explained in some detail in Section 3.

2.3. Selecting a Subset of Algorithms

In the design of operations in [21], the emphasis was on consistency, closure, and genericity; in particular, all operations have been defined to be applicable to all combinations of argument types for which they could make any sense. Whereas this is nice and simple for the user, it leads to a very large set of functionalities for operations. Since it is not always the case that different argument types for one operation can be handled by the same algorithm, the task addressed in this paper – namely to design algorithms for the operations – is a very large one. To make it manageable, we try to reduce the scope of the study a bit, as follows.

1. We do not study algorithms for operations on non-temporal types (shown in Table 2) as such; this kind of algorithms on static objects has been studied before in the computational geometry or spatial database literature. An example would be an algorithm for intersecting two *region* values. However we will study the lifted versions of these operations which involve moving objects.
2. We do not consider the types *moving*(*points*) and *moving*(*line*) or any signature of an operation involving these types. These types have been added in the design of [21] mainly for reasons of closure and consistency; they are by far not as important as the types *moving*(*point*) and *moving*(*region*) which are in the focus of interest.
3. We do not consider predicates based on topology (i.e. dealing with boundaries of objects); these are the predicates **touches**, **attached**, **overlaps**, **on_border**, and **in_interior**. These may be treated in a follow-up study. No doubt they are useful, but we need to limit the scope of this paper.

All other operations and functionalities are systematically considered in Sections 4 and 5. The paper [21] has a very compact notation to describe signatures for operations. Together with the restrictions just mentioned it is not so easy to figure out, which functionalities remain. Therefore in the following sections we list explicitly for each operation which signatures remain to be considered.

3. DATA STRUCTURES

In this section we describe in more detail the various data types involved and the data structures used to represent them. This is the basis for describing and analyzing the algorithms of the following sections.

3.1. General Requirements and Strategy

As already pointed out in [14], the data structures for the different data types defined here must fulfill some requirements. They are intended to be used within a database system, probably as an extension package of some extensible database system, which implies that values will be placed into memory under the control of the DBMS. As a result, the proposed data structures should not use pointers and their representation should consist of a small number of memory blocks that can be moved efficiently between secondary and main memory.

To fulfill these requirements, data types are generally represented by a record (called the *root record*), which contains some fixed size components and possibly one or more components which are references to arrays. Any part of the data structure that is of varying size will be represented by such an array.

As we will see below, all data structures can easily be designed following that schema with one exception: The data structure representing a moving region (i.e., *mapping*(*uregion*)) is conceptually a record containing some pointers to arrays which in turn contain again pointers to arrays. Hence it is a two-level tree. However, we will show in Section 7 how this structure can be mapped to one of the first kind, so that indeed all data structures have the form required above. Section 7 will also explain how data types represented in this way can be managed efficiently as attribute data types in a DBMS.

In this paper, in addition to what has already been explained in [14], we extend data structures by various *summary fields*. These contain auxiliary information derived from the value represented that can help to speed up certain operations.

In the sequel, we describe data structures for data types (roughly) in the order given by Table 4.

3.2. Non-Temporal Data Types

Base Types and Time Type. For the base types *int*, *real*, *string*, *bool* and for type *instant* the implementation is straightforward: they are represented by a record which consists of a corresponding programming language value together with a boolean flag indicating whether the value is defined. For *string* the value is a fixed length array of characters. For *instant*, the value is of a data type *coordinate*

which is a rational number of a certain precision.[8] In order to be able to represent the entire time domain by intervals, we introduce additionally two predefined constants *mininstant* and *maxinstant* describing the first and last representable instants in the past and the future, respectively. This means that we assume a bounded, rather than an infinite, time domain at the discrete level.

Spatial Data Types. A *point* is represented by a record with two coordinates (values of the *coordinate* type) for x and y and a *defined* flag. The coordinate type is also used in all definitions of points in the following three spatial data types.

The data types *points*, *line*, and *region* are illustrated in Figure 2. In addition to the value representation described next they contain summary fields which will be explained at the end of this section.

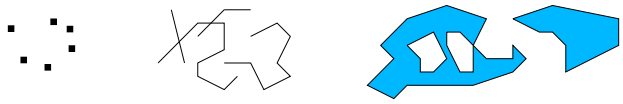

FIGURE 2. A *points* value, a *line* value, and a *region* value

A *points* value is a finite set of points in the plane. It is represented by a (root) record containing a reference to an array. Each element of the array represents one point by its two coordinates. Points are in (x, y)-lexicographic order.

The data structures for *line* and *region* are similar to the ones proposed in [22]. A *line* value at the discrete level is a finite set of line segments that are intersection-free.[9] It is represented as a root record with one array 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 of this 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. Each halfsegment is represented as a pair of *point* values (representing the end points) plus a flag to indicate the dominating end point. Halfsegments are ordered in the array following a lexicographic order extended to treat halfsegments with the same dominating point (see [22] for a definition).

[8] See Section 7 for a discussion.

[9] Here we deviate from the description in [14] where intersecting segments were allowed. An original pair of intersecting segments is instead represented by four segments meeting in an end point. The reason is that we want to reuse the ROSE algebra implementation [22] which has this requirement. See Section 7.

A *region* is given by the set of line segments forming its boundary. There exists an additional structure, however. A region is a finite set of disjoint *faces*. Each face is a polygon possibly containing some polygonal holes. We call the boundary of a simple polygon a *cycle*, hence a face can be represented as a pair (c, H), where c is an *outer cycle* and H is a set of *hole cycles*. A precise definition can be found in [14].

A *region* is represented by a root record with three arrays. The first array (*segments array*) contains the sequence of *halfsegments*, as for *line*. Each record of the segments array contains a halfsegment plus an additional field *next-in-cycle* which links the segments belonging to a cycle (in clockwise order for outer cycles, counter-clockwise for hole cycles, so the area of the face is always to the right). So one can traverse cycles efficiently.

The second and third array (*cycles* and *faces* array) represent the list of cycles and the list of faces belonging to the *region*, respectively. They are also suitably linked together so that one can traverse the list of cycles belonging to a face, for example. More details about such a representation can be found in [22].

Summary Fields. For the three data types *points*, *line*, and *region* representing point sets in the plane, we introduce the following summary fields stored in the respective root records:

- *object_mbb* – the object's bounding box, a rectangle. The minimum bounding rectangle for all points or segments of the object. For *points*, *line*, and *region*.
- *no_components* – an integer. Contains the number of points for *points*, the number of connected components for *line*, and the number of faces for *region*. Used to support the corresponding algebra operation.
- *length* – a real number. The total length of line segments for a *line*.
- *perimeter, area* – real numbers. For *region*.

In addition, obviously for all the arrays used in the representation there is a field giving their actual length. Hence one can determine the number of segments or faces for a *region*, for example.

Summary fields are used to speed up operations. For example, bounding boxes (here *object_mbb*) are widely used in geometric query processing to perform a first check before examining the exact geometries. E.g., if an **intersects** predicate on two *region* values needs to be evaluated, one first checks whether the bounding boxes overlap. Only in this case the exact geometries need to be considered, perhaps in a plane-sweep algorithm. Other summary fields directly support corresponding operations. For example, to determine **perimeter** or **area** of a *region* within a query, one can just look up the stored values in $O(1)$ time instead of starting the algorithm to compute this value.

Sets of Intervals. The range data types *rint*, *rreal*, *rstring*, *rbool*, and *periods* are represented by a root record containing an array whose entries are interval records ordered by value (all intervals must be disjoint and non-adjacent, hence there exists a total order). An interval record contains the four components (s, e, lc, rc), where s and e are the start and end value of the interval, respectively (therefore of type *int*, *real*, etc.), and lc and rc are booleans indicating whether the interval is left-closed and right-closed, respectively.

Summary Fields. For the range types, we store in the root record also:

- *no_components* – an integer. The number of intervals.
- *min, max* – of the corresponding data type. The minimal and maximal value assumed in the set of intervals.
- *duration* – a real number. The sum of the lengths of all intervals, for *periods*.

These summary fields support the corresponding operations **no_components**, **min**, **max**, and **length**.

Instant, Value Pairs. An *intime* value of type *iint*, *ireal*, *istring*, *ibool*, *ipoint* or *iregion* is represented by a corresponding record (*instant, value*), where *value* is of the corresponding data type.

3.3. Temporal Data Types

Sliced Representation. Temporal (moving) data types can be conceptually described as a set of units (see Figure 1). A unit is represented by a record containing a pair of values (*interval, unit-function*). The *interval* defines the time interval for which the unit is valid; it has the same form (s, e, lc, rc) as intervals in the range types. The *unit-function* represents a function from time to the corresponding non-temporal type α which returns a valid α value for each time instant in *interval*. For each temporal type there will be a corresponding *unit-function* data structure. The time intervals of any two distinct units are disjoint; hence units can be totally ordered by time.

Units. Units for discretely changing types *const*(*int*), *const*(*string*), and *const*(*bool*) use as a unit function simply a value of the corresponding non-temporal type. Hence, for a unit (i, v), the function is $f(t) = v$.

The *ureal* unit function is represented by a record (a, b, c, r), where a, b, c are real numbers and r is a boolean value. The function represented by this 4-tuple is $f(t) = at^2+bt+c$ if r is *false*, and $f(t) = \sqrt{at^2 + bt + c}$ if r is *true*. Hence we can represent (piecewise) quadratic polynomials and square-roots thereof. – The data type is designed in this way so that it can represent the results of at least the most important numeric operations, namely time dependent **perimeter** and **area** for moving regions (linear and quadratic polynomials, see Section 5.4) and **distance** between moving points and moving regions (square roots of quadratic polynomials, Sections 5.5 and 6.2).

A *upoint* unit function is represented by a record (x_0, x_1, y_0, y_1), representing the function $f(t) = (x_0 + x_1 t, y_0 + y_1 t)$. Such functions describe a linarly moving point. We also call the tuple (x_0, x_1, y_0, y_1) an *mpoint* ("moving point").

A *uregion* unit function is essentially a *region* whose vertices move linearly (i.e. whose vertex positions are linear functions of time), such that for all time instants in the unit time interval, evaluating the vertex functions yields a correct *region* value. Figure 3 shows an example. For simplicity, this one consists only of a single (moving) face without holes. In general, region

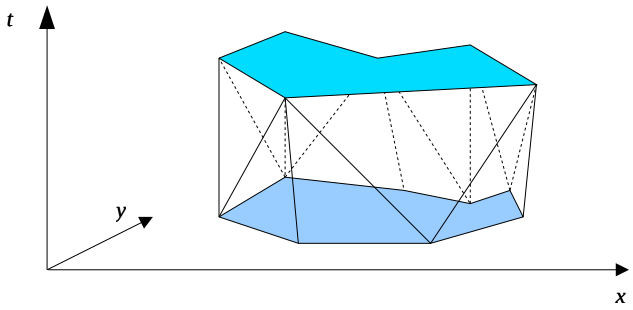

FIGURE 3. A *region* unit

units have the same structure as *region* values, i.e. may consist of multiple faces that may contain holes. The major difference is that they are built from *msegments* ("moving segments") instead of ordinary segments. An *msegment* is a pair of *mpoints* that are co-planar in the 3D space. Hence an msegment, restricted to a time interval, is a trapezium in the 3D space that may degenerate into a triangle. Figure 4 shows two examples. Note that the "walls" of the region unit in Figure 3 are built from such msegments.

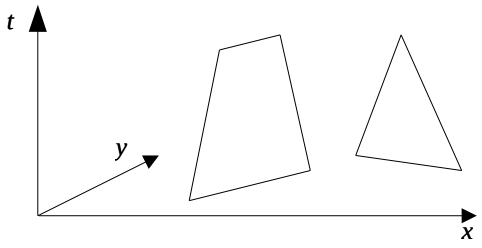

FIGURE 4. Two moving segments (*msegments*)

A *uregion* unit function is represented by a record containing three arrays, namely an *msegments* array, a *cycles* array and a *faces* array. The *msegments* array stores the msegments of the unit, using lexicographic order on the tuples defining the msegment. As for *region*, each msegment record has an additional field *next-in-cycle*, and msegments of a cycle are linked in

cyclic order, having always the interior of the face at their right. The *cycles* and *faces* arrays are managed similarly as for *region*. The cycles array keeps a record for each cycle in the *uregion*, containing a pointer[10] to the *first-mseg-in-cycle* and a pointer to the *next-cycle-in-face*. The faces array stores one record per face, with a pointer to the *first-cycle-in-face*.

Representation of Temporal Types (Moving Objects). A temporal data type is represented as a root record containing an array of units ordered by their time interval.

Note that all the unit types can be represented in a single record except for the *uregion* type. In the latter case, the record contains references to arrays. Hence, as already mentioned in Section 3.1, the *mregion* representation is conceptually a two-level tree which will be mapped to a single root record with some arrays as explained in Section 7.

Summary Fields. We now introduce summary fields, first at the level of the entire moving object, and second at the level of individual units. Summary fields are added to the root record of the moving object, or the record representing the unit, respectively.

Object Level.

- For all temporal types:
  - *no_units* – an integer.
  - *deftime* – a *periods* value. The set of time intervals for which the moving object is defined. Obtained from merging the definition time intervals of the units. We also call this the *deftime index*. Note that a *periods* value is represented by a root record containing a reference to an array. Here the information in the root record is integrated into the root record of the moving object which now contains a *deftime* array as well as its *units* array.
- For the non-spatial temporal types *mint*, *mreal*, *mstring*, *mbool*:
  - *min, max* – of the respective data type. The minimum and maximum value that the object takes in all its definition time.
- For the spatial temporal types *mpoint* and *mregion*:
  - *object_pbb* – the projection bounding box, a rectangle. A projection bounding box represents the minimum bounding box of all points in the 2D space that at some time instant belong to the spatiotemporal object.

Unit Level.

- For *ureal*:
  - *unit_min, unit_max* – real numbers. The minimum and maximum value assumed by the unit function.
- For *upoint* and *uregion*:
  - *unit_pbb* – a rectangle. The bounding box for the spatial projection of the unit (analogous to the *object_pbb*).
- For *uregion*:
  - *unit_no_components* – an integer. The number of moving faces of the unit.
  - *unit_perimeter, unit_area* – real unit functions, describing the development of the perimeter and the area during the unit interval.
  - *unit_ibb* – the interpolation bounding box, a "moving rectangle". This is a more precise filter than the *unit_pbb*. It connects the bounding box of the *uregion* projection at the start time of the unit with the bounding box of the projection at the end time. The *unit_ibb* is stored as a record $(a_{xmin}, b_{xmin}, a_{xmax}, b_{xmax}, a_{ymin}, b_{ymin}, a_{ymax}, b_{ymax})$, representing one linear function f_i for each bounding box coordinate (*xmin*, *xmax*, *ymin* and *ymax*), with the value $f_i = a_i t + b_i$.

These summary fields are used in various algorithms presented in Sections 4 through 6. Some examples: The *deftime* field is used e.g. in the **deftime** and **present** operations (Sections 4.2 and 4.3) and in the lifted **union** and **minus** operations (Section 5.2). The *unit_min* and *unit_max* fields are used in the **rangevalues** operation (Section 4). The various projection bounding boxes are to be used for a sequence of filter steps, as explained in Section 4.1.

4. ALGORITHMS FOR OPERATIONS ON TEMPORAL DATA TYPES

In this section we start by giving algorithmic descriptions of operations on temporal types (i.e. moving objects) as shown in Table 3, namely for projection into domain and range[11] (Section 4.2), for interaction with values from domain and range (Section 4.3), and for rate of change (Section 4.4).

4.1. Common Considerations

Notations. From now on, we denote the first and the second operand of a binary operation by a and b, respectively. We denote the argument of unary operations by a. In complexity analysis, m and n are the numbers of units (or intervals) of, respectively, a and b, while r is the number of units in the result. If a is a type having a variable size, we denote by M the

[10] The "pointers" mentioned here are always represented by array indices.

[11] The **routes** operation mentioned in Table 3 is applicable only to a *moving*(*line*) and therefore not considered in this paper.

number of "components" of a. That is, for example, if a is of type *points* then M is the number of points contained in a, while if a is of type *mregion* then M is the number of moving segments composing a. In any case the size of a is $O(M)$. For the second argument b and for the result of an operation, we use with the same meaning N and R, respectively. If a (resp. b, the result) is of type *mregion*, we denote by u (resp. v, w) the number of moving segments composing one of its units, and by $u_{\max}$ (resp. $v_{\max}$, $w_{\max}$) the maximum number of moving segments contained in a unit. Finally, let d denote the size of the *deftime* index of a moving object.

All complexity analyses done in this paper consider CPU time only. So this assumes that the arguments are in memory already and does not address the problem of whether they need to be loaded entirely or this can be avoided.

A study of I/O complexity is left to future work. It depends on further implementation details such as those we discuss in Section 7. Note that the implementation described there indeed makes it possible to load argument objects only partially.

Algorithmic Schemes. We now describe several algorithmic schemes that are common to many operations. In the following we call an argument of a temporal type a *moving argument*. Every binary operation whose arguments are both moving ones, requires a preliminary step where a *refinement partition* of the units of the two arguments is computed. A refinement partition is obtained by breaking units into other units that have the same value but are defined on smaller time intervals, so that a resulting unit of the first argument and one of the second argument are defined either on the same time interval or on two disjoint time intervals. We denote the number of units in the refinement partition of both arguments by p. Note that $p = O(n + m)$. We use $\hat{M}$ (resp. $\hat{N}$) with the same meaning as M (resp. N) referring to the size of the refined partition of the units of a (resp. b). We compute the refinement partition by a parallel scan of the two lists of units, with a complexity of $O(p)$. This complexity is obvious for all types that have units of a fixed size, hence for all types but *mregion*. Even for the latter type this complexity can be achieved, if region units are not copied, but pointers to the original units are passed to the subalgorithm processing a pair of units for a given interval of the refinement partition. If the refinement partition for two *mregion* arguments is computed explicitly instead (copying units), the complexity is $O(\hat{M} + \hat{N})$.

For many operations whose result is of one of the temporal types, a post-processing step is needed to merge adjacent units having the same value. This requires time $O(r)$. Usually this step will be integrated in the construction of result units.

We also assume, for each endpoint of a unit interval of a region unit, that if the endpoint is included in the unit interval, then the value of the region unit at the time instant corresponding to the endpoint is a valid instance of type *region* (i.e. is not a *degenerate value* which so far it could be according to the definition in [14]). This is not a restriction since if such a requirement is not satisfied for an endpoint $\bar{t}$ of a region unit $\bar{a}$, we can exclude $\bar{t}$ from the unit interval of $\bar{a}$ and add a new region unit whose unit interval consists only of $\bar{t}$ and whose value is obtained evaluating the value of $\bar{a}$ at time $\bar{t}$ and removing degeneracies.

Filtering Approach. Even if not stated, each algorithm filters[12] its arguments using the auxiliary information (i.e. the summary fields) provided by them, which varies according to arguments' types (see Section 3). In particular, minimum and maximum values (stored in the *min* and *max* fields of the root record) for moving non-spatial types, and bounding boxes for non temporal spatial types, are used. For *mpoint* and *mregion* filtering is performed using projection bounding boxes. Moreover, for *mregion*, two more filtering steps, with increased selectivity, are performed using first projection bounding boxes and then interpolation bounding boxes of individual units.

Semantics of Operations. In the sequel we describe algorithms for many operations. For each operation, its meaning is briefly explained informally, we hope, sufficiently clearly. However, if there is any doubt, remember that these operations have originally been defined in [21] and refer to the precise definition of semantics given there.

Signature Abbreviations. Most of the operations are polymorphic, i.e., allow for several combinations of argument and result type. To avoid long listings of signatures, but to be still precise about what signatures are admitted, we introduce the following abbreviation scheme, here illustrated for the **rangevalues** operator:

For $\alpha \in \{$*int*, *bool*, *string*, *real*$\}$:

rangevalues	$m\alpha$	$\rightarrow$	$r\alpha$

Here α is a type variable ranging over the types mentioned; each binding of α results in a valid signature. Hence this specification expands into a list:

rangevalues	*mint*	$\rightarrow$	*rint*
	mbool	$\rightarrow$	*rbool*
	mstring	$\rightarrow$	*rstring*
	mreal	$\rightarrow$	*rreal*

4.2. Projection to Domain and Range

The operations described in this subsection get a *moving* or *intime* value as operand and compute

[12] The term *filter* is widely used in geometric query processing to describe a prechecking on approximations. For example, a spatial join on two sets of regions may be implemented by first finding pairs of overlapping bounding boxes and then performing a precise check of geometries on the qualifying pairs. It is used here and elsewhere also to describe prechecking of approximations of two single spatial data type values.

different kinds of projections either with respect to the temporal component (i.e., the domain) or the function component (i.e., the range) of a moving value.

deftime. This operation returns all times for which a moving object is defined. It has the following signatures:

For $\alpha \in \{\underline{int}, \underline{bool}, \underline{string}, \underline{real}, \underline{point}, \underline{region}\}$:

deftime $\underline{m\alpha} \rightarrow \underline{periods}$

The algorithmic scheme is the same for all operation instances, namely to read the intervals from the *deftime* index incorporated into each argument object. The time complexity is $O(r) = O(d)$.

rangevalues. This operation is defined for one-dimensional argument types only and returns all the values assumed by the argument over time, as a set of intervals. We obtain the following signatures:

For $\alpha \in \{\underline{int}, \underline{bool}, \underline{string}, \underline{real}\}$:

rangevalues $\underline{m\alpha} \rightarrow \underline{r\alpha}$

For the type *mbool*, in $O(1)$ time we look up the minimal range value *min* and the maximal range value *max* of the moving boolean. The result is one of the interval sets $\{[false, false]\}$, $\{[true, true]\}$, or $\{[false, true]\}$.

For the types *mint* and *mstring* we scan the mapping, insert the range values into a binary search tree, and finally traverse the tree and report the ordered sequence of disjoint intervals. This takes $O(m + m \log k)$ time if k is the number of different values in the range.

For the type *mreal* we use the summary field for the minimal range value *unit_min* and the maximal range value *unit_max* of each real unit. As the unit function is continuous, it is guaranteed that all values in the range $[unit\_min, unit\_max]$ are assumed. Hence for each unit we have an interval, and the task is to compute the union of all these intervals as a set of disjoint intervals. This can be done by sorting the end points of intervals and then sweeping along this one-dimensional space, maintaining a counter to keep track of whether the current position is covered or not, in $O(m \log m)$ time.

The projection of a moving point into the plane may consist of points and of lines; these can be obtained separately by the operations **locations** and **trajectory**.

locations. This operation returns the isolated points in the projection of an *mpoint*, as a *points* value. This type of projection is especially useful when the *mpoint* never changes its position, or does it in discrete steps only. The signature is:

locations $\underline{mpoint} \rightarrow \underline{points}$

In a first step we scan all units of the *mpoint* value and compute for each unit the projection of its three-dimensional segment into the xy-plane. As a result, we obtain a collection of line segments and points (the latter given as degenerate line segments with equal end points). This computation takes $O(m)$ time. From this result only the points should be returned, and only those points which do not lie on one of the line segments. Therefore, in a second step we perform a segment intersection algorithm with plane sweep [3] where we traverse the collection from left to right and only insert line segments into the sweep status structure. For each point we test whether there is a segment in the current sweep status structure containing the point. If this is the case, we ignore the point; otherwise the point belongs to the result and is stored (automatically in lexicographical order) in a *points* value. This step and also the total time takes $O((m + k) \log m)$, if k is the number of intersections of the projected segments.

The algorithm described so far is only worthwhile if for $m = p + l$ the number of points p is almost equal to the number of line segments l. Frequently, this will not be the case, and either p is high and l is low, or vice versa. If this information is known, it may be more efficient to check the p points against the l line segments. For that purpose, we scan all units of the *mpoint* value and identify those p units with a constant temporal behavior. Constant point units with a temporally and geometrically connected unit can be ignored. Afterwards the check is performed. Time complexity is then $O(m + p \cdot l)$.

trajectory. This operation computes the more natural projection of a continuously moving point as a *line* value. Its signature is:

trajectory $\underline{mpoint} \rightarrow \underline{line}$

In a first step, we scan all units of the *mpoint* value, ignore those units with three-dimensional segments perpendicular to the xy-plane, and compute for each remaining unit the projection of its three-dimensional segment into the xy-plane. This takes $O(m)$ time. In a second step, we perform a plane sweep algorithm to find all pairs of intersecting, collinear, and touching line segments, and we return a list of intersection-free segments. This needs $O(m' \log m)$ where $m' = m + k$ and k is the number of intersections in the projection. Note that $k = O(m^2)$. In a third step, we insert the resulting segments into a *line* value. Since sorting is necessary for this, $O(m' \log m')$ time is required which is also the total time needed for this algorithm, which can be as bad as $O(m^2 \log m^2)$ in terms of parameter m, the number of units. However, in most cases the projection will not have a quadratic number of intersections.

traversed. This operation computes the projection of a moving region into the plane. Its signature is:

traversed $\underline{mregion} \rightarrow \underline{region}$

Let us first consider roughly how to compute the projection of a single region unit into the plane. We

use the observation that each point of the projection in the plane either lies within the region unit at its start time, or is traversed by a boundary segment during the movement. Consequently, the projection is the geometric union of the start value of the region unit and all projections of moving segments of the region unit into the plane.

The algorithm has four steps. In a first step, all region units are projected into the plane. In a second step, the resulting set of segments is sorted, to prepare a plane sweep. In a third step, a plane sweep is performed on the projections in order to compute the segments forming the contour of the covered area of the plane. In a fourth step, a *region* value has to be constructed from these segments. In a bit more detail the algorithm is as follows:

algorithm traversed (mr)
input: a moving region mr (of type *mapping*(*uregion*))
output: a region representing the trajectory of mr
method
  let L be a list of line segments, initially empty;
  for each region unit **do**
    compute the region value r at start time;
    put each line segment of r together with a flag
      indicating whether it is a *left* or *right*
      segment into L (it is a left segment if the
      interior of the region is to its right);
    project each moving segment of the unit into
      the plane and put these also with
      a left/right flag into L;
  endfor;
  sort the (half)segments of L in (x, y)-lexicographical order;
  perform a plane sweep algorithm over the segments in
    L, keep track in the sweep status structure of
    how often each part of the plane is covered by
    projection areas, and write segments belonging
    to the boundary (i.e., segments which separate
    0-areas from c-areas with $c > 0$) into a list L'.
  sort the segments of L' in lexicographical order, and
    insert them into a *region* value
end traversed.

The time complexity of the first step is $O(M)$. The second step needs $O(M \log M)$, the third $O(M' \log M)$ where $M' = M+K$ and K is the number of intersections of segments in the projection. The final step takes $O(R \log R)$ where R is the number of segments in the contour of the covered area. In the worst case we may have $R = \Theta(M')$. Hence, the total time complexity is $O(M' \log M')$.

inst, **val**. For values of *intime* types, these two trivial projection operations yield their first and second component, respectively, in $O(1)$. Signatures considered are:

For $\alpha \in \{$*int*, *bool*, *string*, *real*, *point*, *region*$\}$:

inst	$\underline{i}\alpha$	$\rightarrow$	*instant*
val	$\underline{i}\alpha$	$\rightarrow$	α

4.3. Interaction with Domain/Range

atinstant. This operation restricts the moving entity given as an argument to a specified time instant. The signatures to be considered in our restricted model are shown below:

For $\alpha \in \{$*int*, *bool*, *string*, *real*, *point*, *region*$\}$:

atinstant	$\underline{m}\alpha$	$\times$	*instant*	$\rightarrow$	$\underline{i}\alpha$

For all types the general algorithmic scheme is the one given in section 5.1 of [14]. Namely, to first perform a binary search on the array containing the units to determine the unit containing the argument time instant t and then to evaluate the moving entity at time t. For types *mint*, *mbool*, and *mstring* this is trivial. For types *mpoint* and *mreal* it is simply the evaluation of low degree polynomial(s) at t. For all these types the time needed is $O(\log m)$. For type *mregion*, each moving segment in the appropriate region unit is evaluated at time t to get a line segment. A proper region data structure is then constructed, after a lexicographic sort of halfsegments, in time $O(R \log R)$. The total complexity is $O(\log m + R \log R)$. For more details see the discussion of the algorithm *uregion_atinstant*(u, t) in section 5.1 of [14].

atperiods. This operation restricts the moving entity given as an argument to a specified set of time intervals. Signatures to be considered are:

For $\alpha \in \{$*int*, *bool*, *string*, *real*, *point*, *region*$\}$:

atperiods	$\underline{m}\alpha$	$\times$	*periods*	$\rightarrow$	$\underline{m}\alpha$

For all types it is essentially required to form an intersection of two ordered lists of intervals where in each list binary search is possible. There are three kinds of strategies:

1. Perform a parallel scan on both lists returning those units of a (or parts thereof) whose time interval is contained in time intervals of b. The complexity is $O(m + n)$.
2. For each unit in a perform a binary search on b for its start time. Then scan along b to determine intersection time intervals and produce corresponding copies of this unit. The complexity is $O(m \log n + r)$. A variant is to switch the role of the two lists and hence to obtain complexity $O(n \log m + r)$.
3. For the first interval in b, perform a binary search for the unit s in a containing (or otherwise following) its start time. For the last interval in b, perform a binary search for the unit e in a containing (or otherwise preceding) its end time. Compute q as the number of units between s and e (using the indexes of s and e). This has taken $O(\log m)$ time so far. Now, if $q < n \log m$ then do a parallel scan of b and the range of a between s and e computing result units. Otherwise, first

for each interval in b perform a binary search on a for its start time, and afterwards scan along a to determine intersection time intervals and produce corresponding copies of this unit. The time required is either $O(\log m + n + q)$, if $q < n \log m$, or $O(n \log m + r)$, if $q \geq n \log m$. The total time required is $O(\log m + n + \min(q, n \log m) + r)$, since if $q < n \log m$ then $q = \min(q, n \log m)$ while otherwise $n \log m = \min(q, n \log m)$.
We expect that often m will be relatively large and n and r be small. For example, let $n = 1$ and $r = 0$. In this case, the complexity reduces to $O(\log m)$. On the other hand, if $n \log m$ is large, then the complexity is still bounded by $O(\log m + n + q)$ (note that $r \leq q$) which is in turn bounded by $O(m + n)$ (because $q \leq m$). Hence this strategy gracefully adapts to various situations, is output-sensitive, and never more expensive than the simple parallel scan of both lists of intervals.

For type *mregion* copying into result units is more expensive, providing a complexity of $O(\log m + n + \min(q, n \log m) + R)$, where R is the total number of *msegments* in the result.

initial, **final**. These operations provide the value of the operand at the first and last instant of its definition time, respectively, together with the value of the time itself. Signatures considered are:

For $\alpha \in \{int, bool, string, real, point, region\}$:

initial, **final**	$m\alpha$	$\rightarrow$	$i\alpha$

For all types the first (last) unit is accessed and the argument is evaluated at the start (end) time instant of the unit. The complexity is $O(1)$ but for type *mregion* where $O(R \log R)$ is required to build the *region* value.

present. This operation allows one to check whether the moving value exists at a specified instant or is ever present during a specified set of time intervals. Signatures considered are:

For $\alpha \in \{int, bool, string, real, point, region\}$:

present	$m\alpha$	$\times$	*instant*	$\rightarrow$	*bool*
	$m\alpha$	$\times$	*periods*	$\rightarrow$	*bool*

When the second parameter is an instant, for all types the approach is to perform a binary search on the *deftime* array for the time interval containing the specified instant. Time complexity is $O(\log d)$.

When the second parameter is a period (a set of time intervals), for all types the approach is similar to the one used for **atperiods**. Differences are: (i) instead of using the list of units of the first parameter its *deftime* array is used, (ii) as soon as the result becomes `true` the computation can be stopped (*early stop*), and (iii) no result units need to be reported. Time complexity is, depending on the strategy followed: (i) $O(d + n)$, (ii) $O(d \log n)$ or $O(n \log d)$, (iii) $O(\log d + n + \min(q, n \log d))$. An overall strategy could be to determine q in $O(\log d)$ time and then – since all parameters are known – to select the cheapest among these strategies.

at. The purpose of this operation is the restriction of the moving entity to a specified value or range of values. Signatures considered are:

For $\alpha \in \{int, bool, string, real\}$:
For $\beta \in \{point, points, line, region\}$:

at	$m\alpha$	$\times$	α	$\rightarrow$	$m\alpha$
	$m\alpha$	$\times$	$r\alpha$	$\rightarrow$	$m\alpha$
	mpoint	$\times$	β	$\rightarrow$	*mpoint*
	mregion	$\times$	*point*	$\rightarrow$	*mpoint*
	mregion	$\times$	*region*	$\rightarrow$	*mregion*

The general approach for the restriction to a specified value is based on a scan of each unit of the first argument which is checked for equality with the second argument. For *mbool*, *mint*, and *mstring*, the equality check for units is trivial, while for *mreal* and *mpoint* one needs to solve equations, produce a small constant number of units in output and possibly merge adjacent result units with the same value. In any of the previous cases complexity is $O(m)$.

For *mregion* $\times$ *point*, use the algorithm for the more general case of operation **inside**(*mpoint* $\times$ *mregion*) [14]. The kernel of this algorithm is the intersection between a line in 3D - corresponding to a (moving) point - and a set of trapeziums in 3D - corresponding to a set of (moving) segments. In the order of time, with each intersection the (moving) point alternates between entering and leaving the (moving) region represented by trapeziums and the list of resulting units is correspondingly produced (for more details see section 5.2 of [14]). In this particular case, point b corresponds to a vertical line in 3D (assuming an (x, y, t)-coordinate system as in Figure 3), and the complexity is $O(M + K \log k_{\max})$, where K is the overall number of intersections between moving segments of a and (the line of) point b and $k_{\max}$ is the maximum number of intersections between b and the moving segments of a unit of a.

For the restriction to a specified range of values different approaches are used. For *mbool* it is simply a scan of a's units, with $O(m)$ complexity. For *mint* and *mstring*, a binary search on b's range is performed for each unit of a, with an $O(m \log n)$ complexity.

For *mreal*, for each unit of a find ranges intersecting b by means of a binary search on b (using the lowest value of a given by the *min* field in the current unit) plus a scan along b. For each intersection of the unit function of a with an interval of b return a unit with the same unit function and an appropriately restricted time interval. Complexity is $O(m \log n + r)$. This is illustrated in Figure 5.

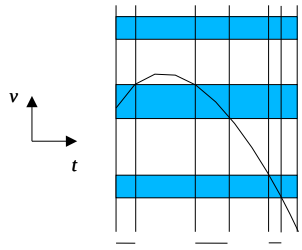

FIGURE 5. The **at** operation on a real unit and a set of real intervals. Three units with time intervals indicated at the bottom are returned.

For *mpoint* × *points*, for each unit of a do a binary search on b with the x-interval of the *unit_pbb* to find the first point of b which is inside that x-interval. Starting from the found point, scan the points of b checking for each of them first if it is in the *unit_pbb* and then whether it intersects the moving point. Sort the resulting units. Complexity is $O(m \log N + \hat{K} + r \log r)$, where $\hat{K}$ is the sum, over all units, of the number of points of b which are inside the x-interval of the respective *unit_pbb*.

Alternative approach (as for case *mpoint* × *line* discussed next): For each unit of a compute the intersection of the *unit_pbb* and of the *object_pbb* of b as a filter. If the bounding boxes intersect, compute the intersection of the moving point with each point of b and then sort the resulting units. Complexity is $O(mN + r \log r)$.

For *mpoint* × *line*, for each unit of a prefilter by intersecting its *unit_pbb* with b's *object mbb* and process intersecting pairs by computing the intersection between the mpoint of a's current unit (a line segment in 3D) and each line segment of b (which is a vertical rectangle in 3D), producing result units corresponding to intersections. Sort the result units. Complexity is $O(mN + r \log r)$.

For *mpoint* × *region*, use the algorithm recalled above for the more general case of operation **inside**(*mpoint* × *mregion*) [14]. That means, initially convert the region value into a region unit, replacing segments by corresponding (vertical) msegments. The complexity is $O(mN + K \log k_{\max})$ where K is the total number of intersections of mpoints (3D segments) in a with msegments in b, and $k_{\max}$ is the maximal number of msegments of b intersected by a single mpoint.

For *mregion* × *region*, use the algorithm for **intersection** in the more general case *mregion* × *mregion* (see Section 6.1).

atmin, atmax. These operations restrict the moving value to the time when it is minimal or maximal. Signatures considered are:

For $\alpha \in \{int, bool, string, real\}$:

atmin, atmax	$m\alpha$	→	$m\alpha$

For all types scan units to see if their value is the minimum (resp. maximum) as given by the *min* (resp. *max*) field of the moving object. For *mreal* the comparison is done with the *unit_min* or *unit_max* summary field. If the unit qualifies, its time interval is reduced to the corresponding instant or interval. Complexity is $O(m)$.

passes. This allows one to check whether the moving value ever assumed (one of) the value(s) given as a second argument. Signatures considered are:

For $\alpha \in \{int, bool, string, real\}$:
For $\beta \in \{point, points, line, region\}$:

passes	$m\alpha$	×	α	→	*bool*
	mpoint	×	β	→	*bool*
	mregion	×	β	→	*bool*

For *mbool*, compare b with index *min* or *max*, with a complexity $O(1)$. For *mint*, *mstring*, and *mreal*, scan each unit (in the latter case use *unit_min* and *unit_max* values) and stop when a matching value is found. Complexity is $O(m)$.

For *mpoint* × β and *mregion* × β, proceed as for the **at** operation, but stop and return *true* as soon as an intersection is discovered. In the worst case complexities are the same as for the **at** operation.

4.4. Rate of Change

The following operations deal with an important property of any time-dependent value, namely its rate of change.

derivative	*mreal*	→	*mreal*
derivable	*mreal*	→	*mbool*
speed	*mpoint*	→	*mreal*
velocity	*mpoint*	→	*mpoint*
mdirection	*mpoint*	→	*mreal*

They all have the same global algorithmic scheme and scan the mapping of the units of the argument moving object a, computing in constant time for each unit of a a corresponding result unit, possibly merging adjacent result units with the same value. The total time needed is $O(m)$. In the sequel we briefly discuss the meaning of the operation and how the result unit is computed from the argument unit.

derivative. This operation has the obvious meaning, i.e., returns the derivative of a moving real as a moving real. Unfortunately in this discrete model it cannot be implemented completely. Recall that a real unit is represented as $u = (i, (a, b, c, r))$ which in turn represents the real function $at^2 + bt + c$ if $r = false$ and the function $\sqrt{at^2 + bt + c}$ if $r = true$, both defined over the interval i. Only in the first case is it possible to represent the derivative again as a real unit, namely the derivative is $2at + b$ which can be represented as a unit $u' = (i, (0, 2a, b, false))$.

In the second case, $r = true$, we assume that the result function is undefined. Since for any moving

object units exist only for time intervals with a defined value, we return no result unit at all.

This partial definition is problematic, but it seems to be better than not offering the operation at all. On the other hand, the user must be careful when applying this function. To alleviate the problem, we introduce next an additional operation **derivable** (not present and not needed in the abstract model of [21]).

derivable. This new operation checks for each unit of a moving real whether or not it describes a quadratic polynomial whose derivative is representable by a real unit. It returns a corresponding boolean unit.

speed. This operation computes the speed of a moving point as a real function. Since each point unit describes linear movement, the resulting real unit contains just a real constant, whose computation is trivial.

velocity. This operation computes the velocity of a moving point as a vector function. Again, due to the linear movement within a point unit, the velocity is constant at all times of the unit's interval $[t_0, t_1]$. Hence, each result unit contains a constant moving point representing the vector function

$$velocity(u,t) = \left(\frac{x(t_1) - x(t_0)}{t_1 - t_0}, \frac{y(t_1) - y(t_0)}{t_1 - t_0} \right)$$

mdirection. We rename the operation **turn** of [21] to **mdirection**, which seems a more appropriate name.[13] For all times of the lifespan of a moving point, it returns the angle between the x-axis and the tangent (i.e., the direction) of a moving point at time t. Due to the linear movement within a point unit, also the direction is constant within the unit's interval.

A special case arises if for two temporally consecutive units u and v two end points coincide, i.e., if, e.g., $x_u(t_1) = x_v(t_0)$ and $y_u(t_1) = y_v(t_0)$. Then $mdirection(v,t)$ (i.e., the direction of the second unit) is assigned to this common end point, in agreement with the formal definition of semantics from [21].

5. ALGORITHMS FOR LIFTED OPERATIONS

In this section we give algorithmic descriptions of *lifted operations*. Recall that these are operations originally defined for non-temporal objects (see Table 2) that are now applied to "moving" variants of the arguments. We consider predicates (Section 5.1), set operations (Section 5.2), aggregation (Section 5.3), numeric properties (Section 5.4), distance and direction (Section 5.5), and boolean operations (Section 5.6).

[13] An operation **turn** which better captures the meaning of the original **turn** operation of [21] is defined in [7]. However, in this discrete model which only has linear movement the operation has no interesting result and is omitted.

5.1. Predicates

isempty. This predicate checks, for each time instant, whether the argument is defined. Signatures considered are:

For $\alpha \in \{\underline{int}, \underline{bool}, \underline{string}, \underline{real}, \underline{point}, \underline{region}\}$:

isempty $\quad \underline{m\alpha} \quad \rightarrow \quad \underline{mbool}$

We remark that the result is defined from *mininstant* to *maxinstant* (the "bounds" of time introduced at the beginning of Section 3.2). Scan the *deftime* index returning units with *true* value for intervals where a is defined and units with *false* value in the other case. Complexity is $O(d)$ where d is the size of the *deftime* index.

$=, \neq$. These predicates check for equality of the arguments over time. Signatures considered are:

For $\alpha \in \{\underline{int}, \underline{bool}, \underline{string}, \underline{real}, \underline{point}, \underline{region}\}$:

(1) $\quad \underline{m\alpha} \times \alpha \rightarrow \underline{mbool}$
(2) $\quad \underline{m\alpha} \times \underline{m\alpha} \rightarrow \underline{mbool}$

The general approach for operations of group (1) (whose second argument is of a non-temporal type), is based on a scan of each unit of the first argument which is checked for equality with the second argument. The equality check for all cases but $\underline{mregion}$ is done as in the corresponding cases for the **at** operation (see Section 4.3), except that a boolean unit is returned, and so complexities are the same. For $\underline{mregion}$ the equality check for units is done as follows. First check whether u and N are equal numbers. If not we report a single unit $(i, false)$ where i is the time interval of the argument unit. Otherwise proceed as follows:

repeat
  take a moving segment s of the current unit of a;
  if s is *static* (i.e. does not change in time) **then**
    search for a matching segment in b; (*)
    if the search fails **then return** $(i, false)$ **endif**
  endif
until s is not *static* or s is the last segment;
if s is the last segment **then return** $(i, true)$ **endif**;
compare s with all segments of b, finding k intersections
  $t_1, \ldots, t_k$ such that s is equal to a segment of b;
if $k = 0$ **then return** $(i, false)$ **endif**;
for each t_i
  for each msegment s of the current unit
    evaluate s at time t_i;
    search for a matching segment in b; (*)
    if no segment was found **then exit**
  endfor;
  if the loop terminated normally **then**
    return $(t_i, true)$ **endif**
endfor;
return $(i, false)$

This algorithm is based on the observation that if the region unit has a single moving segment, then it can

be equal to a static region only in a single instant of time. – Steps labeled (*) take time $O(\log N)$ because halfsegments in the *region* representation are ordered lexicographically. The worst case complexity per unit is $O(kN \log N)$.[14] In the worst case $k = O(N)$ but in most practical cases, k is a small constant. Assuming the latter, the total complexity is $O(mN \log N)$. In fact, in practice in almost all cases during the evaluation of a unit an early stop will occur so that most units will be evaluated in $O(1)$ time, and if the moving region is never equal to the static region, this can be determined in $O(m)$ time.

The general approach for operations of group (2) is based on a parallel scan of units of the refinement partition. Each pair of units is checked for equality. For *mbool*, *mint*, and *mstring*, such a check is trivial. For *mreal* and *mpoint* one first checks whether coefficients are the same: if so, produce the output unit, otherwise intersect the curves and produce output units (at most a small constant number). In any of the previous cases, complexity is $O(p)$.

For *mregion* process each pair of units with time interval i of the refinement partition as follows:

if u and v are different **then return** $(i, false)$ **endif**;
do a parallel scan of the lists of moving segments to find
  a pair (s_1, s_2) of different segments;
if no pair of different segments is discovered **then**
  return $(i, false)$
else
  let s' be the smaller segment among s_1 and s_2 {this
    segment is guaranteed not to appear in the other list};
  compare s' with the remaining segments of the other list
    finding k intersections $t_1, \ldots, t_k$ with equality
endif;
if $k = 0$ **then return** $(i, false)$ **endif**;
for each t_i
  evaluate both units at time t_i;
  sort the obtained segments and do a parallel scan to
    check for equality, stopping early if a non-matching
    pair of segments is found, and otherwise returning
    an appropriate result unit (*)
endfor

Noting that the step labeled (*) requires time $O(u \log u)$, per unit time complexity is $O(ku \log u)$. In the worst case $k = O(u)$ but in most practical cases, k is a small constant. Assuming the latter, the total complexity is $O(p(u_{\max} \log u_{\max}))$. Again, if the two moving regions are never equal, then a pair of units will almost always be handled in $O(1)$ time, and the total time will be $O(p)$.

intersects. This predicate checks whether the arguments intersect each other. Signatures considered are:

intersects	*points*	×	*mregion*	→	*mbool*
	region	×	*mregion*	→	*mbool*
	line	×	*mregion*	→	*mbool*
	mregion	×	*mregion*	→	*mbool*

[14] Probably one can show that it is even bounded by $O(N \log N)$, but we have not yet proved that.

For *points* × *mregion* use the corresponding algorithm for the **inside** predicate.

In Section 6 an algorithm for the case *mregion* × *mregion* is described in detail. This algorithm can be easily specialized to the cases *region* × *mregion* and *line* × *mregion*, the latter due to the fact that the algorithm does not require that a's p-faces form polyhedra in 3D space (the term *p-face* is introduced in Section 6).

inside. This predicate checks if a is contained in b. Signatures considered are:

inside	*mregion*	×	*points*	→	*mbool*
	mregion	×	*line*	→	*mbool*
	mpoint	×	*region*	→	*mbool*
	point	×	*mregion*	→	*mbool*
	mpoint	×	*mregion*	→	*mbool*
	points	×	*mregion*	→	*mbool*
	mpoint	×	*points*	→	*mbool*
	mpoint	×	*line*	→	*mbool*
	line	×	*mregion*	→	*mbool*
	region	×	*mregion*	→	*mbool*
	mregion	×	*region*	→	*mbool*
	mregion	×	*mregion*	→	*mbool*

In the first two cases the result of the operation is always *false*.

For *mpoint* × *region* and *point* × *mregion* use the more general algorithm for case *mpoint* × *mregion* briefly described in Section 4.3 and detailed in [14]. Complexity is $O(\hat{N} + K \log k_{\max})$.

For *points* × *mregion*, for each of the points of a use the algorithm for the case *point* × *mregion*. Complexity is $O(M(N + K \log k_{\max}))$.

For *mpoint* × *points* consider each unit of a, and for each point of b check if the moving point passes through the considered point. If so, produce a unit with *true* value at the right time instant. Sort all produced units by time and then add remaining units with *false* value. Complexity is $O(mN + r \log r)$.

The case *mpoint* × *line* is similar to the previous one, but you have also to consider that if the projection of a moving segment overlaps with a segment of b the corresponding result unit is defined on a time interval rather than a single instant.

For *line* × *mregion*, *region* × *mregion* and *mregion* × *region* proceed as in the more general case *mregion* × *mregion*, described in detail in Section 6.

$<, \leq, \geq, >$. These predicates check the order of the two arguments. Signatures considered are:

For $\alpha \in \{$*int*, *bool*, *string*, *real*$\}$:

$<, \leq, \geq, >$	α	×	$m\alpha$	→	*mbool*
	$m\alpha$	×	α	→	*mbool*
	$m\alpha$	×	$m\alpha$	→	*mbool*

Algorithms are analogous to those for operation =.

5.2. Set Operations

We recall that for set operations a regularized set semantics is adopted. For example, forming the union of a *region* and a *points* value yields the same *region* value, because a region cannot contain isolated points.

intersection. Computes the intersection of the arguments. Signatures considered are:

For $\alpha \in \{int, bool, string, real, point\}$:
For $\beta \in \{points, line, region\}$:

(1)	$m\alpha$	$\times$	α	$\rightarrow$	$m\alpha$
	mpoint	$\times$	β	$\rightarrow$	*mpoint*
	mregion	$\times$	*point*	$\rightarrow$	*mpoint*
	mregion	$\times$	*region*	$\rightarrow$	*mregion*
(2)	$m\alpha$	$\times$	$m\alpha$	$\rightarrow$	$m\alpha$
	mpoint	$\times$	*mregion*	$\rightarrow$	*mpoint*
	mregion	$\times$	*mregion*	$\rightarrow$	*mregion*

For all signatures of group (1) use the corresponding algorithms for operation **at** (see Section 4.3).

For the signatures of group (2) (both arguments are moving ones and belong to the same point type) do a parallel scan of the refinement partition units and for time intervals where the values of the argument are the same, produce a result unit with such a value. For the cases *mpoint* and *mreal* this requires solving equation(s). In any case the complexity is $O(p)$.

The algorithm for case *mpoint* $\times$ *mregion* is analogous to the corresponding one for the **inside** operation (see Section 5.1) but for reporting point units with the same value as a instead of boolean units with *true* value and no unit instead of boolean units with *false* value.

For *mregion* $\times$ *mregion* the algorithm is rather complex and is described in Section 6.

union. Computes the union of the arguments. Signatures considered are:

union	*mpoint*	$\times$	*region*	$\rightarrow$	*mregion*
	mpoint	$\times$	*mregion*	$\rightarrow$	*mregion*
	point	$\times$	*mregion*	$\rightarrow$	*mregion*
	mregion	$\times$	*region*	$\rightarrow$	*mregion*
	mregion	$\times$	*mregion*	$\rightarrow$	*mregion*

For *mpoint* $\times$ *region*, the result is region b for all times for which a is defined (due to the regularized set semantics). Hence d corresponding region units have to be constructed, getting time intervals from scanning the *deftime* index of a. Since sorting is required once to put msegments in the region units into the right order, the complexity is $O(dN + N \log N)$.

For *mpoint* $\times$ *mregion* and *point* $\times$ *mregion* simply return b as result.

For *mregion* $\times$ *region* use the more general algorithm for the case *mregion* $\times$ *mregion*, which is described in Section 6.

minus. Computes the difference of a and b. Signatures considered are:

For $\alpha \in \{int, bool, string, real, point\}$:
For $\beta \in \{points, line, region\}$:

(1)	$m\alpha$	$\times$	α	$\rightarrow$	$m\alpha$
	α	$\times$	$m\alpha$	$\rightarrow$	$m\alpha$
	$m\alpha$	$\times$	$m\alpha$	$\rightarrow$	$m\alpha$
	mpoint	$\times$	β	$\rightarrow$	*mpoint*
	point	$\times$	*mregion*	$\rightarrow$	*mpoint*
	mpoint	$\times$	*mregion*	$\rightarrow$	*mpoint*
(2)	*region*	$\times$	*mpoint*	$\rightarrow$	*mregion*
	mregion	$\times$	*point*	$\rightarrow$	*mregion*
	mregion	$\times$	*mpoint*	$\rightarrow$	*mregion*
	mregion	$\times$	*points*	$\rightarrow$	*mregion*
	mregion	$\times$	*line*	$\rightarrow$	*mregion*
(3)	*mregion*	$\times$	*region*	$\rightarrow$	*mregion*
	region	$\times$	*mregion*	$\rightarrow$	*mregion*
	mregion	$\times$	*mregion*	$\rightarrow$	*mregion*

For all cases where the type of a is a point type (group (1)), algorithms are similar to those for **intersection**, except for the production of result units. Complexities are the same as corresponding algorithms for **intersection**.

Algorithms for cases in group (2) are trivial due to the regularized set semantics. For *region* $\times$ *mpoint* one simply transforms a into a moving region defined on the same definition time as b, with a complexity $O(dM + M \log M)$ (as discussed above for **union**(*mpoint* $\times$ *region*)), while for *mregion* $\times$ *point*, *mregion* $\times$ *mpoint*, *mregion* $\times$ *points*, *mregion* $\times$ *line* one simply returns a as the result.

For *mregion* $\times$ *region* and *region* $\times$ *mregion* use the algorithm for the more general case *mregion* $\times$ *mregion*, described in Section 6.

5.3. Aggregation

Aggregation in the unlifted mode reduces sets of points to points. In the lifted mode it does this for all times of the lifespan of a moving object. In our reduced type system we only have to consider moving regions.

center. This operation computes the center of gravity of a moving region over its whole lifespan as a moving point. The signature is:

center	*mregion*	$\rightarrow$	*mpoint*

The algorithm scans the mapping of region units. Because a region unit develops linearly during the unit interval $i = [t_0, t_1]$, the center of gravity also evolves linearly and can be described as a point unit. It is therefore sufficient to compute the centers of the regions at times t_0 and t_1 and to determine the pertaining linear function afterwards.

For computing the center of a region we first triangulate all faces of the region. This can be done on the basis of [15] in time $O(u \log u)$ and results in $O(u)$ triangles. For each triangle in constant time we compute its center viewed as a vector and multiply this vector by the area of the triangle. For all triangles we sum up these weighted products and divide this sum by the sum of all weights, i.e., the areas of all triangles.

The resulting vector is the center of the region. Note that it may lie outside of all faces of the region. The time complexity for computing the center is $O(u \log u)$.

For each region unit, by interpolation between the centers at its start and end times a corresponding point unit is determined. The total time for the center operation on a moving region is $O(M \log u_{max})$.

5.4. Numeric Properties

These operations compute some lifted numeric properties for moving regions.

no_components	*mregion*	→	*mint*
perimeter	*mregion*	→	*mreal*
area	*mregion*	→	*mreal*

Here **no_components** returns the time dependent number of components (i.e., faces) of a moving region as a moving integer, and **perimeter** and **area** the respective quantities as moving reals.

The algorithmic scheme is the same for all three and very simple: Scan the sequence of units and return the value stored in the respective summary field *unit_no_components*, *unit_perimeter*, or *unit_area*, possibly merging adjacent units with the same unit function. This requires $O(m)$ time for m units.

The values for the summary fields are computed when their region unit is constructed. The *unit_no_components* value is determined as a by-product when the structure of faces within the unit is set up (see Section 3).

For the *unit_perimeter* function, we observe that the boundary of a region unit consists of moving segments; for each of them the length evolves by a linear function. Hence the perimeter, being the sum of these lengths, also evolves by a linear function. The perimeter function can be computed by either summing up the coefficients of all the moving segments' length functions, or by linear interpolation between the start and end time perimeter of the unit.

For the *unit_area* function, the computation is slightly more complex. The area of a simple static polygon (a cycle) c consisting of the segments $s_0, \ldots, s_{n-1}$ with $s_i = ((x_i, y_i), (x_{(i+1) \bmod n}, y_{(i+1) \bmod n}))$ may be determined by calculating the areas of the trapezia under each segment s_i down to the x-axis[15] and by subtracting the areas of the trapezia under the segments at the bottom of the cycle from the areas of the trapezia under the segments at the top of the cycle. We can express this by the formula

$$area(c) = \sum_{i=0}^{n-1} (x_{(i+1) \bmod n} - x_i) \frac{y_{(i+1) \bmod n} + y_i}{2}$$

Note that if cycles are connected clockwise, then in this formula top segments will yield positive area contributions and bottom segments negative ones, as desired. Hence, the formula computes correctly a positive area value for outer cycles (see Section 3). Indeed, for hole cycles (represented in counter-clockwise order) it computes a negative value which is also correct, since the areas of hole cyles need to be subtracted from the region area. That means, we can simply compute for all cycles of a region their area according to the formula above and form the sum of these area contributions to determine the area of the region.

In a region unit where we have moving segments, we can replace each x_i and each y_i by a linear function. For a moving unit cycle c we therefore have

$$area(c, t) = \sum_{i=0}^{n-1} (x_{(i+1) \bmod n}(t) - x_i(t)) \frac{y_{(i+1) \bmod n}(t) + y_i(t)}{2}$$

Each factor in the sum is the difference respectively sum of two linear functions. Hence, it is a linear function again, and therefore the product is a quadratic polynomial. The sum of all quadratic polynomials is a quadratic polynomial as well. Again we can sum up the area function contributions over all moving cycles of a region unit, to get the area function for the unit.

The cost of computing *unit_perimeter* and *unit_area* fields is clearly linear in the size of the unit, i.e., $O(u)$ time. In all cases, it is dominated by the remaining cost for constructing the region unit.

5.5. Distance and Direction

In this subsection we discuss lifted distance and direction operations.

distance. The distance function determines the minimum distance between its two argument objects for each instant of their common lifespan. The pertaining signatures are:

For $\alpha, \beta \in \{$*point*, *region*$\}$:

distance	*mreal*	×	*real*	→	*mreal*
	mreal	×	*mreal*	→	*mreal*
	$m\alpha$	×	β	→	*mreal*
	$m\alpha$	×	$m\beta$	→	*mreal*

For all function instances the algorithm scans the mapping of the units of the moving object(s) and returns one or more real units for each argument unit.

The computation of the distance between an *mreal* value and a *real* value s leads to several cases. If $ur = (i, (a, b, c, r)) \in$ *ureal* with $i = [t_0, t_1], t_0 < t_1$, and $r =$ *false*, the unit function of ur describes the quadratic polynomial $at^2 + bt + c$. The distance between ur and s is then given by the function $f(t) = at^2 + bt + c - s$, which is a quadratic polynomial, too. Unfortunately, this function usually does not always yield a positive value for all $t \in i$, as required in the definition of

[15] This assumes that y-values are positive. If they are not, one can instead form trapeziums by subtracting a sufficiently negative y-value.

distance. Therefore it is necessary to determine the instants of time when $f(t) = 0$ and to invert the value of the function in those time intervals when it is negative. This is illustrated in Figure 6. To program this, one

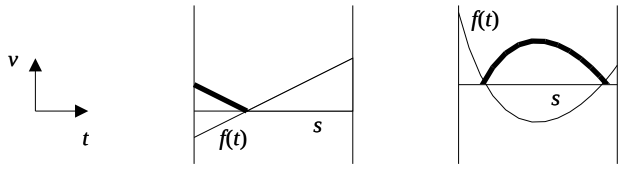

FIGURE 6. Computing the "distance" between a real function and a real constant

needs to distinguish various cases, which is a bit tedious. In any case we obtain as a result either one, two, or three new real units.

If $r = true$, the function of ur describes the square root polynomial $\sqrt{at^2 + bt + c}$. The distance between ur and s is then given by the function $\sqrt{at^2 + bt + c} - s$. Unfortunately, this term is not expressible by a square root polynomial and thus not by a real unit. Hence, strictly speaking, this operation is not implementable within this discrete model.

Similarly as discussed above for the **derive** operation, we believe it is better to offer a partial implementation than none. Hence, for square root polynomial units we consider the result as undefined and return no unit at all (again, as for derive). The **derivable** operation can also here be used to check for which part of the argument the result could be computed.

In both cases, the time complexity is $O(1)$ per unit and $O(m)$ for a moving real.

The algorithm for computing the distance between two *mreal* values is similar to the previous one, because a *real* value in the above context can be regarded as a "static" moving real. The difference is that first a refinement partition of both moving reals has to be computed which takes $O(m + n)$. If $ur = (i, (a, b, c, r))$ and $vr = (i, (d, e, f, s))$ are corresponding real units of both refined moving reals with $r = s = false$, their distance is given by the quadratic polynomial $(a-d)t^2+(b-e)t+(c-f)$ which has to be processed as in the previous algorithm. If $r = true$ or $s = true$, no unit is returned. The time complexity of this algorithm is $O(m + n)$.

We now consider the case of an *mpoint* value and a *point* value $p = (x', y')$ with $x', y' \in$ `real`. If $up = (i, (x_0, x_1, y_0, y_1)) \in$ *upoint* with $x_0, x_1, y_0, y_1 \in$ `real`, the evaluation of the linearly moving point at time t is given by $(x(t), y(t)) = (x_1 t + x_0, y_1 t + y_0)$. Then the distance is

$$\begin{aligned} &distance((up, p), t) \\ &= \sqrt{(x(t) - x')^2 + (y(t) - y')^2} \\ &= \sqrt{(x_1 t + x_0 - x')^2 + (y_1 t + y_0 - y')^2} \end{aligned}$$

Further evaluation of this term leads to a square root of a quadratic polynomial in t which is returned as a real unit. The time complexity for a moving point and a point is $O(m)$.

The distance calculation between two *mpoint* values requires first the computation of the refinement partition in $O(m + n)$ time. The distance of two corresponding point units up and vp is then determined similarly as in the previous case and results again in a square root of a quadratic polynomial in t which is returned as a real unit. This algorithm requires $O(m + n)$ time.

The remaining operation instances can be grouped according to two algorithmic schemes. The first algorithmic scheme, which is described in Section 6.2.1, relates to the distance computation between a moving point and a region, between a moving point and a moving region, and between a moving region and a point. The second algorithmic scheme, which is described in Section 6.2.2, refers to the distance computation between a moving region and a region as well as between two moving regions. The grouping is possible, because the spatial argument objects can be regarded as "static" spatio-temporal objects. Therefore, the first algorithmic scheme deals with the distance between a moving point and a moving region, and the second algorithmic scheme deals with the distance between two moving regions.

direction. This operation returns the angle of the line from the first to the second point at each instant of the common lifespan of the argument objects.

direction	*mpoint*	×	*point*	→	*mreal*
	point	×	*mpoint*	→	*mreal*
	mpoint	×	*mpoint*	→	*mreal*

Unfortunately, the result of these operation instances cannot be represented as a moving real, because their computation requires the use of the arc tangent function. This can be shown as follows: given two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$, the slope between the horizontal axis and the line through p and q can be determined by $\tan\alpha = \frac{y_2 - y_1}{x_2 - x_1}$. Thus $\alpha = \arctan\frac{y_2 - y_1}{x_2 - x_1}$ holds. We can continue this to the temporal case. For two point units (after the calculation of the refinement partition) as well as for a point unit and a *point* value, this leads to $\alpha(t) = \arctan\frac{y_2(t) - y_1(t)}{x_2(t) - x_1(t)}$ and $\alpha(t) = \arctan\frac{y_2(t) - y_1}{x_2(t) - x_1}$ respectively. Consequently, this operation is not implementable in this discrete model.

5.6. Boolean Operations

Boolean operations are included in the scope of operations to be temporally lifted.

and, **or**. These operators represent the lifted logical conjunction respectively disjunction connectives. Their signatures are:

and, **or** $\quad \underline{mbool} \times \underline{bool} \to \underline{mbool}$
$\underline{mbool} \times \underline{mbool} \to \underline{mbool}$

For the first operator instance we scan all boolean units in a and to evaluate for each unit the unlifted logical connective applied to its boolean value and to b, returning a corresponding unit. Time complexity is $O(m)$. For the third operator instance we compute the refinement partition and then proceed in the same way for pairs of units. Time complexity is $O(p)$.

not. This operation is the lifted logical negation operator. Its signature is

not $\quad \underline{mbool} \to \underline{mbool}$

Here one just scans the units, negating their values, in $O(m)$ time.

6. SELECTED ALGORITHMS

In this section we describe some of the more complex algorithms. Section 6.1 develops a single uniform algorithm for the operations **intersection**, **union**, and **difference** on moving regions. Predicates **intersects** and **inside** can be implemented by variants of this algorithm. Section 6.2 considers the computation of distances between moving points and moving regions.

6.1. Algorithm for Set Operations on Moving Regions

6.1.1. Overall Algorithm Description

We describe an algorithm to compute for two given moving regions a and b their union, intersection, or difference. As for algorithms of Sections 4 and 5, we first compute a refinement partition of the two argument mappings. Then we consider in turn each pair of region units defined on the same unit interval. To simplify exposition, we describe the algorithm referring just to a pair of *uregion* unit functions $\bar{a}$ of a and $\bar{b}$ of b.

We recall that each *uregion* is a set of moving segments (Figures 3 and 4). Each moving segment defines a polygonal face (either a triangle or a trapezium) in the 3D space (x, y, t) which we call *p-face* to avoid confusion with faces of a *uregion*. For the purpose of description we view $\bar{a}$ and $\bar{b}$ as two sets of p-faces.

The intersection of a p-face f of $\bar{a}$ with another p-face g of $\bar{b}$ is a segment s in the 3D space, lying within both f and g. Segment s can be computed by (i) computing the supporting planes P_f and P_g of the two p-faces in the 3D space and (ii) clipping their intersection (a line) by both p-faces. Since both p-faces are convex, the result can only be a single segment.

Depending on the operation type (i.e., union, intersection, or difference) different parts of f and g will be selected as a result p-face. Figure 7 below shows two intersecting p-faces f and g: the viewpoint is in the direction of the intersection segment s. The short

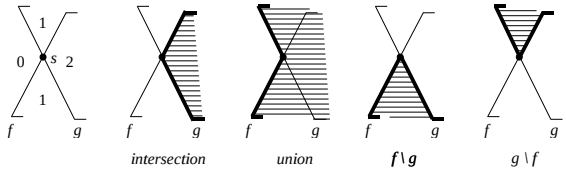

FIGURE 7. Cases for two intersecting p-faces f and g

rightwards pointing bars at the end of f and g indicate on which side of them their interior is. The numbers in the leftmost drawing indicate how many times space is covered by the two region units. Hence, 0 denotes space outside both region units (0-covered), 1 the space within exactly one of the two (1-covered) and 2 the one within both (2-covered). Intersection asks for the boundary of the 2-covered zone, union for the boundary of the 0-covered zone, and difference for the boundary of one of the 1-covered zones $f \setminus g$ or $g \setminus f$. Given one of the participating p-faces, say f, and the intersection segment s, one can therefore determine, for each of these operations, on which side of s the part of f contributing to the result will be.

The algorithm works in three steps. In the first one, all segments which are on the boundary of a result p-face are computed. In the second one, the segments facing each other on opposite sides of a result p-face are suitably linked together. In the third one, result region units are produced by computing for each of them the proper structure in terms of cycles of moving segments and their nesting, to compose faces.

Step 1. Computing Result Segments. The first step of the algorithm computes for each pair of p-faces their intersection segment. For each p-face f we store in a list L_f each intersection segment lying within it together with an indication on which side of s the result p-face is (see Figure 8 (left)).

FIGURE 8. s is an intersection segment lying on p-face f

The first step can be described as follows:

for each p-face f in $\bar{a}$ **do**
  for each p-face g in $\bar{b}$ **do**
    if f intersects g **then**
      mark f and g as having an intersection;
      compute the intersection segment s;
      append s to L_f together with an indication on which side of s the result part of f lies;
      append s to L_g together with an indication on which side of s the result part of g lies;

endif
endfor
endfor;

Step 2. Finding Mates. Now consider a p-face f of one of the two input region units, say $\bar{a}$. If f is marked as not having intersections with p-faces of $\bar{b}$, then it is either entirely inside or entirely outside $\bar{b}$. Depending on this fact and on the considered operation, either f is a whole p-face of the result object or no part of f is a p-face of the result object. On the other hand, if f does have intersections with p-faces of $\bar{b}$, the intersection segments lying within it, possibly together with (part of) the boundary of f, form boundaries of a (set of) result p-face(s), as shown in Figure 8 (right).

For the p-face f, its *face normal vector* is perpendicular to the face and points to the outside of the region unit. The face normal vector can easily be computed initially for each p-face (msegment) based on the cyclic order in which msegments are linked.

For the result p-faces, we should also keep track on which side the result region is. This depends on the operation: For **union**, **intersection**, and $\bar{a} \setminus \bar{b}$, result p-faces have their normal vectors pointing in the same direction as f; for $\bar{b} \setminus \bar{a}$ they point in the opposite direction. One can easily check this considering the result p-faces in Figure 9 (left) lying on p-face f under various operations.

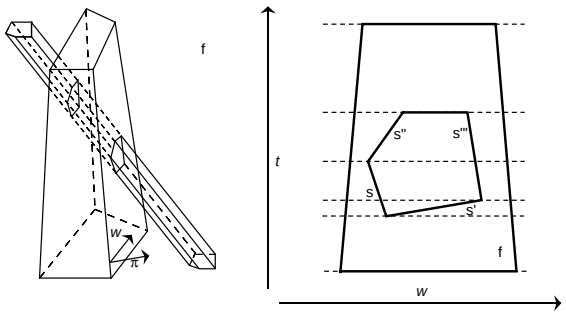

FIGURE 9. Classification of left and right segments and their mating

In general, result p-faces are not just triangles or trapeziums, as it is required to form moving segments. For this reason it is necessary to cut them at suitable points on the t-axis (see Figure 9 (right)), together with every other p-face (possibly lying within an input p-face other than f) whose projection on the t-axis intersects these cutting points.

To determine a coordinate system for the face f, let w be the axis orthogonal to both the t-axis and the result p-face's normal vector π (hence lying in the result p-face's supporting plane), directed so that a rotation of 90 degrees bringing a halfline from the position of the positive t semi-axis to the position of the positive w semi-axis is seen from the positive π semi-axis as a clockwise rotation. See again Figure 9 (left), where the direction of w is explicitly drawn.

Let us call *left* (resp., *right*) boundary of $f \in \bar{a} \cup \bar{b}$ the segment, of the two ones not orthogonal to the t-axis, such that the value $w(\bar{t})$ of its w coordinate at a given value $\bar{t}$ of the t coordinate is lower (resp., greater) than the corresponding value $w(\bar{t})$ of the other segment (see once more Figure 9 (right)).

We now determine which parts (if any) of the left and right boundaries of f are also boundaries of some result p-face. For example, in Figure 8 (right), the left boundary of f is divided into two segments of which the lower one is part of the boundary of the result, while the whole right boundary of f is also part of the boundary of the result. This task is easy to accomplish using information computed in the first step of the algorithm (note that for this computation intersection segments of f orthogonal to the t-axis *are* needed, for details see subsection 6.1.2).

Let S_f be the set of intersection segments of f not orthogonal to the t-axis plus the set of segments corresponding to the parts of f's boundary computed above. We now extend the classification in terms of left and right segments to each $s \in S_f$ as follows: s is a *left* (resp., *right*) boundary of the result p-face if a line in the t-w plane, parallel to the w-axis and directed accordingly to it, encounters s after an even (resp., odd) number of other intersections. In Figure 10, where bold lines denote the boundary of the result p-face lying in p-face f in case of union between f and g, the line denoted with h encounters first the left boundary of f, second, the intersection segment s, which is then classified as right, third, s''', classified as left, and finally, the right boundary of f, classified as right.

We also *mate* each left (resp., right) segment $s \in S_f$ with a (list of) right (resp., left) segments according to the following rule: two segments s and t are mated iff a segment h lying in the plane t-w and orthogonal to the t-axis exists such that h intersects both s and t and is completely inside the result p-face. As an example, consider again Figure 10 where segments h' and h''' allow to mate, respectively, the left boundary of f with s and s''' with the right boundary of f. The same drawing may be used to check the case of intersection between f and g, where the result p-face is only the smaller pentagonal polygon inside the trapezium, since in this case segment h'' allows to mate intersection segments s and s'''.

To do the mating, we perform a plane sweep, in order of increasing t values, of the set S_f. A time instant t_e is *relevant* for the plane sweep if a segment $s \in S_f$ has the t value of one of its endpoints equal to t_e. A boundary segment s is *active* at time t_e if it intersects a sweep plane at position t_e. During the sweep we maintain a dictionary D_f of active boundary segments sorted according to their current w coordinate. When a new segment s is inserted into D_f we check, if it exists, its immediate predecessor s' along the w-axis. If s' does

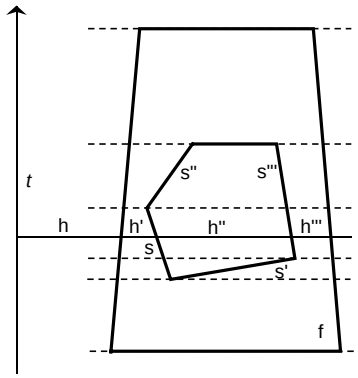

FIGURE 10. Classification of left and right segments and their mating

not exist or is a right boundary, we mark s as a left boundary. Otherwise we mark s as a right boundary and append s and s' to each other' lists of mates. Note that in this way for each segment its list of mates will be ordered from bottom to top, i.e., in increasing t-order.

In more detail, the second step of the algorithm proceeds as follows:

```
for each p-face f in ā do
  if f is marked as having intersections then
    insert from L_f into S_f intersection segments of f not
      orthogonal to the t-axis
    compute and insert into S_f parts of the left (resp.
      right) boundary of f contributing to a result p-face
      {see subsection 6.1.2}
  else {f is either entirely inside or entirely outside b̄}
    check whether f is inside b̄; {see subsection 6.1.2}
    if f is inside b̄ then
      case union, difference ā \ b̄: S_f := ∅
        {f does not contribute to result}
      case intersection, difference b̄ \ ā: insert into S_f
        the entire left and right boundaries of f
    else {f outside b̄}
      case union, difference ā \ b̄: insert into S_f
        the entire left and right boundaries of f
      case intersection, difference b̄ \ ā: S_f := ∅
    endif {f inside b̄}
  endif {f intersect b̄}
  sort S_f lexicographically by (t, w) coordinates of
    lower end points
  initialize D_f to empty
  for each relevant time value t_e
    let New_te be the sublist of S_f with t = t_e
    perform a parallel scan of D_f and New_te,
      removing from D_f all segments whose uppermost
      endpoint has t = t_e and inserting into it segments
      from New_te
    for each segment of New_te encountered during this scan:
      let s' be the immediate predecessor of s (along the
        w-axis) in D_f
      if s' does not exist or s' is right boundary then
        mark s as left boundary
      else {s' is left boundary}
        mark s as right boundary
        append s' to the list of the mates of s
        append s to the list of the mates of s'
      endif
    endfor
    if the last segment s of New_te is a left boundary then
      let s' be the immediate successor of s in D_f
      append s' to the list of the mates of s
      append s to the list of the mates of s'
    endif
  endfor
endfor;
for each p-face g in b̄ do
  (analogous)
endfor;
```

Step 3. Computing Result Region Units. To produce result region units we merge together the sets S_f, for all $f \in \bar{a} \cup \bar{b}$, in a list S_T. Note that each segment s lies within both a p-face f of $\bar{a}$ and a p-face g of $\bar{b}$, and is part of both a result p-face h_{sf} coplanar with f and a result p-face h_{sg} coplanar with g. In particular s is a left boundary of one of h_{sf} and h_{sg} and is a right boundary of the other. This means that there are two distinct instances of s, one stored in S_f and the other in S_g: we say they are each other's *buddy*. Buddies store different lists of mates and we need to couple them to properly reconstruct cycles of moving segments in the result region units.

Therefore, we sort segments in S_T according to a (t, x, y)-lexicographical order on their lower end points. Since t is the most significant coordinate, after the sorting buddies appear consecutively in S_T, and with a linear scan we retain in S_T for each segment only one of the two buddies, but with both t-ordered lists of mates.

Next we perform a bottom to top plane sweep of the 3D space (i.e., in order of increasing t coordinate). During the sweep, a list A of segments currently intersecting the sweep plane (*active* segments), is maintained. In each step, the plane sweep advances from one relevant value of time, t', to the next t''. At time t'', the following actions are performed: (i) the list A of active segments is traversed, and a region unit for the time interval $[t', t'')$ is constructed from them (see below, how), (ii) during the traversal, segments whose top point has time coordinate t'' are removed from A, and (iii) segments whose bottom point has coordinate t'' are appended to A.

Note that at time t with $t' < t < t''$ for each active segment s lying within p-faces f of $\bar{a}$ and g of $\bar{b}$, exactly one of its mates lying within f and one of its mates lying within g are also active segments. We call them the *active mates* of s at time t. Then we use active segments to construct a result *uregion* as follows. For each active segment s we produce the moving segment m connecting s with its mate s', of the two active ones, such that m has the interior of the *uregion* to its right, when traversed from s to s'. Then m is inserted into a lexicographically sorted list MS of moving segments of the current result *uregion*, and linked, to form a

cycle, with moving segments produced while visiting the active mates of s if these have been already inserted in MS.

When all active segments have been processed, MS contains all moving segments of the current result region unit linked into cycles. It only remains to connect outer cycles of the region unit's faces with the associated hole cycles (i.e. to construct the faces array of the *uregion*). First of all, for each cycle we determine whether it is an outer or a hole cycle with a linear scan of its moving segments (for details see subsection 6.1.2). Then, for each hole cycle c we perform a plumbline algorithm scanning moving segments of outer cycles and sort intersection points to find the outer cycle directly enclosing c.

In more detail, the third step of the algorithm proceeds as follows:

```
S_T := ∅;
for each face f of ā ∪ b̄
  insert all segments in S_f into S_T
endfor;
(t, x, y)-lexicographically sort segments of S_T;
scan through S_T to couple buddies, delete one of them,
  and store both lists of mates in a single representative;
let A be the list of active segments; A := ∅;
scan S_T and append to A all segments with bottom
  t-coordinate t_0;
from now on scan S_T and process in each step a set of
  segments with bottom t-coordinate t_e:
for each relevant time value t_e
  initialize to empty the list of moving segments MS;
  scan the list A:
  for each active segment s in A
    let s_1 and s_2 be the active mates of s;
    produce the moving segment m connecting s with its
      active mate s_j such that m has the interior of the
      uregion to its right, when traversed from s to s';
    insert m into MS;
    for i = 1, 2
      if while previously visiting s_i a moving segment m_i
        has been inserted into MS then
        link appropriately m and m_i
      endif
    endfor;
    if the uppermost endpoint of s has t = t_e
      for i = 1, 2
        remove s from the head of s_i's list of mates {so
          that the new head of the list now contains
          the new active mate of s_i}
      endfor
    endif
  endfor
  scan moving segments of MS to determine whether each
    cycle is a hole or an outer one {see subsection 6.1.2};
  for each hole cycle c
    intersect a halfline starting from a point on c's boundary
      and directed according to its normal with all moving
      segments of outer cycles and collect intersections;
    sort intersections and repeatedly delete adjacent
      intersections belonging to the same cycle: the only
      one remaining identifies the cycle containg c
  endfor
  report in output MS together with information about
    cycles and their nesting;
  append the group of segments starting at time t_e to A
endfor
```

Analysis. To analyze the time complexity of the algorithm we introduce new parameters:

- $\bar{k}$ is the number of intersecting pairs of p-faces (f, g) with $f \in \bar{a}$ and $g \in \bar{b}$;
- $\bar{R}$ is the total number of moving segments of result region units produced by the algorithm with input region units $\bar{a}$ and $\bar{b}$;
- $\bar{h}$ is the total number of hole cycles of result region units produced by the algorithm with input region units $\bar{a}$ and $\bar{b}$.

In the first step of the algorithm constant time computations are performed for each pair of p-faces, hence the step requires time $O(uv)$.

The analysis of the second step is more complex. Consider the case of a p-face f intersecting p-faces of the other input region unit. The computation of parts of f's left and right boundaries contributing to a result p-face requires to sort segments of L_f intersecting such boundaries (for details see subsection 6.1.2). The overall cost for all p-faces of this computation is therefore $O(\bar{k} \log \bar{k})$. Consider now the case of a p-face f not intersecting any p-face of the other input region unit. To determine whether f is inside the other region unit requires to scan through all other unit's moving segments (for details see subsection 6.1.2). Hence the overall cost for all p-faces of such a computation is $O(uv)$.

Successively, in any of the two cases, all segments lying within f are ordered along the t-axis, which costs, for all p-faces, $O(\bar{k} \log \bar{k})$.

Then, at each relevant time instant t_e, some segments are removed from the list D_f of active segments, while other segments belonging to the sublist New_{t_e} are inserted into D_f.

Since each segment is removed from D_f and inserted in New_{t_e} only once per p-face it lies within, and since each segment lies within only two p-faces, the overall cost of these operations, for all p-faces and for all relevant time instants, is $O(\bar{k})$. Moreover, at each relevant time instant t_e, linear scans of D_f are also performed to merge D_f and New_{t_e} and to mate segments. Since for any relevant time instant t_e and for any segment contained in D_f at time t_e a moving segment in a result region unit is produced in the third step of the algorithm, the overall cost of all scans, for all p-faces and for all relevant time instants, is $O(\bar{R})$.

In the third step the global list of segments S_T is constructed and sorted, which requires time $O(\bar{k} \log \bar{k})$.

During the plane sweep, for each relevant time instant, the list of active segments is traversed and

for each segment contained in it a moving segment is produced and inserted into a sorted list. The overall cost of these operations for all relevant time instants is therefore $O(\bar{R} \log \bar{R})$.

To determine whether cycles are hole or outer ones requires time $O(\bar{R})$ (for details see subsection 6.1.2), and for each hole cycle a sorting of outer cycles' moving segments is required to discover the outer cycle directly enclosing it, hence the overall cost of these operations for all cycles of all result units is $O((1 + \bar{h})\bar{R} \log \bar{R})$. Note that $\bar{h}$ may be 0.

Since $\bar{k} = O(\bar{R})$, the total cost of the algorithm is $O(uv + (1 + \bar{h})\bar{R} \log \bar{R})$. Summing up this cost for all pairs $(\bar{a}, \bar{b})$ of the refinement partition's region units, we have that the cost of this algorithm to perform a set operation on two input moving regions is $O(p \cdot u_{\max} v_{\max} + R \log \bar{R}_{\max} + p \cdot \bar{h}_{\max} \bar{R}_{\max} \log \bar{R}_{\max})$, where $\bar{h}_{\max}$ and $\bar{R}_{\max}$ are the respective maximal values of $\bar{h}$ and $\bar{R}$ over all elements of the refinement partition. We have written the bound in this way, since the last term is zero, if there are no hole cycles (and generally the number of hole cycles will be a small constant), and the second term only depends on the size of the output.

6.1.2. Detailed Description of Subalgorithms

In this subsection we give details of how to perform three tasks required by the algorithm for set operations: to determine parts of the boundaries of a p-face contributing to a result p-face (done during step 2), to check whether a p-face is inside a region unit (step 2), and to determine whether a cycle is a hole or an outer one (step 3).

To compute parts of the left boundary of a p-face f contributing to a result p-face, we proceed as follows. We compute all intersections of the left boundary with some intersection segment of f (intersection segments of f are stored in the list L_f during the first step of the algorithm).

If no intersection is found, we consider an intersection segment s' having an end point with smallest w coordinate, and we check whether the left boundary of f is part of the boundary of a result p-face using the indication on which side of s the result p-face lies (such indication is stored together with s in the first step of the algorithm).

Otherwise we sort intersections by t values and observe that they split the left boundary into subsegments. Then, for an intersection between a segment s and the left boundary, we use the indication on the side of s where the result p-face lies, to determine whether the corresponding subsegment of the left boundary is part of the boundary of a result p-face (special care must be taken to handle the case of multiple segments intersecting the left boundary in the same point). In more detail we proceed as follows:

for s in L_f
  if s intersects the left boundary **then** insert s in the list L_{left} **endif**
endfor
if no intersections are found **then**
  let s' be a segment of L_f having an end point with smallest w coordinate
  decide whether the left boundary contributes to a result p-face using the indication on which side of s the result p-face lies
else
  sort L_{left} by increasing values
  split the left boundary at all intersection points with a segment of L_{left} generating segments $l_0, \ldots, l_{|L_{\text{left}}|+1}$
  for each l_i
    let s_i be the segment of L_{left} whose intersection with the left boundary generated l_i
    decide whether l_i contributes to a result p-face using the indication on which side of s the result p-face lies
  endfor
endif

Since segments of L_{left} are a subset of the segments in L_f, the overall complexity, for all p-faces, is $O(\bar{k} \log \bar{k})$.

To check whether a p-face f of $\bar{a}$ is inside unit region $\bar{b}$, we consider f at any particular time, for example at time t', and check whether the resulting line segment $f(t')$ is inside $\bar{b}(t')$, the region at time t'. For this, the "plumbline algorithm" can be used: we count the number of line segments of $\bar{b}(t')$ intersecting a halfline extending from one end point of $f(t')$ in y-direction. This is implemented by a loop:

let p be the smaller end point of $f(t')$;
count := 0;
for each face g in $\bar{b}$ **do**
  if $g(t')$ is above p **then** count := count + 1 **endif**
endfor;
return count is odd

The complexity of the above loop is $O(v)$, hence the overall cost of this task, for all p-faces is $O(uv)$.

To determine whether a cycle is a hole or an outer one we consider one of its moving segments m. We evaluate m at a time instant t and using also the coordinates of the moving segment following m in the cycle, we determine on which side of m the interior of the region unit is. Then we consider a halfline extending from a point of $m(t)$ towards the interior of the region unit, and count the number of intersections with other moving segments of the cycle. If such a number is odd the cycle is an outer one, while in the other case it is a hole cycle. Since to count intersections we have to examine all moving segments of the cycle, the cost of this computation, for all cycles of all region units resulting from the application of the algorithm to a pair of unit regions, is $O(\bar{R})$.

6.1.3. Use of Projection Bounding Boxes

It is possible to improve the algorithm using projection bounding boxes (the *unit_pbb* fields) of $\bar{a}$ and $\bar{b}$. First, one can check in advance whether the projection

bounding boxes of $\bar{a}$ and $\bar{b}$ intersect. If they don't intersect, then union returns the set of p-faces $\bar{a} \cup \bar{b}$, intersection is empty, difference $\bar{a} \setminus \bar{b}$ is $\bar{a}$, and $\bar{b} \setminus \bar{a}$ is $\bar{b}$. In this case the running times are $O(u + v)$, $O(1)$, $O(u)$, and $O(v)$, respectively. Second, if the projection bounding boxes overlap, then let I be their intersection rectangle. Reduce $\bar{a}$ to $\bar{a}'$, the p-faces whose xy-projection intersects I, and similarly $\bar{b}$ to $\bar{b}'$. This step takes $O(u + v)$ time. The first step of the above algorithm, determining intersection segments is then run with the sets $\bar{a}'$ and $\bar{b}'$. The second step proceed as before except when one determines whether a p-face f of $\bar{a}$ not intersecting p-faces of $\bar{b}$ is inside or outside $\bar{b}$. In order not to lose the efficiency gain, one should find a technique to decide this, if possible, based on $\bar{b}'$ rather than $\bar{b}$.

Consider the relationship between I and B, the projection bounding box of $\bar{b}$. We can distinguish two cases (see Figure 11):

1. I has a common boundary with B.
2. I is completely inside B.

FIGURE 11. Relationship between I and B

For the plumbline algorithm, we have the freedom to select any of the directions above, below, left, or right from the given end point of a segment. Hence, in the first case, we can select a direction with a common boundary of B and I (left or below in Figure 11 (left)) and so reduce search to the segments in $\bar{b}'$. To support the second case, select one of the four rectangles adjacent to I and connecting to the boundary of B, for example, the one with minimal area (see Figure 11 (right)), and call it B'. Before running the second step of the algorithm, compute the set $\bar{b}''$ of faces of $\bar{b}$ that overlap B'. For any face f of $\bar{a}'$ for which the inside test is needed, checking can then be done just with the faces in $\bar{b}''$ as well as those in $\bar{b}'$.

Any faces of $\bar{a} \setminus \bar{a}'$ or $\bar{b} \setminus \bar{b}'$ can be reported directly, as described above for disjoint projection bounding boxes. Let u', v', and v'' be, respectively, the sizes of the sets $\bar{a}'$, $\bar{b}'$, and $\bar{b}''$. The running time for overlapping projection bounding boxes is $O(u+v+u'v'+u'v''+(1+\bar{h})\bar{R}\log\bar{R})$ for processing one pair of units.

6.1.4. Algorithm for the Intersects Predicate

The algorithm for the **intersects** predicate follows the same approach as the generalized algorithm for set operations described in subsection 6.1.1. However, since the aim is not to construct region units representing the result of an intersection operation, but only to find time intervals when intersections occur, it is not needed to mate intersection segments and to consider in any way an intersection segment with respect to both p-faces it lies within. Again, we describe the algorithm referring just to a pair of input region units $\bar{a}$ of a and $\bar{b}$ of b.

In a first step, for each p-face f of $\bar{a}$ we scan p-faces of $\bar{b}$ to find intersection segments. If f has no intersection with $\bar{b}$'s p-faces, we check whether f is completely inside $\bar{b}$. If that is the case then $\bar{a}$ and $\bar{b}$ intersect each other on the whole unit interval where both are defined. Hence the algorithm halts returning a single *ubool* unit with *true* value.

Otherwise, i.e. if none of the faces of $\bar{a}$ is completely enclosed, we store intersection segments in a global list S_T. We also compute parts of the left and right boundaries of f that belong to boundaries of p-faces of $\bar{a} \cap \bar{b}$ and insert them into S_T. When all p-faces of $\bar{a}$ have been processed, we sort S_T by increasing t values and then traverse it keeping a counter for the number of active segments.[16] The t values relevant for the result are those corresponding to transitions of the counter from 0 to 1 or from 1 to 0. The former type of transition indicates that the corresponding relevant t value is the end point of a result unit with *false* value and the start point of a subsequent result unit with *true* value, while the latter type of transition indicates the inverse situation.

If we denote by $\bar{k}$ the number of intersection segments, the algorithm requires time $O(uv + \bar{k}\log\bar{k})$ where the first term is due to the search of intersection segments and the second to the sorting operations performed on such segments.

Summing up this cost for all pairs $(\bar{a}, \bar{b})$ of refinement partition's region units, we have that the cost of this algorithm to perform a set operation on two input moving regions is $O(p \cdot u_{\max} v_{\max} + K \log \bar{k}_{\max})$, where $\bar{k}_{\max}$ is the maximum number of intersection segments generated by a pair of region units, while K is the total number of intersection segments for all pairs of input region units.

6.1.5. Algorithm for the Inside Predicate

In this subsection we describe an algorithm for the case *mregion* × *mregion* of the **inside** predicate. Also this algorithm is similar to the one for set operations described in subsection 6.1.1 and regards the arguments as sets of p-faces. Since the algorithm does not use the fact that a's p-faces form polyhedra in 3D space, it handles also the operation **inside**(*line* × *mregion*).

As usual the algorithm considers one after the other pairs of units $\bar{a}$ of a and $\bar{b}$ of b. In a first step, each p-face f of $\bar{a}$ is processed separately as follows. For each p-face g of $\bar{b}$ the intersection segment of f and g is computed. If f does not intersect any p-face of $\bar{b}$ then it is either

[16] Here the sweep works slightly differently than in Section 6.1.1: for each segment we produce one entry in S_T for the lower end point and one for the upper one to be able to decrement the counter on meeting the upper end.

completely inside or completely outside $\bar{b}$. Otherwise, intersection segments and parts of the boundary of f determine a set P of p-faces of the object $f \cap \bar{b}$ (see Figure 12). Then, at time $\hat{t}$, f is inside b if and only if

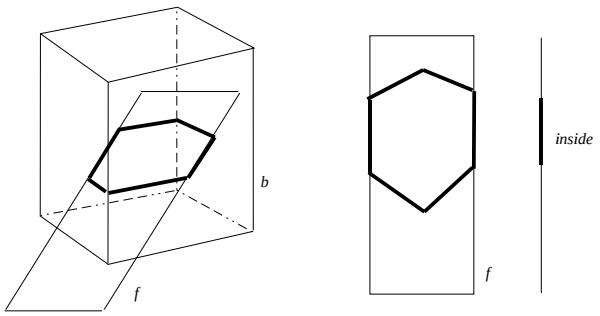

FIGURE 12. A face f intersecting $\bar{b}$ and a hexagonal p-face in $f \cap \bar{b}$ forming the set P

all the points of f having their t coordinate equal to $\hat{t}$ are contained in the set-union of p-faces in P. To verify such a condition, one performs a sweep of f's support plane maintaining a list of active segments which belong to the boundaries of p-faces in P: the condition is verified when there are only two active segments one of which is a subsegment of the left boundary of f and the other a subsegment of the right boundary of f. If it is discovered that f is completely outside $\bar{b}$ or it is never entirely inside $\bar{b}$, the algorithm halts reporting a single result unit with *false* value. Otherwise a set of disjoint time intervals where f is inside $\bar{b}$ is discovered.

In a second step, we insert such sets of intervals for all p-faces of $\bar{a}$ into a sorted list and then traverse the list maintaining, for each value of t, a counter c of how many intervals contain t. Since two intervals that contain the same t value come from two different p-faces, $\bar{a}$ is inside $\bar{b}$ when the value of the counter is equal to the number of p-faces of $\bar{a}$.

Let $\bar{k}$ be the number of intersecting pairs (f, g) of p-faces, with $f \in \bar{a}$ and $g \in \bar{b}$. The total number, for all $f \in \bar{a}$, of the intersection segments lying within f and the subsegments of f's boundaries which bound p-faces of $f \cap \bar{b}$ is $O(\bar{k})$. Hence, also the sum for all $f \in \bar{a}$ of the number of time intervals where f is inside $\bar{b}$, is $O(\bar{k})$. Then the cost of the algorithm is $O(uv + \bar{k} \log \bar{k})$. The overall cost, for all pairs $(\bar{a}, \bar{b})$ of the refinement partition's region units, is therefore $O(p \cdot u_{\max} v_{\max} + K \log \bar{k}_{\max})$, where $\bar{k}_{\max}$ is the maximum number of intersecting pairs of p-faces for a pair of region units, while K is the total number of intersecting pairs of p-faces for all pairs of input region units.

6.2. Computing the Distance between Two Moving Spatial Objects

In this section we deal with the computation of the distance between a moving point and a moving region (Section 6.2.1) and the distance between two moving regions (Section 6.2.2). The result is a moving real. One of the two operands of each operation may also be a spatial data type (i.e., *point* or *region*, respectively) which in this context can be interpreted as a "static" or "constant" spatio-temporal data type. Finally, in Section 6.2.3, we investigate the use of filtering techniques for accelerating computation.

6.2.1. Distance between a Moving Point and a Moving Region

Assuming that we are given a moving point *mp* and a moving region *mr*, in a first step we have to find out when *mp* was outside *mr*, because only then the distance function yields a value greater than 0. For that purpose, we employ the algorithm **inside** of Section 5.2 in [14] (already described with the **at** operation in Section 4), which for *mp* and *mr* returns a moving boolean *mb* representing when *mp* was inside *mr*. This takes $O(m+n+N)$ time (in practical cases) [14]. Hence, the first step of the algorithm can be formulated as follows:

$$mb := \textbf{not inside}(mp, mr)$$

where **not** is the negation operator on moving booleans (see Section 5.6). The negation of the moving boolean takes $O(b)$ time, if b is the number of boolean units. All boolean units with a *false* value indicate a distance equal to 0, whereas the other boolean units point to a distance greater than 0 and require further computation.

In a second step we compute the refinement partition between *mp* and *mr* and afterwards between the result and *mb*. This enables us later to identify all those point units, region units, and boolean units that have the same unit interval and the value *true* in the boolean unit. This constellation indicates that the point unit is located outside of the region unit in the same unit interval. This step takes $O(m + n + b)$ time.

Finally, the third step of the algorithm scans the refinement partition of *mp*,*mr*, and *mb* and performs the distance computation. For each refinement interval we check whether corresponding point, region, and boolean units exist. Only if this is the case, the distance function has to be computed. If the value of the boolean unit is *false*, the distance is 0, because the point unit is located inside the region unit. Otherwise, the distance between *mp* and *mr* has to be explicitly calculated by a function, say, *upoint_uregion_dist*. In the worst case, *mp* and *mr* are completely disjoint and have the same lifespan. Then the time complexity of the third step is $O(m + n + b)$ times the time complexity of the function *upoint_uregion_dist* described next.

The function *upoint_uregion_dist* takes a point unit *up* and a region unit *ur* as operands and returns a set *urls* of real units representing the distance of *up* and *ur* for each time of their common lifespan. It works as follows:

let $up = (i', mpo)$, $ur = (i'', F)$, $i := i' \cap i''$, $up' = (i, mpo)$, $ur' = (i, F)$;
$rlus := \varnothing$;
for each moving segment s of ur' do
  $units := upoint\_mseg\_dist(up', s)$;
  $rlus := concat(rlus, units)$;
endfor;
compute the refinement $rlus'$ of all real units in $rlus$ and attach the pertaining distance functions to each resulting time interval;
$urls := \varnothing$;
for each refinement interval i of $rlus'$ **do**
  $units := min\_func(rlus', i)$;
  $urls := concat(urls, units)$;
endfor

This algorithm consists of three parts. In the first part, the distance functions, which are represented as real units, are determined between the point unit and each moving segment of the region unit, i.e., between the 3D segment and each lateral face in 3D space. For a point unit and a single moving segment this is done by the operation *upoint_mseg_dist* (explained below). All resulting real units are collected in an unsorted list *rlus*. An example of such a collection of real units has been given on the left side of Figure 13. It has been computed with respect to a point unit and three moving segments, the latter forming a unit region. The distance computation yields the real units u_1 and u_2 for the first moving segment, the units u_3, u_4 and u_5 for the second moving segment, and the units u_6 and u_7 for the third moving segment.

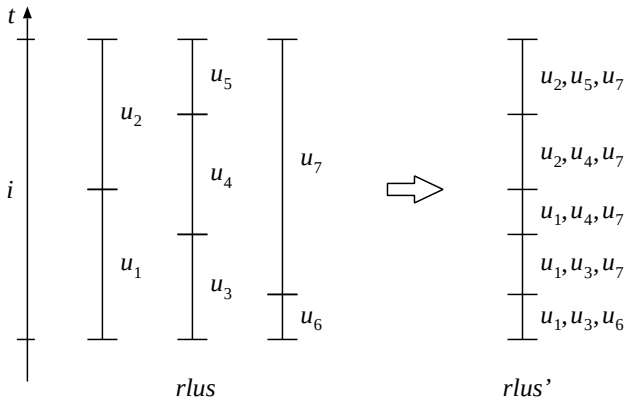

FIGURE 13. Distance functions, which are represented as real units, for a point unit and the moving segments of a region unit in time interval i (left side), and their refinement showing for each subinterval the attached active distance functions (right side).

The function *upoint_mseg_dist* calculates the distance between a point unit up and a moving segment s, the latter spanning a triangle or trapezium. Let P be the supporting plane uniquely determined by s. Two cases have to be distinguished: either a part of the projection of up into P lies inside s, or outside of s (Figure 14a). Because up is t-monotonic, at most one connected part of up can lie inside s, and at most two connected parts can lie outside of s. Both cases have to be distinguished, because the distance function is a linear polynomial in the first case (Figure 14b) and a quadratic polynomial in the second case, as we will see.

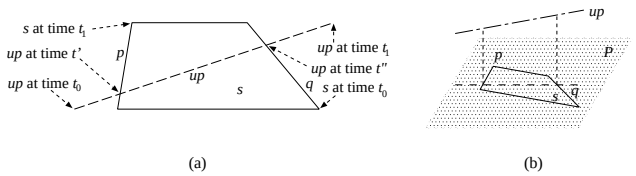

FIGURE 14. (a) Projection of point unit up into the supporting plane P determined by the moving segment $s = (p, q)$, (b) linear development of the distance between up and P and between up and the projection of up lying inside s.

An intersection of the projected point unit up and the moving segment s yields those segment parts lying inside or outside of s. All these segment parts are transformed back to the temporal domain, and we assume that up has the unit interval $[t_0, t_1]$, enters s at time t' and leaves s at time t'' (Figure 14a).

In the first case, the minimum distance at each time $t \in [t', t'']$ is given by up and a point inside the lateral face spanned by s. According to Figure 14b, in this time interval the distance evolves linearly. To compute the distance we consider the situation at some time t: the points $up(t)$, $p(t)$, and $q(t)$ form a triangle. The perpendicular through $up(t)$ on $s(t)$ is the minimum distance between $up(t)$ and $s(t)$, denoted by $dist((up, s), t)$. The distance between up and s at time t can then be calculated as

$$distance_1((up, s), t) = dist((up, s), t') + \frac{dist((up, s), t'') - dist((up, s), t')}{t'' - t'} \cdot (t - t')$$

which is a linear function, needs $O(1)$ time, and can be represented as a real unit.

In the second case, the minimum distance at each time $t \in [t_0, t']$ or $t \in [t'', t_1]$ is either given by the two point units up and p or by up and q. We have dealt with calculating the distance of two moving points as well as two point units in Section 5.5. The distance function is described by a quadratic polynomial. This also needs $O(1)$ time so that the first part of the algorithm *upoint_uregion_dist* needs $O(v)$ time. In Section 6.2.3 we will show how the number of moving segments to be considered in the further parts of this algorithm can be drastically reduced by using a filter technique.

In the second part of this algorithm, the task is to refine all real units of the list *rlus* into the list *rlus'* and to attach all *active* (i.e., valid) distance functions to each refinement interval of i. For this, each real unit $ur_j = (i_j, u_j)$ with $i_j = [t', t'']$ and $i_j \subseteq i$ is represented twice, namely by its *left* end point (t', l, u_j) and its *right*

end point (t'', r, u_j). This can be done by a scan of *rlus* in $O(v)$ time. Afterwards, all end points are sorted with respect to time and for equal times by the condition that $r < l$. This takes $O(v \log v)$ time. From this sorted list of end points we now derive the list *rlus'* of real units where each resulting subinterval is annotated with the current list of active distance functions. In each subinterval, $O(v)$ distance functions are active, each related to exactly one of the at most v moving segments (compare to the stripes of the left side of Figure 13). Moreover, for each right end point with $t' < t_1$, a new left end point exists at time t'.

Algorithmically, we maintain a list *adf* of active distance functions, which is empty at the beginning. For one or several consecutive left end points at the same time t', a new *active* refinement unit is created which is annotated with a *copy* of *adf* and the corresponding new distance functions. For one or several consecutive right end points at the same time t', the currently active refinement unit is closed, and all distance functions related to these right end points are removed from *adf*. At time t_1, list *adf* is empty again. The right side of Figure 13 shows the result for our example.

At most $O(v)$ real units (distance functions) and hence at most $O(v)$ refinement units can exist. Therefore, the time complexity of the second part of the algorithm is $O(v^2)$. For each interval of an active refinement unit, the list *adf* has then $O(v)$ entries which have to be attached to the active refinement unit.

In the third part of the algorithm *upoint_uregion_dist*, for each of the $O(v)$ refinement units of *rlus'*, the function *min_func* computes the minimum of the distance functions associated with each refinement interval i. This means to compute the lower contour of all function graphs (see Figure 15).

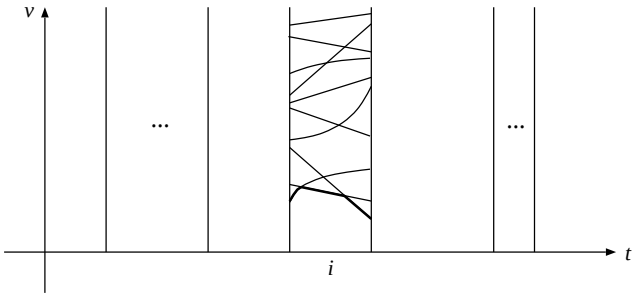

FIGURE 15. The lower contour of an arrangement of function graphs in time interval i.

Computational Geometry [24] gives us a solution to this problem. By using a combination of *divide and conquer* and *sweep* technique, we can compute the lower contour of k different t-monotonic function graphs, which are defined over the same time interval and where any two function graphs intersect each other in at most two points, in time $O(k \log k)$. In our case, k is bounded by v so that the time complexity of function *min_func* is $O(v \log v)$. Hence, the time complexity of the third part of the algorithm is $O(v^2 \log v)$.

In summary, the time complexity of the function *upoint_uregion_dist* is $O(v)$ for the first part, $O(v^2)$ for the second part, and $O(v^2 \log v)$ for the third part. Consequently, its overall time complexity is $O(v^2 \log v)$.

The overall time complexity of the whole algorithm is as follows. The first step needs $O(m + n + b + N)$ time, the second step $O(m + n + b)$ time, and the third step $O((m + n + b)\ v_{\max}^2 \log v_{\max})$ time. Since $N \leq v_{\max} n$, altogether, the algorithm requires $O((m + n + b)\ v_{\max}^2 \log v_{\max})$ time.

6.2.2. Distance between Two Moving Regions

The algorithmic scheme for computing the distance between two moving regions mr_1 and mr_2, respectively between a moving region and a region, is similar to the one in Section 6.2.1. For the first step of the algorithm, we use now the **intersects** operation on two moving regions (Section 6.1.4), since the distance is 0 whenever regions intersect. This needs $O((m + n)u_{\max}v_{\max} + K \log \bar{k}_{\max})$ time (with the notations of Section 6.1.4). As a result of this step we obtain the time intervals when mr_1 was intersecting or disjoint from mr_2 as a moving boolean mb. In the second step, the refinement partition is computed between mr_1, mr_2, and mb. This takes $O(m + n + b)$ time. Concerning the third step of the algorithm, we have to replace the operation *upoint_uregion_dist* by the operation *uregion_uregion_dist*. The second and third parts of these two operations are identical. But the first part has to be changed in a way that we use two nested loops, each traversing the moving segments of one of the two region units, to compute the distance functions for each pair of moving segments.

The distance computation for two moving segments is performed by the function *mseg_mseg_dist* (replacing *upoint_mseg_dist*). If their supporting planes are parallel, we can determine their distance in constant time. If they are not parallel, at least one of the four end point units of both moving segments must be involved in the distance calculations. Hence, we have to perform four distance computations, each between a point unit and a moving segment. This can be done by the function *upoint_mseg_dist* in constant time (see Section 6.2.1).

In total, the first part requires $O(uv)$ time and produces $O(uv)$ real units. This has a negative effect on the runtimes of the second and third part of the operation *uregion_uregion_dist*. The time complexity of the second parts develops to $O(u^2v^2)$ time, since $O(uv)$ refinement units exist and since for each interval of an active refinement unit, the list *adf* of active distance functions has $O(uv)$ entries, which have to be attached to the active refinement unit. Consequently, the third part requires $O(u^2v^2 \log uv)$ time, which is also the overall time complexity of *uregion_uregion_dist*.

The overall time complexity of the whole algorithm

is as follows. The first step needs $O((m+n)u_{\max}v_{\max} + K \log \bar{k}_{\max})$ time, the second step $O(m+n+b)$ time, and the third step $O((m+n+b)\ u_{\max}^2 v_{\max}^2 \log u_{\max} v_{\max})$ time. Thus, the whole algorithm requires $O((m+n+b)u_{\max}^2 v_{\max}^2 \log u_{\max} v_{\max} + K \log \bar{k}_{\max})$ time, where K is the total number of intersections between moving segments of the two arguments, and $\bar{k}_{\max}$ the maximal number of intersections between a pair of units.

This seems very expensive. Nevertheless, the filtering technique described next may help a lot.

6.2.3. Using a Filtering Technique

A problem of the two algorithms described in the two previous subsections is the large number of distance computations. The distance computation between a moving point and a moving region yields $O(v)$ distance functions for each refinement unit. For two moving regions these are even $O(uv)$ functions per unit. A filtering technique to reduce the number of distance functions (real units) would therefore be very attractive.

In case of a point unit and a region unit, we consider the projection of the point unit and all moving segments of the region unit into the xy-plane. This gives us a 2D-segment and a list of 2D-faces (triangles, trapezia). By a scan through all faces, we try to exclude as many faces as possible that cannot have a minimum distance to the segment. During the scan, for each face we compute its minimum and maximum distance to the segment; we thus obtain a distance interval. We also keep in mind the current minimum *minmax* of all right end points of distance intervals. It is initialized with the right end point of the distance interval of the first face in the scan. If the next distance interval in the scan is considered, and its left end point is greater than *minmax*, this interval can be ignored. Otherwise, it is inserted into a candidate list, which is initialized with the distance interval of the first face, and *minmax* is assigned the minimum of its current value and the right end point of the distance interval just inserted. The consequence is that during the scan, the value of *minmax* decreases. Because it can happen during the scan that *minmax* falls below a left end point of a distance interval inserted already earlier into the candidate list, a second scan of the candidate list is necessary which removes all those intervals with a left end point greater than *minmax*. Only for the moving segments remaining to the final candidate list, the exact distance computations have to be executed. This filtering step takes only $O(v)$ time.

In case of two region units we apply the same algorithm. Here, we perform a nested loop on two sets of 2D-faces and must be able to determine the minimum and maximum distance between two faces. The candidate list is maintained in the same way. This takes $O(uv)$ time.

In practice, this filtering technique lets us expect a drastic reduction of distance computations which can be bounded, say, by a very small constant $c << v$. Under this assumption, the first part of the function *upoint_uregion_dist* still needs $O(v)$ time, because we have to consider all 2D-faces of moving segments, and the second and third part require constant time per refinement unit. Hence, *upoint_uregion_dist* needs $O(v)$ time, and the runtime of the whole algorithm for a moving point and a moving region is $O((m+n+b)\ v_{\max})$. This assumption applied to the first part of the function *uregion_uregion_dist* still leads to $O(uv)$ for the first part, because we have to consider the product of the 2D-faces of both region units, and to a constant time per refinement unit for the second and third part. Hence, *uregion_uregion_dist* needs $O(uv)$ time, and the runtime of the whole algorithm for two moving regions is $O((m+n+b)\ u_{\max}v_{\max} + K \log \bar{k})$.

7. A PROTOTYPE IMPLEMENTATION

A prototypical implementation of the data structures and algorithms described in this paper is underway and has been partially completed. In this section we report on the details and current status of this implementation, as well as the problems faced during its development and how have they been addressed.

7.1. Implementation Environment

The prototype is being developed as an algebra module for the experimental extensible database system SECONDO [10] and as an Informix Datablade. The package is being built over a database interface layer that isolates the core of the package from database specific details, allowing us to use it in both database systems by just changing the interface layer.

SECONDO is an experimental extensible database system supporting the implementation of a wide range of data models and query languages. It is more flexible than common extensible and object-relational systems, offering the full extensibility of second-order signature (see [20]). Extensibility is provided by algebra modules defining and implementing new types (type constructors, in fact) and operators. The current SECONDO version provides two algebra modules: the standard algebra module, which provides basic alphanumeric data types *int*, *real*, *bool* and *string*, and the relational algebra module, which provides a relational algebra implementation. For more information about SECONDO see [10].

The spatiotemporal algebra has been built as two modules, one of them providing all the spatial data types and operations (the ROSE algebra module [23]) and the other providing the spatiotemporal support (the ST algebra module). Both modules make use of the standard and relational algebras of SECONDO and work together with them. The standard algebra is used to provide the alphanumeric types, whereas all data types provided by the spatiotemporal algebra can be used in SECONDO's relational algebra as new attribute

types. This is similar for the Illustra version with regard to its relational algebra and the alphanumeric types provided by it.

The ROSE algebra module has been designed to work independently of the ST module, allowing the user to install it alone if only spatial support is needed. For its development an existing ROSE algebra implementation has been taken as a basis, adapting it to be used within a database system as part of a spatiotemporal algebra. For that purpose, two main changes have been done:

1. It no longer uses the *realms* machinery [23] which basically provides a set of static geometries as a basis for defining values of spatial data types, since this strategy does not work any more in the spatiotemporal case with continuous change. Instead, the new implementation uses fixed size rational coordinates for the space, together with a preprocessing (*realmization*) step that makes explicit the intersection points of the arguments for each binary operation and a post-processing step that joins those segments of the result that can be represented as one (see [8]). These changes make it possible to use the original algorithms proposed in [22], although without ensuring consistency in the results of different operations (guaranteed before by the realm basis). However, if consistency is required, it can be easily achieved by using also a *dual grid* representation [8], where some additional restrictions on the precision of coordinates of spatial objects are made to ensure that the resulting spatial data types are closed under the ROSE algebra operations and therefore consistency is achieved. This solution is entirely satisfactory in the static case. We discuss in Section 7.4 possible options to achieve consistency in the dynamic case.
2. The arrays used in the original implementation are replaced by *database arrays*, a tool that automatically stores its data under the control of the database system, simplifying the development of new complex data types in algebra modules. Its implementation is described in Section 7.2.

The package is being implemented in C++, making an intensive use of templates. They are used, for example, for implementing the type constructors (*range* and *mapping*) and the database arrays.

The alphanumeric data types and operations are provided using the existing database support (existing algebra modules in SECONDO), whereas the spatial data types and operations are implemented in the ROSE algebra module. In its current version, the ST algebra module provides all the range and temporal data types and part of the spatiotemporal operations considered in this paper. Table 6 shows the status of their implementations. Some operations over range types (including *periods*) not considered in this paper are also provided, such as **duration**, **min**, **max** and **before**.

Completely (all signatures) implemented:		
deftime	**atinstant**	**speed**
rangevalues	**atperiods**	
trajectory	**initial**, **final**	$=, \neq$
traversed	**present**	**intersects**
inst,**val**	**atmin**, **atmax**	$<, \leq, \geq, >$
Partially (some signatures) implemented:		
at	**intersection**	**distance**
	union	
inside	**minus**	
Not yet implemented:		
locations	**isempty**	**direction**
passes		
derivative	**center**	**and**, **or**, **not**
derivable	**no_components**	
velocity	**perimeter**	
mdirection	**area**	

TABLE 6. Spatiotemporal operations currently implemented in the ST algebra module.

In the following subsections we address some specific implementation issues. First, we show how arrays are stored under the database control through the use of database arrays. Second, the *mapping* template, used for implementing the temporal data types, is described. Finally, some numerical robustness and consistency aspects are taken into consideration.

7.2. Database Arrays and Large Object Management

In Section 3, the data structures for complex data types were defined as using arrays for storing part of their data. For implementing them in an algebra module, each array is replaced by a database array (*dbarray*), an array implementation whose data are stored under the control of the database system. This is done by storing its content in a large object, which is a piece of storage space referenced by some identifier. Large object support is widely provided by current extensible database systems. In the SECONDO extensible DBMS, database arrays support is already provided and, thanks to its use of so-called *faked large objects* [9], they are automatically either represented *inline* in a tuple representation, or outside in a separate list of pages, depending on their size.

A rough description[17] of the data structure and methods of a *dbarray* can be seen in Table 7. A *dbarray* uses an *nlob* (a specialization of large objects implementing the *nestable* interface, whose peculiarities are described below) to store its content. The template argument type *T_Elem* can be any type, as far as it does not contain any pointer or other *dbarray*.

[17] Although using a C++ like syntax, some liberties have been taken for the sake of clarity: return values have been removed for methods that only return error codes, and integer values are always represented by *int*, regardless of which numerical precision is required.

template <class T_Elem> class dbarray		
private:	int numElem;	nlob data;
public:	create(int numElem);	setNumElem(int newSize);
	destroy();	T_Elem * get(int index, void * buffer);
	open(int mode);	T_Elem * getRange(int index, int numElem, void * buffer);
	close();	put(int index, T_Elem value);
	int getNumElem();	putRange(int index, int numElem, T_Elem * buffer);
	nest(nlob * extrn);	bool isNested();
	unNest();	restoreNestedRef(nlob * extrn);

TABLE 7. Definition of the *dbarray* template class.

Methods *create*(), *destroy*(), *open*() and *close*() are self-explanatory. Method *getNumElem*() returns the size (number of entries) of the array, whereas *setNumElem*() sets it. Methods *get*() and *put*() allow to retrieve (respectively set) the content of the element *index* in the array. With *getRange*() and *putRange*() a range of elements can be retrieved/set at a time.

Both *nlob* and *dbarray* classes implement the *nestable* interface, which allows them to either store their content in their own large object or as part of another external *nlob*, as shown in Figure 16. In both cases the nested state of the object will be transparent for the user, except that when nested the object will be read-only.

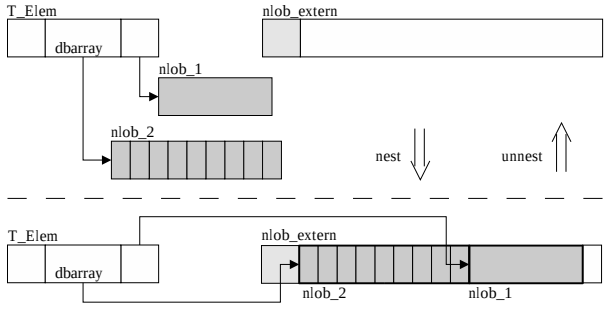

FIGURE 16. Behaviour of the *nest*() and *unnest*() methods.

The *nestable* interface provides the methods shown in Table 8. The purpose of these methods is the following:

- *nest*(): this method gets as argument a pointer *extrn* to an *nlob*. If the object whose *nest*() method is called is an *nlob*, it appends its data to *extrn*, keeping an internal reference to it as well as the required indexes to be able to find on it its data. If the object is not an *nlob*, then it calls the *nest*() method (with the same argument) of all the nestable objects it uses.
- *unNest*(): the object for which it is called must have been previously nested. If it is an *nlob*, it reads its data from the external *nlob* it uses and stores them in its own large object. If it is not an *nlob*, then it calls the *unnest*() method of all the nestable objects it uses.
- *isNested*(): this method returns *true* if the object is nested, *false* otherwise.
- *restoreNestedRef*(): the object for which it is called must have been previously nested. It restores all internal references to the *nlob* on which the object has been nested, making them point to the *nlob* passed as argument to the function (*extrn*). It is used to correct these references when the external *nlob* object has been reallocated. When called for an *nlob* it just replaces its internal pointer by the new one. For any other object, it calls the *restoreNestedRef*() method (with the same argument) of all the nestable objects used by it.

The *dbarray* class is used for the implementation of most of the non-alphanumeric types, whereas its nesting capabilities are designed to support the implementation of temporal data types, as shown in next section.

7.3. Representation of Temporal Data Types

In the data model proposed in this paper, temporal data types (e.g., *mint*, *mreal*, *mregion*, etc.) are represented following a common schema, by decomposing them into units, each of them containing a time interval for which it is defined and the function that represents its behavior during that time interval. Therefore, it seems advisable to represent all of them using a common data structure, providing the same interface and simplifying the implementation of algorithms that work on the temporal types. With this purpose, we introduce the *mapping* data structure. It consists of an array (*dbarray*) of units, a *deftime* field (of type *periods*) and an additional *nlob* object *extrn*.

The methods provided by this data structure are shown in Table 9. The template type *T_Unit* specifies the type of unit to use, whereas *T_Value* is the non-temporal type associated to it (e.g., *point* for a *upoint*).

Methods *create*(), *destroy*(), *open*() and *close*() behave similarly to the ones of the *dbarray* class. Method *unitIndex*() is used for getting the index of the unit that defines the object's value for a given time instant t, whereas *getUnit*() allows to retrieve a unit by its index. Method *setUnit*() allows to store one unit in a given position in the array, whereas *appendUnit*() appends it to the end of the array (increasing its size by 1). Method *deftime*() returns the periods of time for which the mapping contains units and

Methods of the nestable interface (for a class T_Elem)	
T_Elem::nest(nlob * extrn);	bool T_Elem::isNested();
T_Elem::unNest();	T_Elem::restoreNestedRef(nlob * extrn);

TABLE 8. Definition of the *nestable* interface.

template <class T_Unit, class T_Value> class mapping		
private:	dbarray<T_Unit> intervals;	int num_of_units;
	periods deftime;	nlob extrn;
public:	create(int num_of_units);	appendUnit(T_Unit u);
	destroy();	setUnit (int index, T_Unit u);
	open (int mode);	Periods * deftime ();
	close ();	mapping * atperiods (Periods * p);
	int numUnits ();	mapping * at (T_Value * v);
	int unitIndex(Instant t);	bool present (Instant * t);
	T_Unit * getUnit(int index);	bool present (Periods * p);

TABLE 9. Definition of the *mapping* template class.

methods *atperiods*() and *at*() return a copy of the mapping data structure restricted to the *periods* passed as argument, or to the instants at which it takes the non-temporal value passed as argument, respectively. Method *present*() allows to check whether the object's value is defined at a given time instant or is ever defined during a given set of time intervals.

Any *T_Unit* type designed to be stored in the *mapping* data structure must implement the *nesting* interface. Whenever a unit is stored (methods *setUnit*() and *appendUnit*()) it is nested over *extrn* and stored in the *dbarray*. When a unit is retrieved (method *getUnit*()), it is read from the *dbarray* and its *restoreNestedRef*() method is called (with *extrn* as argument) to ensure it points to the right place. The object is returned nested, so if this is relevant the user code should call its *unnest*() method. In general that will not be needed, because in most cases the unit will only be read.

With the implementation described above, a *mapping* data structure can contain units with *dbarray* components, allowing the implementation of a *mregion* as described in section 3, with a root record containing a *mapping* data structure for *uregions*, instead of the array and the *deftime* field, and the *uregions* using a *dbarray* instead of a conventional one.

7.4. Numbers and Robustness

As already mentioned when describing the ROSE algebra module, the current implementation uses fixed size rationals for space coordinates, and the same holds for time values.

Although the use of such a representation for the space, combined with some restriction approach (e.g., *dual grid*), is enough to ensure consistency among the spatial operations to be implemented in this package, it is not enough to ensure it for the whole spatiotemporal algebra, where continuously changing values are represented. Moreover, not even consistency between spatial operations can be ensured if spatiotemporal operations returning spatial values are used, because even if the user applications use *dual grid* restrictions for the objects they store in the system, the results of those spatiotemporal operations would not.

One solution would be to provide an extra operation which would get a spatial value and return an approximation conforming *dual grid* restrictions. That way, for applications that need spatial consistency the spatial result of any spatiotemporal operations can be approximated to a *dual grid* conforming value, allowing to use the spatiotemporal capabilities of the algebra module and at the same time keep the consistency among spatial operations (assuming of course that the spatial values stored by the user applications also conform to the *dual grid* restrictions). For applications where spatial consistency is not relevant, this extra operation can be simply ignored.

A second solution would be to replace the fixed size rationals by varying length rationals, what would ensure consistency among operations for the whole spatiotemporal algebra. Although this option is not being provided in the current implementation, we plan to develop in the future a version with support for varying length rationals and explore the costs that this could have in its performance. The approach used in the current version for implementing the *mapping* data structure (where units using *dbarrays* are *nested*) could be also used to deal with a varying size representation for coordinates and time.

8. CONCLUSIONS

This paper provides the third major step of a development started in the papers [11, 21, 14]. The three steps are:

1. Design a comprehensive system of data types and operations for moving objects (or, more generally, time-dependent geometries), first at the level of an *abstract model* [21].

2. Define finite representations for all the types of the abstract model that can be mapped into efficient data structures [14]. This is the data type part of a *discrete model* for [21].
3. Design efficient algorithms for the operations of the abstract model, which is the operational part of the *discrete model* for [21].

When we started the work on this third step, it appeared rather overwhelming, due to the fact that the generic design of operations in [21] literally produces hundreds of signatures for which efficient algorithms should be found, and it was not clear initially how many of them can be treated in a generic way by the same algorithm, or need specialized techniques. And indeed, this paper is quite long. Nevertheless, after reducing the scope slightly, we succeeded in mapping this unknown terrain, and we feel that the paper strikes a reasonable balance between treating briefly large sets of relatively straightforward algorithms and going into detail on the more involved ones.

As a result, this paper offers a very good basis for the implementation of a quite powerful DBMS extension package for moving objects. Note that the model implemented here includes the case of geometries changing in discrete steps; so this can be used as a quite general package supporting spatio-temporal database management.

So far we have assumed that the operations of our moving objects algebra may be used in a query language and are mapped directly to algorithms of the physical algebra for query execution. A very interesting issue for further research is the design of new auxiliary operations in the physical algebra to support compound operations in the logical algebra, so that query optimization can map the latter to more efficient algorithms. Two examples:

The compound operation **min**(**deftime**(_)): *mpoint* → *instant* can be implemented in $O(1)$ time by looking up the first instant of the *deftime* index of the argument, instead of first producing a copy of this index and then looking up the instant. Hence, introducing such an operation, called e.g. **mindeftime** would be beneficial.

The compound operation

atperiods(_ ,**deftime**(**at**(_ ,*true*))):
mregion × *mbool* → *mregion*

restricts the moving region argument to the times when its second argument is true. This can be implemented by a parallel scan of the two argument mappings in $O(m + n + R)$ time. This compound operation is particularly interesting as it implements the **when** operator of the abstract model (which was always supposed to be implemented by rewriting, see [21]). Again, one might introduce a corresponding **when** operator at the physical level.

Other future work is, of course, the implementation of a DBMS extension package as described in this paper (for us: completion of the prototype described in Section 7), and the experimental evaluation of such a package.

ACKNOWLEDGMENTS

We thank the anonymous referees for their careful reviews and numerous suggestions that have helped us to improve the presentation considerably. Thanks also to Nicole Richter for her implementation of parts of the spatio-temporal algebra.

REFERENCES

[1] T. Abraham and J.F. Roddick. Survey of Spatio-Temporal Databases. *GeoInformatica*, 3:61–99, 1999.

[2] P.K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. 19th Symp. on Principles of Database Systems*, pages 175–186, Dallas, Texas, 2000.

[3] J.L. Bentley and T. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers*, C-28:643–647, 1979.

[4] M. Cai, D. Keshwani, and P. Revesz. Parametric Rectangles: A Model for Querying and Animation of Spatiotemporal Databases. In *Proc. 7th. Int. Conf. on Extending Database Technology*, pages 430–444, Konstanz, Germany, 2000.

[5] J. Chomicki and P. Revesz. Constraint-Based Interoperability of Spatio-Temporal Databases. In *Proc. 5th Int. Symp. on Large Spatial Databases (SSD)*, pages 142–161, Berlin, Germany, 1997.

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

[7] José A. Cotelo Lema, L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. Algorithms for Moving Objects Databases. Technical Report Informatik 289, FernUniversität Hagen, October 2001.

[8] José A. Cotelo Lema and R.H. Güting. Dual Grid: A New Approach for Robust Spatial Algebra Implementation. *GeoInformatica*, 6(1):57–76, 2002.

[9] S. Dieker and R.H. Güting. Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering*, 32(3):247–269, 2000.

[10] S. Dieker and R.H. Güting. Plug and Play with Query Algebras: Secondo. A Generic DBMS Development Environment. In *Proc. of the IntDatabase Engineering and Applications Symposium (IDEAS)*, pages 380–390, September 2000.

[11] 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.

[12] M. Erwig and M. Schneider. Developments in Spatio-Temporal Query Languages. In *IEEE Int. Workshop on Spatio-Temporal Data Models and Languages (STDML)*, pages 441–449, Florence, Italy, 1999.

[13] M. Erwig and M. Schneider. Spatio-Temporal Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):881–901, 2002.

[14] L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–330, 2000.

[15] M. Garey, D. S. Johnson, F. P. Preparata, and R. J. Tarjan. Triangulating a Simple Polygon. *Information Processing Letters*, 7(4):175–180, 1978.

[16] S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin. DEDALE, A Spatial Constraint Database. In *Proc. Int. Workshop on Database Programming Languages*, pages 38–59, 1997.

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

[18] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating Interpolated Data is Easier than You Thought. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 156–165, Cairo, Egypt, 2000.

[19] S. Grumbach, P. Rigaux, and L. Segoufin. Spatio-Temporal Data Handling with Constraints. *GeoInformatica*, 5:95–115, 2001.

[20] R.H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 277–286, Washington, May 1993.

[21] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M.Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

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

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

[24] R. Klein. *Algorithmische Geometrie*. Addison-Wesley, 1997.

[25] G. Kollios, D. Gunopulos, and V.J. Tsotras. On Indexing Mobile Objects. In *Proc. 18th Symp. on Principles of Database Systems*, pages 261–272, Philadelphia, Pennsylvania, 1999.

[26] D. Peuquet. Making Space for Time: Issues in Space-Time Data Representation. *GeoInformatica*, 5:11–32, 2001.

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

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

[29] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 395–406, Cairo, Egypt, 2000.

[30] J.-M. Saglio and J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *GeoInformatica*, pages 71–93, 2001.

[31] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 331–342, Dallas, Texas, 2000.

[32] A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. Int. Conf. on Data Engineering*, pages 422–432, Birmingham, U.K. 1997.

[33] J. Su, H. Xu, and O. Ibarra. Moving Objects: Logical Relationships and Queries. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases (SSTD)*, pages 3–19, Redondo Beach, California, 2001.

[34] Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. 6th Int. Symp. on Spatial Databases*, pages 147–164, Hong Kong, China, 1999.

[35] E. Tossebro and R.H. Güting. Creating Representations for Continuously Moving Regions from Observations. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases (SSTD)*, pages 321–344, Redondo Beach, California, 2001.

[36] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. On the Generation of Time-Evolving Regional Data. *GeoInformatica*, 6(3):207–231, 2002.

[37] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases (SSTD)*, pages 20–35, Redondo Beach, California, 2001.

[38] 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 Int. Conference on Data Engineering*, pages 588–596, Orlando, Florida, 1998.

[39] O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7:257–387, 1999.

[40] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. of the 10th Int. Conference on Scientific and Statistical Database Management*, pages 111–122, Capri, Italy, 1998.

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