MOVING OBJECT LANGUAGES

Ralf Hartmut Güting
Fakultät für Mathematik und Informatik
Fernuniversität Hagen, Germany

SYNONYMS

Moving Object Languages; Query Languages for Moving Objects; Spatio-Temporal Query Languages

DEFINITION

The term refers to query languages for *moving objects databases*. Corresponding database systems provide concepts in their data model and data structures in the implementation to represent moving objects, i.e., continuously changing geometries. Two important abstractions are *moving point*, representing an entity for which only the time dependent position is of interest, and *moving region*, representing an entity for which also the time dependent shape and extent is relevant. Examples of moving points are cars, trucks, air planes, ships, mobile phone users, RFID equipped goods, or polar bears; examples of moving regions are forest fires, deforestation of the Amazone rain forest, oil spills in the sea, armies, epidemic diseases, hurricanes, and so forth.

There are two flavors of such databases. The first, which we call the *location management* perspective, represents information about a set of currently moving objects. Basically one is interested in efficiently maintaining their locations and asking queries about the current and expected near future positions and relationships between objects. In this case, no information about histories of movement is kept. The second we call the *spatio-temporal data* perspective; here the complete histories of movements are represented. The goal in the design of query languages for moving objects is to be able to ask any kind of questions about such movements, perform analyses, derive information, in a way as simple and elegant as possible. Yet such queries must be executed efficiently.

HISTORICAL BACKGROUND

The field of moving objects databases with the related query languages came into being in the late nineties of the last century mainly by two parallel developments. First, the group around Wolfson in a series of papers [13, 15, 16, 17] developed a model that allows one to keep track in a database of a set of time dependent locations, e.g., to represent vehicles. They observed that one should store in a database not the locations directly, which would require high update rates, but rather a motion vector, representing an object's expected position over time. An update to the database is needed only when the deviation between the expected position and the real position exceeds some threshold. At the same time this concept introduces an inherent, but bounded uncertainty about an object's real location. The group formalized this model introducing the concept of a *dynamic attribute*. This is an attribute of a normal data type which changes implicitly over time. This implies that results of queries over such attributes also change implicitly over time. They introduced a related query language FTL (future temporal logic) that allows one to specify time dependent relationships between expected positions of moving objects. Hence this group established the location management perspective.

Second, the European project CHOROCHRONOS set out to integrate concepts from spatial and temporal databases and explored the *spatio-temporal data* perspective. This means, one represents in a database time-dependent geometries of various kinds such as points, lines, or regions. Earlier work on spatio-temporal databases had generally admitted only discrete changes. This restriction was dropped and continuously changing geometries were considered. Güting and colleagues developed a model based on the idea of *spatio-temporal data types* to represent histories of continuously changing geometries [5, 9, 6, 2]. The model offers data types such as *moving point* or *moving region* together with a comprehensive set of operations. For example, there are operations to compute the projection of a moving point into the plane, yielding a *line* value, or to compute the distance between a moving point and a moving region, returning a time dependent real number, or *moving real*, for short. Such data types can be embedded into a DBMS data model as attribute types and can be implemented as an extension package.

A second approach to data modeling was pursued in CHOROCHRONOS by Grumbach and colleagues who applied the constraint model to the representation of moving objects [7, 11] and implemented a prototype called Dedale. Constraint databases can represent geometries in *n*-dimensional spaces; since moving objects exist in 3D (2D + time) or 4D (3D + time) spaces, they can be handled by this approach. Several researchers outside CHOROCHRONOS also contributed to the development of constraint-based models for moving objects.

SCIENTIFIC FUNDAMENTALS

In the following two subsections we describe two major representatives for the location management and the spatio-temporal data flavor of moving objects databases in some detail, namely the MOST model and FTL language, and the approach of spatio-temporal data types. In a short closing subsection some further work related to languages for moving objects is mentioned.

Modeling and Querying Current Movement - the MOST Model and FTL Language

In this section, moving objects databases based on the location management perspective and a related query language are discussed. That is, the database keeps track of a collection of objects moving around currently and one wishes to be able to answer queries about the current and expected near future positions. Such sets of entities might be taxi-cabs in a city, trucks of a logistics company, or military vehicles in a military application. Possible queries might be:

- Retrieve the three free cabs closest to Cottle Road 52 (a passenger request position).
- Which trucks are within 10 kms of truck T70 (which needs assistance)?
- Retrieve the friendly helicopters that will arrive in the valley within the next 15 minutes and then stay in the valley for at least 10 minutes.

Statically, the positions of a fleet of taxi-cabs, for example, could be easily represented in a relation

```
    taxi-cabs(id: int, pos: point)
```

Unfortunately this representation needs frequent updates to keep the deviation between real position and position in the database small. This is not feasible for large sets of moving objects.

The MOST (moving objects spatio-temporal) data model [13, 16], discussed in this section, stores instead of absolute positions a motion vector which represents a position as a linear function of time. This defines an expected position for a moving object. The distance between the expected position and the real position is called the *deviation*. Furthermore, a *distance threshold* is introduced and a kind of contract between a moving object and the database server managing its position is assumed. The contract requires that the moving object observes the deviation and sends an update to the server when it exceeds the threshold. Hence the threshold establishes a bound on the *uncertainty* about an object's real position.

The MOST model relies on a few basic assumptions: A database is a set of object classes. Each object class is given by its set of attributes. Some spatial data types like *point*, *line*, or *polygon* with suitable operations are available. Object classes may be designated as spatial which means they have a single spatial attribute. Spatial operations can then be directly applied to objects, e.g. *distance*(o_1, o_2) for two objects o_1 and o_2. Besides object classes, the database contains an object called *Time* which yields the current time at every instant. Time is assumed to be discrete and can be represented by *integer* values. The value of the *Time* object increases by one at each clock tick (e.g. every second).

Dynamic Attributes

A fundamental new concept in the MOST model is that of a *dynamic attribute*. Each attribute of an object class is classified to be either static or dynamic. A dynamic attribute is of a standard data type (e.g. *int*, *real*) within the DBMS conceptual model, but changes its value automatically over time. This means that queries involving such attributes also have time dependent results, even if time is not mentioned in the query and no updates to the database occur.

For a data type to be eligible for use in a dynamic attribute, it is necessary that the type has a value 0 and an addition operation. This holds for numeric types but can be extended to types like *point*. A dynamic attribute *A* of type *T* is then represented by three subattributes *A.value*, *A.updatetime*, and *A.function*, where *A.value* is of type *T*, *A.updatetime* is a time value, and *A.function* is a function f: $int \rightarrow T$ such that at time $t = 0, f(t) = 0$. The semantics of this representation is called the value of A at time t and defined as

$$value(A, t) = A.value + A.function(t - A.updatetime) \qquad \text{for } t \geq A.updatetime$$

When attribute A is mentioned in a query, its dynamic value $value(A, t)$ is meant.

Representing Object Positions

A simple way of modeling objects moving freely in the *xy*-plane would be to introduce an attribute *pos* with two dynamic subattributes *pos.x* and *pos.y*. For example, one might define an object class for cars:

```
cars(license_plate: string, pos: (x: dynamic real, y: dynamic real))
```

For vehicles, a more realistic assumption is that they move along road networks. A more sophisticated modeling of time dependent positions in MOST uses a *loc* attribute with six subattributes *loc.route*, *loc.startlocation*, *loc.starttime*, *loc.direction*, *loc.velocity*, and *loc.uncertainty*. Here *loc.route* is a (pointer to) a *line* value (a polyline) describing the geometry of the road on which the vehicle is moving. Say that on *loc.route*, one chooses a point as origin and a particular direction as positive direction. Then, the initial location *loc.startlocation* is given by its distance from the origin; this distance is positive if the direction from the origin to the initial location is in the positive direction, otherwise it is negative. The velocity also is negative or positive accordingly. *Startlocation*, *starttime*, and *velocity* correspond to the components of a dynamic attribute explained above. The value of *loc* at time t, $value(loc, t)$ is now a position on the *route* polyline defined in the obvious way. Query evaluation may take the *uncertainty* into account.

Semantics of Queries, Query Types

In traditional databases, the semantics of a query are defined with respect to the current state of a database. This is not sufficient for the MOST model, as queries may refer to future states of a database. A *database state* is a mapping that associates each object class in the database with a set of objects of appropriate types, and the *Time* object with a time value. We denote by $o.A$ and $o.A.B$ attribute A of object o and subattribute B of attribute A of object o, respectively. In database state s, the value of $o.A$ is denoted $s(o.A)$ and the value of the *Time* object as $s(Time)$. For each dynamic attribute A, its value in state s is $value(A, s(Time))$.

Semantics of queries are now defined relative to a database history. A *database history* is an infinite sequence of database states, one for each clock tick, beginning at some time u, hence is $s_u, s_{u+1}, s_{u+2}, \ldots$ An update at some time $t > u$ will affect all database states from t on. Hence with each clock tick we get a new database state, and with each update a new database history. We denote by $Q(H, t)$ a query Q evaluated on database history H assuming a current time t.

There are now two types of queries, namely *instantaneous* and *continuous* query.[1] An instantaneous query issued at time t is evaluated once on the history starting at time t, hence as

$Q(H_t, t)$ (*instantaneous query*)

In contrast, a continuous query is (conceptually) reevaluated once for each clock tick, hence as a sequence of instantaneous queries

$Q(H_t, t), Q(H_{t+1}, t+1), Q(H_{t+2}, t+2), ...$ (*continuous query*)

The result of a continuous query changes over time; at time u the result of $Q(H_u, u)$ is valid. Of course, reevaluating the query on each clock tick is not feasible, instead, the evaluation algorithm for such queries is executed only once and produces a time dependent result, in the form of a set of tuples with associated time stamps. Reevaluation is necessary only for explicit updates.

The Language FTL (Future Temporal Logic)

The query language associated with the MOST model is called FTL (future temporal logic). We first show a few example queries formulated in FTL.

1. Which trucks are within 10 kms of truck T70?

   ```
   RETRIEVE t
   FROM trucks t, trucks s
   WHERE s.id = 'T70' ∧ dist(s, t) <= 10
   ```

   Here nothing special happens, yet, the result is time dependent.

2. Retrieve the helicopters that will arrive in the valley within the next 15 minutes and then stay in the valley for at least 10 minutes.

   ```
   RETRIEVE h
   FROM helicopters h
   WHERE eventually_within_15 (inside(h, Valley) ∧
     always_for_10 (inside(h, Valley))
   ```

   Here *Valley* is a polygon object.

The general form of a query in FTL is[2]

```
RETRIEVE <target-list> FROM <object classes> WHERE <FTL-formula>
```

The interesting part is the FTL formula. FTL formulas are similar to first-order logic, hence are built from constants, function symbols, predicate symbols, variables and so forth. Some special constructs in the definition of formulas are the following:

- If f and g are formulas, then f **until** g and **nexttime** f are formulas

The semantics of a formula are defined with respect to

1. There exists a further query type, persistent query, which is omitted here.
2. In the original literature about FTL, a single class of moving objects is assumed and the FROM clause omitted.

- a variable assignment μ which associates with each variable in the formula a corresponding database object (e.g. for s, t in the first example query $\mu = [(s, T_{10}), (t, T_{20})]$ where T_i are truck objects in the database)
- a database state s on history h

We then define what it means for a formula to be satisfied at state s on history H with respect to variable assignment μ (satisfied at (s, μ) for short). The semantics of the special constructs are defined as follows:

- f **until** g is satisfied at (s, μ) :⇔ either g is satisfied at (s, μ), or there exists a future state s' on history H such that (g is satisfied at (s', μ) ∧ for all states s_i on history H before state s', f is satisfied at (s_i, μ)).
- **nexttime** f is satisfied at (s, μ) :⇔ f is satisfied at (s', μ) where s' is the state immediately following s in history H.

Based on these temporal operators with well-defined semantics, some derived notations can be defined:

- **eventually** g ≡ *true* **until** g
- **always** g ≡ (¬ **eventually** (¬ g))

In addition, it is useful to have bounded temporal operators:

- f **until_within_c** g asserts that there exists a future time within c units of time from now such that g holds and until that time f will hold continuously
- f **until_after_c** g asserts that there exists a future time after at least c units of time from now such that g holds and until that time f will hold continuously.

Based on these, one can again define further bounded temporal operators:

- **eventually_within_c** g ≡ *true* **until_within_c** g
- **eventually_after_c** g ≡ *true* **until_after_c** g
- **always_for_c** g ≡ g **until_after_c** *true*

So now we have explained the semantics of the constructs used in example query 2 above.

Evaluation

The algorithm for evaluating FTL queries can here only be briefly sketched. The basic idea is to compute a relation for every subformula of the given FTL formula bottom up, starting from atomic formulas like `dist(s, t) <= 10`. The relation R_f for subformula f has an attribute for each free variable occuring in f, and two attributes for time stamps t_{start}, t_{end}. Hence the relation for formula `dist(s, t) <= 10` has schema (s, t, t_{start}, t_{end}). It has a tuple (o, o', T, T') for every pair of objects O, O' and for every maximal time interval [T, T'] such that objects O, O' are within distance 10 throughout the interval [T, T']. For operators combining two subformulas such as $f \wedge g$ or f **until** g it is then possible to compute their relations essentially by joins over common object identifier attributes (equal variables in both subformulas), manipulating the time stamps in an appropriate way.

The result relation for the complete formula is the result for a continuous query. As time progresses, tuples of this relation can be added to or removed from the current result. It can also be used to answer an instantaneous query selecting just the tuples valid at the time of issuing this query.

Modeling and Querying History of Movement - Spatio-Temporal Data Types

In this section we discuss the spatio-temporal data perspective for moving objects databases. That is, we consider the geometries stored in spatial databases and allow them to change continuously over time. Some of the most important abstractions used in spatial databases are

- point - an object for which only the position in space is relevant
- line - a curve often representing connections such as roads, rivers
- region - representing objects for which the extent is relevant
- partition - subdivisions of the plane, e.g. of a country into states
- network - graph or network structures over roads, rivers, power lines, etc.

Such abstractions are usually captured in *spatial data types*, consisting of the type together with operations.

Spatio-Temporal Data Types

The idea of the approach [5] presented in the following is to introduce spatio-temporal data types that encapsulate time dependent geometries with suitable operations. For moving objects, point and region appear to be most relevant, leading to data types *moving point* and *moving region*, respectively. The *moving point* type can represent entities such as vehicles, people, or animals moving around whereas the *moving region* type can represent hurricanes, forest fires, armies, or flocks of animals, for example. Geometrically, values of spatio-temporal data types are embedded into a 3D space (2D + time) if objects move in the 2D plane, or in a 4D space if movement in the 3D space is modeled. Hence, a moving point and a moving region can be visualized as shown in Figure 1.

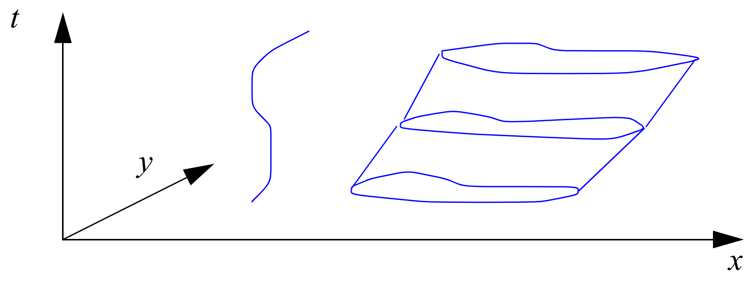

Figure 1: A moving point and a moving region

In the following, we first motivate the approach by introducing an example database with spatio-temporal data types, providing a number of operations on these data types, and formulating queries using these operations. We then discuss the underlying design principles for types and operations, the distinction between abstract model and discrete model in defining the semantics of types, and the structure of type system and operations in more detail. Finally, the implementation strategy is briefly outlined.

Example Operations and Queries

As a simple example, suppose we have two relations representing cars and weather conditions, whose movements have been recorded.

```
cars (license_plate: string, trip: moving(point))
weather (id: string, area: moving(region))
```

Assume further, some available operations on these types, used in queries below, are the following:

Signature		*Operation*
moving(*point*)	→ *line*	**trajectory**
moving(*region*)	→ *region*	**traversed**
moving(α)	→ *periods*	**deftime**
moving(*point*) × *moving*(*region*)	→ *moving*(*point*)	**intersection**
moving(α) × *instant*	→ *intime*(α)	**atinstant**
intime(α)	→ *instant*	**inst**
intime(α)	→ α	**val**
periods	→ *int*	**duration**
int × *int* × *int* × *int*	→ *instant*	**theinstant**

In these signatures, *moving* is viewed as a type constructor that transforms a type α into a time dependent version of that type, *moving*(α). These operations use further data types:

- *instant*, representing an instant of time
- *periods*, representing a set of disjoint time intervals
- *intime*(α), where *intime* is a type constructor building pairs of an *instant* and a value of another type α.

The operations have the following meaning: **Trajectory** and **traversed** compute the projection of a moving point or moving region into the 2D plane; **deftime** computes the projection on the time axis. The **intersection** of a moving point and a moving region is a moving point again containing the parts of the moving point inside the moving region. **Atinstant** evaluates the moving object at a particular instant of time, returning an (*instant*, α) pair. **Inst** and **val** allow access to the components of such pairs. **Duration** returns the total length of time intervals in a *periods* value (say, in seconds). Operator **theinstant** constructs instants of time for a variable number of integer arguments (here four) in the order year, month, day, hour, minute, and second returning the first instant of time of such a time interval.

The following queries can then be formulated:

1. What was the route taken by the car “DO-GL 871”?

```
SELECT trajectory(trip) AS route
FROM cars WHERE license_plate = “DO-GL 871”
```

2. What was the total area swept by the fog region with identifier “F276”?

```
SELECT traversed(area) AS fogarea
FROM weather WHERE id = "F276"
```

3. How many cars stayed in the fog area for more than 30 minutes?

```
SELECT count(*)
FROM cars AS c, weather AS w
WHERE duration(deftime(intersection(c.trip, w.area))) > 1800
```

4. Where was the fog area at 5pm (on the respective day January 8, 2007)?

```
SELECT val(atinstant(area, theinstant(2007, 1, 8, 17)))
FROM weather WHERE id = "F276"
```

Goals in the Design of Types and Operations

The examples have illustrated the basic approach of using spatio-temporal data types. Starting from this idea, the question is how exactly data types and operations should be chosen. Such a systematic design was given in [9] and the types and operations above are already part of that design. The design pursues the following goals:

- *Closure of type system*. Type constructors should be applied systematically and consistently. This means in particular:
  - For all base types of interest, there exist related temporal ("moving") types.
  - For all temporal types (whose values are functions from time into some domain), there exist types to represent their projection into domain and range.
- *Genericity*. There will be a large set of data types. Operations should be designed in a generic way to cover as many types as possible.
- *Consistency between nontemporal and temporal types*. The structure of a temporal type, taken at a particular instant of time, should agree with the structure of the corresponding static type.
- *Consistency between nontemporal and temporal operations*. For example,

$$\mathbf{val}(\mathbf{atinstant}(\mathbf{intersection}(mp, mr), t)) = \mathbf{intersection}(\mathbf{val}(\mathbf{atinstant}(mp, t)), \mathbf{val}(\mathbf{atinstant}(mr, t)))$$

Abstract and Discrete Model

Before considering the type system in more detail, one should understand the meaning of data types. There exists a choice at what level one defines the semantics of types. For example, a moving point could be viewed in two ways:

- A continuous function from time (viewed as isomorphic to the real numbers) into the Euclidean plane, i.e., a function $f\colon \mathbb{R} \to \mathbb{R}^2$.
- A polyline in the 3D space representing such a function.

The essential difference is that in the first case, one defines the semantics of the type in terms of infinite sets without fixing a finite representation. In the second case a finite representation is chosen. The first is called an *abstract model* and the second a *discrete model*. Note that there are many discrete models for a given abstract model. For example, a moving point might also be represented as a sequence of splines. The properties of such models can be summarized as follows:

- Abstract models are mathematically simple, elegant, and uniform, but not directly implementable.
- Discrete models are more complex and heterogeneous, but can be implemented.

As a consequence, the design of spatio-temporal data types proceeds in two steps: First, an abstract model of types and operations is designed. Second, a discrete model to represent (a large part of) the abstract model is constructed.

Type System

The structure of the type system is illustrated in Figure 2. It reflects the design goals stated above.

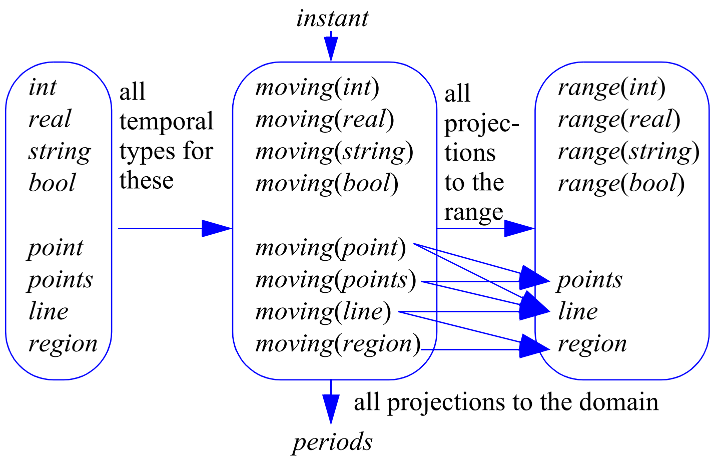

Figure 2: Structure of the type system

Projections of standard types are represented as sets of disjoint intervals over the respective base type; the *range* type constructor yields such types *range*(α) for base type α. Projections of time dependent geometries can generally be of different data types. For example, a moving point can “jump around”, i.e., change position in discrete steps, yielding a *points* value (set of points) as a projection. Or it can move continuously, yielding a *line* value (a curve in the plane). Note that *line* and *region* values can have multiple components, so that the respective projection operations are closed. The *intime*(α) types have been omitted in this figure.

Operations

The design of operations proceeds in three steps:

1. Carefully design operations for nontemporal types, using generic definition techniques.
2. By a technique called lifting make them all time dependent in a way consistent with the static definition.
3. Add specialized operations for the temporal types.

There is a comprehensive set of operations first on the nontemporal types having classes of operations such as predicates, set operations, aggregate, numeric, distance and direction operations. Second, these are all lifted which means each of their arguments may become time dependent which makes the result time dependent as well. Third, specialized operations on temporal types have classes of operations addressing projection to domain and range, interaction with values in domain and range, and operations to deal with rate of change (e.g. derivative).

Implementation

Implementation is based on the discrete model proposed in [6] using algorithms for the operations studied in [2]. The discrete model uses the so-called *sliced representation* as illustrated in Figure 3. A temporal function value is represented as a time-ordered

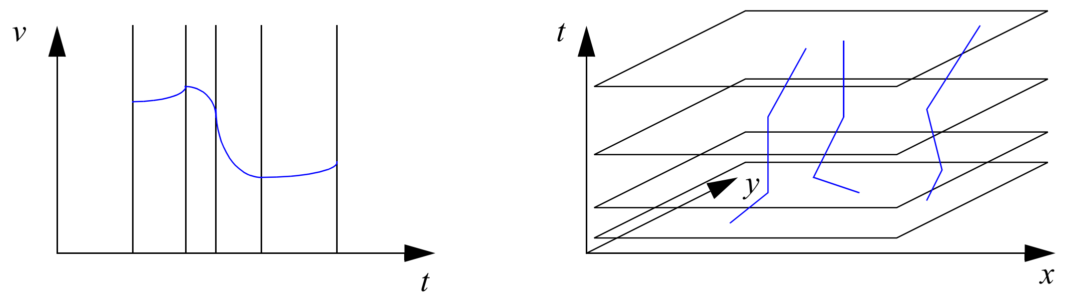

Figure 3: Sliced representation of a *moving*(*real*) and a *moving*(*points*) value

sequence of *units* where each unit has an associated time interval and time intervals of different units are disjoint. Each unit is capable of representing a piece of the moving object by a "simple" function. Simple functions are linear functions for moving points or regions, and quadratic polynomials (or square roots of such) for moving reals, for example.

Within a database system, an extension module (data blade, cartridge, extender, etc.) can be provided offering implementations of such types and operations. The sliced representation is basically stored in an array of units.[1] Because values of moving object types can be large and complex, the DBMS must provide suitable storage techniques for managing large objects. A large part of this design has been implemented prototypically in the SECONDO extensible DBMS [1] which is available for download [12].

Some Further Work on Moving Object Languages

Moving Objects in Networks

The model of spatio-temporal data types presented in the previous section has been extended to model movement in networks [8]. A network is modeled as a set of routes and junctions between routes. Four data types *gpoint*, *gline*, *moving*(*gpoint*) and *moving*(*gline*) are introduced to represent static and moving network positions and regions, respectively. Representative entities on a highway network would be gas stations, construction areas, vehicles, or traffic jams, for example. Some advantages over a model with free movement

1. It is a bit more complicated in case of variable size units as for a moving region, for example.

are that descriptions of moving objects become much more compact (as they do not contain geometries any more), that relationships between moving objects and the underlying network can be easily used in queries, and that connectivity of the network is considered, e.g. for network distance or shortest path computations.

Spatio-Temporal Predicates and Developments

Erwig and Schneider [3, 4] extend the spatio-temporal data type approach by considering developments of topological relationships over time. They develop a language to describe such developments in predicates that can then be used for filter and join conditions on moving objects. For example, consider an air plane traversing a storm area. The topological relationship between the moving point and the moving region will be *disjoint* for a period of time, then *meet* for an instant, then *inside* for a time interval, then *meet* again and finally *disjoint* again. The framework first allows one to obtain basic spatio-temporal predicates by aggregating a static topological relationship over all instants of a time interval. This is essentially done by lifting (as explained above) and existential or universal quantification. Existing predicates can then be sequentially composed to derive new predicates. For example, one may define a predicate **Cross** to describe a development like the passing of the air plane through the storm as:

$$\mathbf{Cross} := \mathbf{Disjoint} \rhd \mathbf{meet} \rhd \mathbf{Inside} \rhd \mathbf{meet} \rhd \mathbf{Disjoint}$$

Uncertain Trajectories

A *moving point* (Figure 1) in the 2D + time space is in the literature often called a *trajectory*. If we represent it discretely as a polyline and take an uncertainty threshold into account, geometrically we obtain the shape of a kind of slanted cylinder (Figure 4). It is

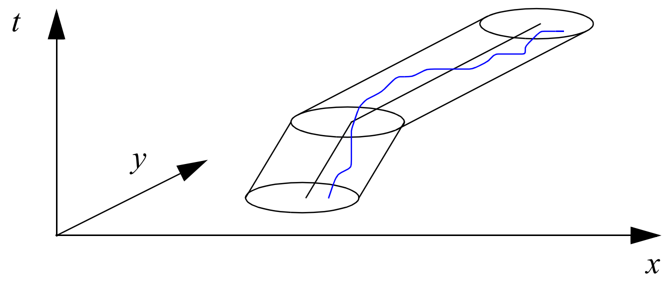

Figure 4: Geometry of an uncertain trajectory

only known that the real position is somewhere inside this volume. Based on this model, Trajcevski et al. [14] have defined a set of predicates between a trajectory and a region in space taking uncertainty and aggregation over time into account, namely

PossiblySometimeInside	**PossiblyAlwaysInside**
SometimePossiblyInside	**AlwaysPossiblyInside**

DefinitelySometimeInside	**DefinitelyAlwaysInside**
SometimeDefinitelyInside	**AlwaysDefinitelyInside**

They also give algorithms for evaluating such predicates.

A text book covering the topics presented in this article in more detail is [10].

KEY APPLICATIONS

Query languages of the first kind, that is, for querying current and near future movement like the MOST model described, are the foundation for location-based services. Service providers can keep track of the positions of mobile users and notify them of upcoming service offers even some time ahead. For example, gas stations, hotels, shopping centres, sightseeing spots, or hospitals in case of an emergency might be interesting services for car travelers.

Several applications need to keep track of the current positions of a large collection of moving objects, for example, logistics companies, parcel delivery services, taxi fleet management, public transport systems, air traffic control. Marine mammals or other animals are traced in biological applications. Obviously, the military is also interested in keeping track of fighting units in battlefield management.

Query languages of the second kind - for querying history of movement - are needed for more complex analyses of recorded movements. For example, in air traffic control one may go back in time to any particular instant or period to analyse dangerous situations or even accidents. Logistics companies may analyze the paths taken by their delivery vehicles to determine whether optimizations are possible. Public transport systems in a city may be analyzed to understand reachability of any place in the city at different periods of the day. Movements of animals may be analyzed in biological studies. Historical modeling may represent movements of people or tribes and actually animate and query such movements over the centuries.

Query languages of the second kind not only support moving point entities but also moving regions. Hence also developments of areas on the surface of the earth may be modeled and analyzed like the deforestation of the Amazone rain forest, the Ozone hole, development of forest fires or oil spills over time, and so forth.

FUTURE DIRECTIONS

Recent research in databases has often addressed specific query types like continuous range queries or nearest neighbour queries, and then focused on designing efficient algorithms for them. An integration of the many specific query types into complete language designs as presented in this entry is still lacking. Uncertainty may be treated more completely also in the approaches for querying history of movement. A seemless query language for querying past, present, and near future would also be desirable.

CROSS REFERENCES

1. Spatio-Temporal Data Types
2. Trajectory

RECOMMENDED READING

[1] Almeida, V.T., Güting, R.H. and Behr, T. (2006). Querying Moving Objects in SECONDO. *Proc. Mobile Data Management Conf.*, 47-51.

[2] Cotelo Lema, J.A., Forlizzi, L., Güting, R.H., Nardelli, E., and Schneider, M. (2003). Algorithms for Moving Object Databases. *The Computer Journal*, 46(6), 680-712.

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

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

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

[6] Forlizzi, L., Güting, R.H., Nardelli, E., and Schneider, M. (2000). A Data Model and Data Structures for Moving Objects Databases. *Proc. ACM SIGMOD Conf.* (Dallas, Texas, USA), 319-330.

[7] Grumbach, S., Rigaux, P., and Segoufin, L. (1998). The DEDALE System for Complex Spatial Queries. *Proc. ACM SIGMOD Conf. (Seattle, Washington)*, 213-224.

[8] Güting, R.H., Almeida, V.T. de, and Ding, Z. (2006). Modeling and Querying Moving Objects in Networks. *VLDB Journal,* 15(2), 165-190.

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

[10] Güting, R.H. and Schneider, M. (2005). *Moving Objects Databases*. Morgan Kaufmann Publishers.

[11] Rigaux, P., M. Scholl, L. Segoufin, and S. Grumbach (2003). Building a Constraint-Based Spatial Database System: Model, Languages, and Implementation. *Information Systems*, 28(6), 563-595.

[12] SECONDO System. Available for download at http://www.informatik.fernuni-hagen.de/import/pi4/Secondo.html/

[13] Sistla, A.P., Wolfson, O., Chamberlain, S., and Dao, S. (1997). Modeling and Querying Moving Objects. *Proc. 13th Int. Conf. on Data Engineering* (ICDE, Birmingham, U.K.), 422-432.

[14] Trajcevski, G., Wolfson, O., Hinrichs K., and Chamberlain, S. (2004). Managing Uncertainty in Moving Objects Databases. *ACM Transactions on Database Systems* (*TODS*), 29(3), 463-507.

[15] Wolfson, O., Chamberlain, S., Dao, S., Jiang, L., and Mendez, G. (1998). Cost and Imprecision in Modeling the Position of Moving Objects. *Proc. 14th Int. Conf. on Data Engineering* (ICDE, Orlando, Florida), 588-596.

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

[17] Wolfson, O., Xu, B., Chamberlain, S., and Jiang, L. (1998). Moving Objects Databases: Issues and Solutions. *Proc. 10th Int. Conf. on Scientific and Statistical Database Management* (SSDBM, Capri, Italy), 111-122.