Managing Uncertainty in Moving Objects Databases

GOCE TRAJCEVSKI
Northwestern University
OURI WOLFSON
University of Illinois at Chicago
KLAUS HINRICHS
Westfälische Wilhelms-Universität Münster
and
SAM CHAMBERLAIN
Army Research Laboratory

This article addresses the problem of managing Moving Objects Databases (MODs) which capture the inherent imprecision of the information about the moving object's location at a given time. We deal systematically with the issues of constructing and representing the *trajectories* of moving objects and querying the MOD. We propose to model an uncertain trajectory as a three-dimensional (3D) cylindrical body and we introduce a set of novel but natural spatio-temporal *operators* which capture the *uncertainty* and are used to express spatio-temporal range queries. We devise and analyze algorithms for processing the operators and demonstrate that the model incorporates the uncertainty in a manner which enables efficient querying, thus striking a balance between the modeling power and computational efficiency.

We address some implementation aspects which we experienced in our DOMINO project, as a part of which the operators that we introduce have been implemented. We also report on some experimental observations of a practical relevance.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*spatio-temporal data*; *uncertainty*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General

General Terms: Algorithms

Additional Key Words and Phrases: Moving Objects Databases

Authors' addresses: G. Trajcevski, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208; email: goce@ece.northwestern.edu; O. Wolfson, Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607; email: wolfson@cs.uic.edu; K. Hinrichs, FB 10—Institut für Informatik, Westfälische Wilhelms-Universität Münster, Einstrastr. 62, 48149 Münster, Germany; email: khh@uni-muenster.de; S. Chamberlain, Army Research Laboratory, Aberdeen Proving Ground, MD 21005-5067; email: wildman@arl.mil.

1. INTRODUCTION AND MOTIVATION

Miniaturization of computing devices and advances in wireless communication and sensor technology are some of the forces that are propagating computing from the stationary desktop to the mobile outdoors. Important classes of new applications that will be enabled by this revolutionary development include location-based services, tourist services, mobile electronic commerce, and the digital battlefield [Brundick and Hartwig 1997; Chamberlain 1995; Pitoura and Samaras 2001; Want and Schlidt 2001]. Many existing applications will also benefit from this development: transportation and air traffic control, weather forecasting, emergency response, mobile resource management, and mobile workforce [Hartwig et al. 1996; Hightower and Borrielo 2001]. Location management, that is, the management of transient location information, is an enabling technology for all these applications. It is also a fundamental component of other technologies such as fly-through visualization, context awareness, augmented reality, cellular communication, and dynamic resource discovery.

Database researchers have addressed some aspects of the problem of modeling and querying the location of moving objects. The largest efforts were made in the area of access methods. Aside from purely spatial (Gaede and Günther [1998] surveyed 50+ structures) and temporal databases [Tansel et al. 1993], there are several recent results which tackle various problems of indexing *spatio-temporal* objects and dynamic attributes [Agarwal et al. 2000; Kollios et al. 1999a, 1999b; Pfoser et al. 1999; Saltenis et al. 2000; Tayeb et al. 1998; Theodoridis et al. 1999a, 1999b]. Representing and querying the location of moving objects as a function of time is introduced in Sistla et al. [1997], and [Wolfson et al. 1998, 1999] addressed policies for updating and modeling imprecision and communication costs. Modeling and querying location uncertainties due to sampling and GPS imprecision was presented in Pfoser and Jensen [1999]. Algebraic specifications of a system of abstract data types, their constructors and a set of operations were given in Erwig et al. [1998], Forlizzi et al. [2000], and Güting et al. [2000].

In this article we deal in a systematic way with the issue of *uncertainty* of the *trajectory* of a moving object. Uncertainty is an inherent aspect in databases which store information about the location of moving objects. Due to continuous motion and network delays, the database location of a moving object will not always precisely represent its real location. Unless uncertainty is captured in the model and the query language, the burden of factoring uncertainty into answers to queries is left to the user.

Traditionally, the trajectory of a moving object was modeled as a polyline in three-dimensional space (two dimensions for geography, and one for time). In this article, in order to capture uncertainty we model the trajectory as a cylindrical volume in 3D. Typically, spatio-temporal range queries ask for the objects that are inside a particular region, during a particular time interval. However, for the moving objects one may query the objects that are inside the region *sometime* during the time interval, or for the ones that are *always* inside during the time interval. Similarly, given the uncertainty of the objects whereabouts, one may query the objects that are *possibly* inside the region or

the ones that are *definitely* there. For example, a trucking company may be interested in a query like:

Q1: *"Retrieve the current location of the delivery trucks that will* possibly *be inside a region R,* sometime *between 3:00PM and 3:15PM."*

As another example, a military analyst may be concerned with:

Q2: *"Retrieve the number of tanks which will* definitely *be inside the region R* sometime *between 1:30PM and 1:45PM."*

and a police dispatcher may want to:

Q3: *"Retrieve the squad cars which will* possibly *be inside the region R,* always *between 2:30AM and 2:40AM."*

We provide the syntax of the operators for spatio-temporal range queries with uncertainty, and their processing algorithms. It turns out that most of these algorithms have a strong geometric flavor.

The model and the operators that we introduce in this paper have been implemented and integrated in our DOMINO system [Trajcevski et al. 2002a; Wolfson et al. 2002].

When implementing a real-life application, one needs to consider the existing, commercially available, technology. Due to the recent trend for supporting *universal applications*, commercial Object-Relational Database Management Systems (ORDBMSs) are now offering new complex data types, inheritance, user-defined routines which implement operators/methods over the user-defined types and various extensions to SQL ([Carey et al. 1999; Oracle Corporation 2000]). In particular, IBM's DB2 Spatial Extender [Davis 1998], Oracle's Spatial Cartridge [Oracle Corporation 2000], and Informix Spatial DataBlade [Team 1999] provide several two-dimensional (2D)—spatial types (e.g., line, polyline, polygon, . . .) and include a set of predicates (e.g., *intersects, contains*) and functions for spatial calculations (e.g., *distance*). The ability to employ a mature technology brings the benefits of scalability and usage of tested and optimized components; however, extending it toward new problem domains is always a welcome challenge. We implemented our operators as UDFs in *Oracle9i* as part of our DOMINO system. We conducted several experiments in order to determine values for parameters that are of interest in a practical application. For example, we generated over 1000 trajectories using a map of Chicagoland and analyzed their average size—approximately 7.25 line segments/mi. Thus, for fleets of thousands of vehicles the trajectories database can indeed be stored in main memory. We also experimented with some implementation alternatives and we conducted real-drive testing of the (impact of the) uncertainty model on the updates of the trajectory. Thus, we demonstrated that our approach can be implemented on top of off-the-shelf ORDBMSs and gained some beneficial practical experiences.

Throughout this article we address both "sides of the coin"—the theoretical aspects of the impact of the uncertainty of the moving object's whereabouts, as well as some practical aspects of incorporating it in a real application. Our

main contributions can be summarized as follows:

(1) We introduce a trajectory model and its construction based on electronic maps.
(2) We present a model of the uncertainty associated with the moving object's trajectory.
(3) We introduce a set of operators for querying trajectories with uncertainty. We provide both linguistic constructs and processing algorithms, and we analyze the complexity of each algorithm.
(4) We address the issues of incorporating our model in a real system and we present some experimental observations of practical significance for applications which deal with moving objects management.

The rest of the article is structured as follows. In Section 2 we define the model of a trajectory and show how it can be constructed based on electronic maps. Section 3 defines the uncertainty concepts for a trajectory. In Section 4 we present the impact of incorporating the uncertainty in the trajectory model. We give the *syntax* and the *semantics* of the new operators for querying trajectories with uncertainty and illustrate the relationships that exist among them. Section 5, the longest one of this article, is the procedural counterpart of Section 4 and it provides the theoretical analysis of the algorithms for processing our operators. In Section 6, on the other hand, we discuss practical implementation issues and present some experimental observations for the database server part of the DOMINO system. Section 7 positions the article with respect to the relevant works and Section 8 gives the concluding remarks and outlines the directions for future work. For clarity of presentation, we have moved the lengthier proofs and calculations to the Appendix.

2. REPRESENTING AND CONSTRUCTING THE TRAJECTORIES

In this section we define our model of a *trajectory*, and we describe how to construct it from the data available in electronic maps. We also introduce the basic terminology which will be used in the rest of this article.

2.1 Basic Definitions

In order to capture the spatio-temporal nature of a moving object, we use the following:

Definition 2.1. A *trajectory* of a moving object is a polyline in three-dimensional space (two-dimensional geometry, plus time), represented as a sequence of points $(x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_n, y_n, t_n)$ $(t_1 < t_2 < \cdots < t_n)$. For a given trajectory T, its projection on the XY plane is called the *route* of T. The object is at (x_i, y_i) at time t_i, and during each segment $[t_i, t_{i+1}]$, it is assumed to move along a straight line from (x_i, y_i) to (x_{i+1}, y_{i+1}) at a constant speed.

A trajectory defines the location of a moving object as an implicit function of time, and the speed of the moving object along the line segment between (x_i, y_i)

and (x_{i+1}, y_{i+1}) can be calculated as

$$v_i = \frac{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}{t_{i+1} - t_i}.$$

Definition 2.2. Given a trajectory T, the *expected location* of the object at a point in time t between t_i and t_{i+1} $(1 \leq i < n)$ is obtained by a linear interpolation between (x_i, y_i) and (x_{i+1}, y_{i+1}).

Note that a trajectory can represent both the past and future motion of objects. As far as future time is concerned, one can think of the trajectory as a set of points describing the *motion plan* of the object. Namely, we have a set of points that the object is going to visit, and we assume that between the points the object is moving along the shortest path. Given an electronic map, along with the beginning time of the object's motion, we construct a trajectory as a superset of the set of the given—"to-be-visited"—points. In order to explain how we do so, we need to define an electronic map (or a map, for brevity).

Definition 2.3. A *map* is a graph, represented as a relation where each tuple corresponds to a block with the following attributes:

—*Polyline:* Each block is a polygonal line segment. Polyline gives the sequence of the endpoints: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

—*Length:* Length of the block.

—*Fid:* The block *id* number.

—*Drive_Time:* Typical drive time from one end of the block to the other, in minutes.

—*Level:* An integer value denoting a relative level of a block.

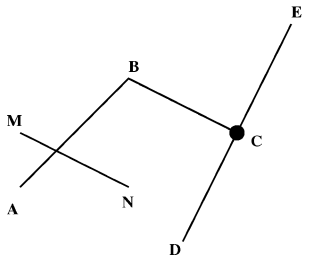
—Plus, among others, a set of geo-coding attributes which enable translating between a *(x, y)* coordinate and an address, such as *"1030 North State St."*: (e.g., – *L_f_add:* Left side from street number.)

Such maps are provided by, among others, Geographic Data Technology[1] (GDT) Co. [2000]. An intersection of two streets corresponds to the endpoint of the four block-polylines. On the other hand, the intersections among blocks in points different from the blocks' endpoints can only occur if the blocks are at different levels (e.g., a highway overpassing a regular street). In other words, if two blocks intersect, then they are at different levels.

The concepts are illustrated in Figure 1, which gives an example of four blocks and their respective attributes. Observe that three of the blocks intersect at the point C and they are all at Level 0. On the other hand, the block $\overline{MN}$ at Level 1 overpasses the block $\overline{ABC}$.

The route of a moving object O is specified by giving the starting address or (x, y) coordinate, namely, the *start_point*; the starting time; and the destination address or (x, y) coordinate, namely, the *end_point*. An external routine, available in most Geographic Information Systems, which we assume is given a priori, computes the shortest cost (distance or travel-time) path in the map

[1]Website: www.geographic.com.



Fid	Polyline	Length	Drive Time	Level
ABC	(3,3)(5,5)(7,4)	1 mile	2 min	0
DC	(6,2)(7,4)	0.6 mile	1.5 min	0
CE	(7,4)(8,6)	0.6 mile	1.2 min	0
MN	(3,4)(5,3)	0.5 mile	0.9 min	1

Fig. 1. Blocks in a map.

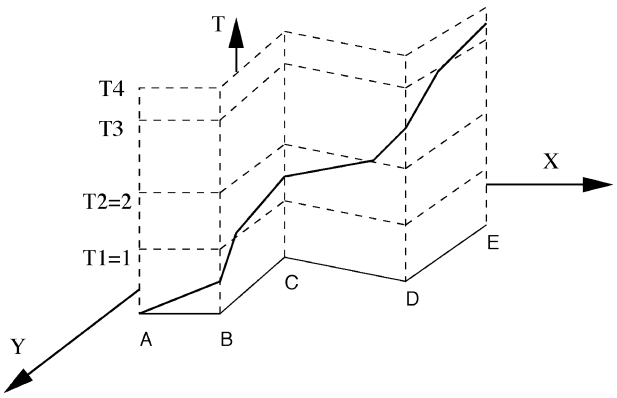


Fig. 2. Trajectory and its construction based on a route and speed profiles.

graph. This path, denoted $P(O)$, is a sequence of blocks (edges), that is, tuples of the map. Since $P(O)$ is a path in the map graph, the endpoint of one block polyline is the beginning point of the next block polyline. Thus, the route represented by $P(O)$ is a polyline denoted by $L(O)$. Given that the trip has a starting time, we compute the trajectory by computing for each straight line segment on $L(O)$ the time at which the object O will arrive to the point at the end of the segment (cf. Trajcevski et al. [2002b]). For this purpose, the only relevant attributes of the tuples in $P(O)$ are *Polyline* and *Drive_Time*. However, the *Drive_Time* attribute in the available maps is *static*, in the sense that it does not consider the different speed patterns within a period of time, say, a day. Clearly, the objects will move more slowly during the rush hours as opposed to nonrush hours. Therefore, we need to take into consideration an attribute like *Speed_Profile*, the values of which can be obtained by monitoring the traffic. The only modification is that now we need to employ a *time-dependent* version of the shortest cost routine, similar in spirit to the one presented in Dreyfus [1969] (essentially, an A^* extension of Dijkstra's algorithm), where the cost of an edge in a graph depends on the start time to travel along that edge.

The concepts are illustrated in Figure 2. Observe how the moving object enters two different speed profiles while traveling along the route segments $\overline{BC}$, $\overline{CD}$, and $\overline{DE}$. Thus, a trajectory may have more than one segment corresponding to a given route segment.

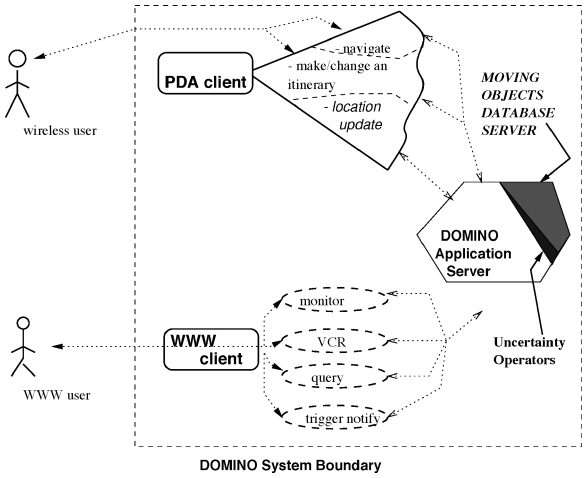


Fig. 3. Components of the DOMINO system.

Observe that a trajectory can be constructed based on past motion. Specifically, consider a set of 3D points (x_1, y_1, t_1), (x_2, y_2, t_2), $\ldots$, (x_n, y_n, t_n) which were transmitted by a moving object periodically, during its past motion. One can construct a trajectory by first "snapping" the points on the road network, then simply connecting the snapped points with the shortest path on the map.

Finally, let us point out that the proposed model of the trajectory can be represented as a user-defined type (UDT) in an ORDBMS. We define the `trajectory` as a row type `LIST` of `point`, which is another row type having `X,Y,` and `T` attributes. Thus, we have a schema for representing moving objects trajectories

```
MOD(oid,trajectory,...other static attributes),
```

where the *oid* is a unique identifier assigned for each moving object and the *trajectory* attribute is as described above. The "`...other static attributes...`" may include, for example, year, make, model, mileage, etc.

Figure 3 gives a "birds-eye" view of the main modules of the DOMINO system [Trajcevski et al. 2002a; Wolfson et al. 2002]. There are two basic categories of users of the system: the *mobile* ones, which are assumed to have some minimal processing power on-board the moving vehicle (e.g., a PDA); and "static" ones, which access the system through a Web browser. Each user can specify the start-time, the start-location, and the end-location for the trip either by entering the address on the PDA or by clicking the locations on the map (for a Web browser interface), and the server will generate the trajectory, as we already explained, which will be transmitted to the user. The trajectories are stored in the MOD server module, which constitutes the database that is used for answering queries. The DOMINO Application Server provides several services (e.g., an interface for the query specification for the users, regeneration of the future portion of the trajectories upon updates, etc.). As illustrated in Figure 3, the operators that we introduce in this article are part of the MOD server.

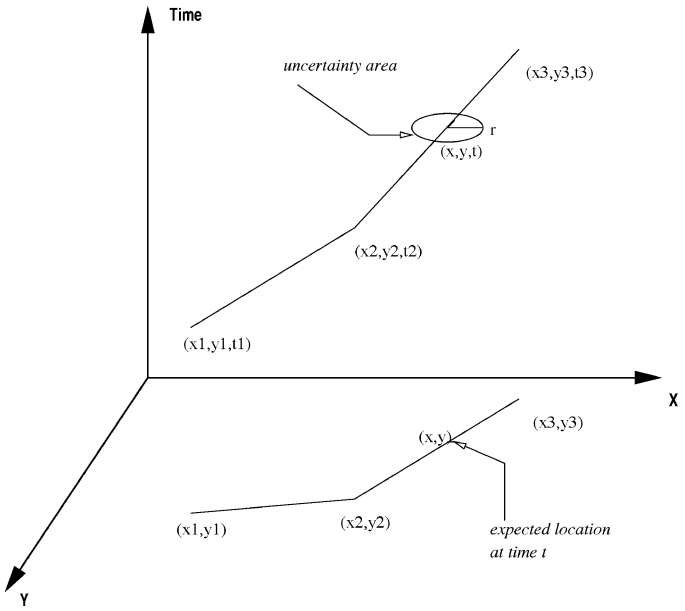


Fig. 4. Uncertainty area of a trajectory.

3. UNCERTAINTY CONCEPTS FOR TRAJECTORIES

In this section we give a formal presentation of the uncertainty model of a trajectory.

An uncertain trajectory is obtained by associating an uncertainty threshold r with each line segment of the trajectory. For a given motion plan, the line segment together with the uncertainty threshold constitutes an "agreement" between the moving object and the server. The agreement specifies the following: the moving object will update the server if and only if it deviates from its expected location (according to the trajectory) by r or more. How does the moving object compute the deviation at any point in time? Its on-board computer receives a GPS update every 2 s, so it knows its actual location. Also, it has the trajectory, so by interpolation it can compute its expected location at any point in time. The deviation is simply the distance between the actual and the expected location.

Observe that, due to the inherent imprecision of the GPS, the uncertainty can be associated with the past trajectories of the moving objects, when constructing them as specified in Section 2.

Definition 3.1. Let r denote a positive real number and T denote a trajectory between the times t_1 and t_n. An *uncertain trajectory UTr* is the pair (T, r). r is called the *uncertainty threshold*.

For each point (x, y, t) along T, its *r-uncertainty area* (or the *uncertainty area* for short) is a horizontal disk (i.e., the circle and its interior) with radius r centered at (x, y, t), where (x, y) is the expected location at time $t \in [t_1, t_n]$.

Definition 3.1 is illustrated in Figure 4.

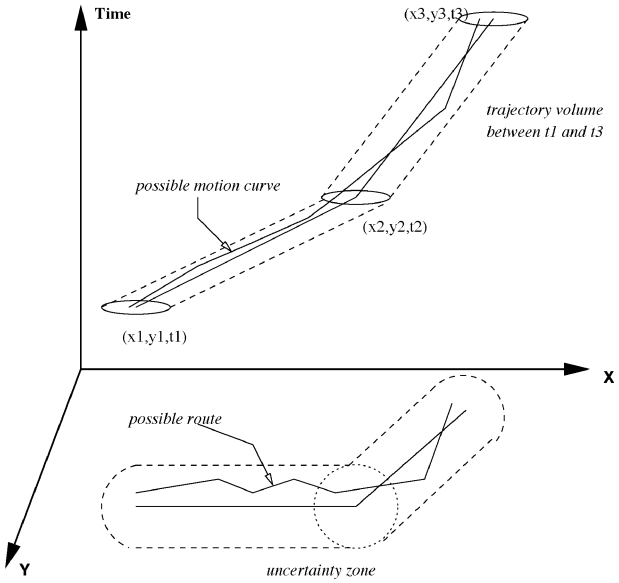


Fig. 5. Possible motion curve and trajectory volume.

Note that our model of uncertainty is a little simpler than the one proposed in Pfoser and Jensen [1999]. There, the uncertainty associated with the location of an object traveling between two endpoints of a line segment was an ellipse with foci at the endpoints.

Definition 3.2. Let $UTr = (T, r)$ be an uncertain trajectory between t_1 and t_n. A *Possible Motion Curve* PMC^T of T is any *continuous function* f_{PMC^T}: *Time* $\rightarrow R^2$ defined on the interval $[t_1, t_n]$ such that for any $t \in [t_1, t_n]$, the 3D point $(f_{PMC^T}(t), t)$ is inside the uncertainty area of the expected location at time t.

Intuitively, a possible motion curve describes a *possible route* with its associated times, which a moving object may take, without generating an update. In other words, a moving object need not update the database as long as it is on some possible motion curve of its uncertain trajectory (see Figure 5). We will refer to the *XY* projection of a possible motion curve as a *possible route*.

Definition 3.3. Given an uncertain trajectory $UTr = (T, r)$ and two endpoints (x_i, y_i, t_i), $(x_{i+1}, y_{i+1}, t_{i+1}) \in T$, the *trajectory volume* of *UTr between* t_i and t_{i+1} is the union of all the disks with radius r centered at the points along the line segment $\overline{(x_i, y_i, t_i), (x_{i+1}, y_{i+1}, t_{i+1})}$. The *XY* projection of the trajectory volume between t_i and t_{i+1} is called an *uncertainty segment*. The *trajectory volume* of *UTr* is the union of all the trajectory volumes between t_1 and t_n, where t_1 and t_n denote the begin and end time-values of T, and the *uncertainty zone* of *UTr* is the union of all the uncertainty segments between t_1 and t_n.

Definitions 3.2 and 3.3 are illustrated in Figure 5.

Viewed in 3D, a trajectory volume between t_1 and t_n is a sequence of volumes, each bounded by a sheared cylinder. The axis of each sheared cylinder coincides with the respective straight line segment of the trajectory, and the bases are the disks with radius r in the horizontal planes t_i and t_{i+1} $(1 \leq i < n)$. Observe that the sheared cylinder is different from a tilted cylinder. The intersection of a tilted cylinder with a horizontal plane (parallel to the XY plane) yields an ellipse, whereas the intersection of our sheared cylinder with such a plane yields a circle. Let v_i^x and v_i^y denote the x and y components of the velocity of a moving object along the ith segment of the route (i.e., between (x_i, y_i) and (x_{i+1}, y_{i+1})). Given the Definitions 3.1 and 3.3, the ith segment of the trajectory volume can be characterized by the following set of inequalities:

$$t_i \leq t \leq t_{i+1}, \tag{1}$$

$$\left(x - \left(x_i + v_i^x \cdot t\right)\right)^2 + \left(y - \left(y_i + v_i^y \cdot t\right)\right)^2 \leq r^2. \tag{2}$$

As a straightforward consequence of the Definitions 3.2 and 3.3, we have the following:

THEOREM 3.4. *Let UTr(T,r) denote an uncertain trajectory and let VTr denote its trajectory volume. If PMC^T is a possible motion curve of T, then $PMC^T \subset VTr$.*

We conclude this section with the observation that all we need to represent the uncertain trajectories in an ORDBMS is to extend the schema given in Section 2 with one more attribute `uncertainty` of type `real`. Thus, we have:

```
MOD(oid,trajectory,uncertainty,...other static attributes...).
```

4. QUERYING MOVING OBJECTS WITH UNCERTAINTY

In this section we introduce two categories of operators for querying moving objects with uncertainty. The first category, discussed in Section 4.1, deals with *point* queries and consists of two operators which pertain to a single trajectory. The second category, discussed in Section 4.2, is a set of six (Boolean) predicates which give a qualitative description of a relative position of a moving object with respect to a region, within a given time interval. Thus, each one of these operators corresponds to a spatio-temporal *range* query.

4.1 Point Queries

The two operators for point queries are defined as follows:

— *Where_At(trajectory T, time t)—returns the* expected location *on the route of T at* time t.

— *When_At(trajectory T, location l)—returns the* times *at which the moving object whose trajectory is T, is expected to be at* location l. The answer may be a set of times, in case the moving object passes through a certain point more than once. If the location $l = (x_l, y_l)$ is not on the route of the trajectory T, we find the set of all the points C on this route which are *closest* to l. The function then returns the set of times at which the object is expected to be at each point in C. Let us point out that, in case the object is stationary during

some time-interval, its trajectory is a vertical line segment. In such case(s), the *When_at* operator may also return time-interval(s).

The algorithms which implement the point query operators are straightforward. The *Where_at* operator can be implemented in $O(\log n)$ by a simple binary search (note that, by Definition 2.1, the points of the trajectory T are sorted on their *time* dimension), where n is the number of line segments of the trajectory. As for the *When_at* operator, it can be implemented in linear time—$O(n)$, by examining each line segment of a trajectory.

As we will demonstrate in Section 6, any reasonable trajectory has no more than several thousand line segments and it can be stored in main memory which, in turn, implies that the processing time of each one of the above operators is acceptable.

4.2 Operators for Spatio-Temporal Range Queries

The second category of operators is a set of conditions (i.e., Boolean predicates). Each condition is satisfied if the moving object is *inside* (which is the property/predicate of interest) a given region R, during a given time-interval $[t_b, t_e]$. Clearly, this corresponds to a spatio-temporal range query. But then, why more than one operator? The answer is twofold: (1) The location of the object changes continuously; hence one may ask if the condition is satisfied *sometime* or *always* within $[t_b, t_e]$; (2) due to the uncertainty, the object may *possibly* satisfy the condition or it may *definitely* do so, at a particular time-point $t \in [t_b, t_e]$. If there exists some possible motion curve PMC^T which at the time t is inside the region R, there is a possibility that the moving object will take PMC^T as its actual motion and will be inside R at t. However, this need not be the case, since the moving object may have chosen another possible motion curve as its actual motion. Similarly, if every possible motion curve PMC^T is inside the region R at the time t, then regardless of which one describes the actual object's motion, the object is guaranteed to be inside the region R at time t. Given the trajectory volume VTr of the trajectory T as a domain (recall that, by Theorem 1, VTr is, in a sense, the closure of all the possible motion curves of T), the meaning of *possibly* corresponds to $\exists PMC^T \subset VTr$ and the meaning of *definitely* corresponds to $\forall PMC^T \subset VTr$.

Thus, we have two domains of quantification, with two quantifiers in each. Combining all of them, since the order of quantification matters, yields $2^2 \cdot 2! = 8$ operators.

Now we proceed with the declarative specifications of the operators. Throughout this section, we will assume that the region R is a connected, closed, and bounded subset of the XY plane.

In what follows, we let PMC^T denote a possible motion curve, of a given uncertain trajectory $UTr = (T, r)$.

—*Possibly_Sometime_Inside(T, R, t_b, t_e)*. This is *true* iff there exists a possible motion curve PMC^T and there exists a time $t \in [t_b, t_e]$ such that PMC^T, at the time t, is inside the region R. In other words:

$$(\exists PMC^T)(\exists t) Inside(R, PMC^T, t).$$

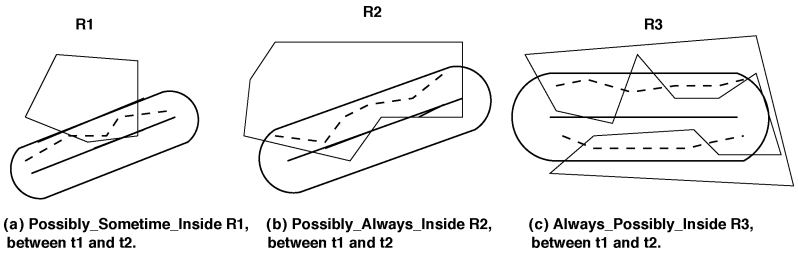


Fig. 6. *Possible* positions of a moving point with respect to region R_i.

Intuitively, the truth of the predicate means that the moving object may take a possible route, within its uncertainty zone, such that the particular possible route will intersect the query region R between the times t_b and t_e.

—*Sometime_Possibly_Inside(T, R, t_b, t_e)*. This is *true* iff there exists a time $t \in [t_b, t_e]$ and a possible motion curve PMC^T of the trajectory T which, at the time t, is inside the region R.

Observe that this operator is semantically equivalent to the *Possibly_Sometime_Inside* one, that is, $(\exists PMC^T)(\exists t)Inside(R, PMC^T, t) \equiv (\exists t)(\exists PMC^T)Inside(R, PMC^T, t)$. Similarly, it will be clear that the predicate *Definitely_Always_Inside* is equivalent to *Always_Definitely_Inside*. Therefore, in effect, we have a total of six operators for spatio-temporal range queries with uncertainty.

—*Possibly_Always_Inside(T, R, t_b, t_e)*. This is *true* iff there exists a possible motion curve PMC^T of the trajectory T which is inside the region R for every t in $[t_b, t_e]$. In other words:

$$(\exists PMC^T)(\forall t)Inside(R, PMC^T, t).$$

Now, the motion of the object is such that it may take (at least one) specific possible route, which is entirely contained within the region R, during the whole query time interval.

—*Always_Possibly_Inside(T, R, t_b, t_e)*. This is *true* iff for every time value $t \in [t_b, t_e]$, there exists some (not necessarily unique) PMC^T which is inside (or on the boundary of) the region R at t. In other words:

$$(\forall t)(\exists PMC^T)Inside(R, PMC^T, t).$$

Figure 6 illustrates the XY projection of a plausible scenario for each of the three predicates that we introduced. Dashed lines indicate the possible motion curve(s) due to which the predicates are satisfied; solid lines indicate the routes and the boundaries of the uncertainty zone.

Recall that, in First-Order Logic[2] [Genesereth and Nilsson 1987], given a predicate P, a constant A and the variables x and y, regardless of the

[2]For clarity of presentation, we have slightly abused the standard terminology of Logic Programming where lower_case letters denote variables and Upper_case letters denote constants. In our settings, it should be clear that the region R is a constant given in the query, and that PMC^T and t are variables from their respective domains.

interpretation domains, the following is a tautology:

$$(\exists x)(\forall y)P(A, x, y) \Rightarrow (\forall y)(\exists x)P(A, x, y). \quad (3)$$

Thus, the predicate *Possibly_Always_Inside* is stronger than *Always_Possibly_Inside*, in the sense that whenever *Possibly_Always_Inside* is true, *Always_Possibly_Inside* is guaranteed to be true.

Note that the converse, in general, is not true. As illustrated in Figure 6, the predicate *Always_Possibly_Inside* may be satisfied due to two or more possible motion curves, none of which satisfies *Possibly_Always_Inside* by itself. However, as the next theorem indicates, this situation cannot occur when the region R is convex.

THEOREM 4.1. *Let $UTr = (T, r)$ denote an uncertain trajectory, and t_b and t_e denote two time values. If R is convex, then Possibly_Always_Inside(T,R,t_b,t_e) is true iff Always_Possibly_Inside(T,R,t_b,t_e) is true.*

PROOF. ($\Rightarrow$) A straightforward consequence of (3) (R need not be convex for this case).

($\Leftarrow$) Assume now that the predicate *Always_Possibly_Inside* is satisfied. Let (x_I, y_I, t_I) denote the first trajectory point after t_b. In other words, the uncertainty area centered at (x_I, y_I, t_I) is the ending base of (the portion of) the first segment of the trajectory volume between t_b and t_e and the beginning base of the next (second) segment of the trajectory volume between t_b and t_e. Let S_b denote the set of all the possible motion curves of T such that $f_{PMC}(t_b) \in R$. Since *Always_Possibly_Inside* is satisfied, S_b is not empty and it may have uncountably many elements. Let PMC_b^T denote one possible motion curve from S_b and let (x_{S_b}, y_{S_b}, t_b) be a point on PMC_b^T. For similar reasons, there must exist a set, denote it S_I, of possible motion curves, such that $f_{PMC}(t_I) \in R$. Let PMC_I^T denote one of those curves (again, there may be uncountably many of them) and let (x_{S_I}, y_{S_I}, t_I) be a point on PMC_I^T.

Since R is convex, and both $(x_{S_b}, y_{S_b}) \in R$ and $(x_{S_I}, y_{S_I}) \in R$, it follows that the line segment $\overline{(x_{S_b}, y_{S_b})(x_{S_I}, y_{S_I})}$ will be entirely in R. Also, since each segment of the trajectory volume is a convex 3D body, it follows that the 3D line segment $\overline{(x_{S_b}, y_{S_b}, t_b)(x_{S_I}, y_{S_I}, t_I)}$ is entirely in the segment of the trajectory volume between t_b and t_I. This straight line segment is the desired possible motion curve between t_b and t_I. By repeating the argument for every segment between t_I and t_e we can construct a possible motion curve which is the witness of satisfying the *Possibly_Always_Inside* predicate. □

The other three predicates are specified as follows:

—*Always_Definitely_Inside(T, R, t_b, t_e)*. This is *true* iff at every time $t \in [t_b, t_e]$, every possible motion curve PMC^T of the trajectory T is in the region R. In other words:

$$(\forall t)(\forall PMC^T)Inside(R, PMC^T, t).$$

Thus, no matter which possible motion curve the object takes, it is guaranteed to be within the query region R throughout the entire interval $[t_b, t_e]$. Note that this predicate is semantically equivalent to *Definitely_Always_Inside*,

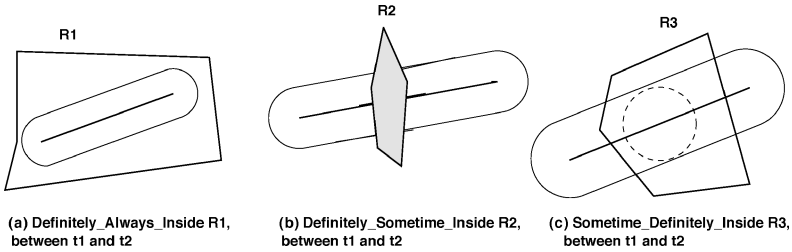


Fig. 7. *Definite* positions of a moving point with respect to region R_i.

that is:

$$(\forall t)(\forall PMC^T)Inside(R, PMC^T, t) \equiv (\forall PMC^T)(\forall t)Inside(R, PMC^T, t).$$

—*Definitely_Sometime_Inside(T, R, t_b, t_e)*. This is *true* iff for every possible motion curve PMC^T of the trajectory T, there exists some time $t \in [t_b, t_e]$ in which the particular motion curve is inside the region R. In other words:

$$(\forall PMC^T)(\exists t)Inside(R, PMC^T, t).$$

Intuitively, no matter which possible motion curve within the uncertainty zone is taken by the moving object, it will intersect the region at some time t between t_b and t_e. However, the time of the intersection may be different for different possible motion curves.

—*Sometime_Definitely_Inside(T, R, t_b, t_e)*. This is *true* iff there exists a time point $t \in [t_b, t_e]$ at which every possible route PMC^T of the trajectory T is inside the region R. In other words:

$$(\exists t)(\forall PMC^T)Inside(R, PMC^T, t).$$

Satisfaction of this predicate means that no matter which possible motion curve is taken by the moving object, at the specific time t the object will be inside the query region.

The intuition behind the last three predicates is depicted in Figure 7.

Again we observe that the predicate *Sometime_Definitely_Inside* is stronger than *Definitely_Sometime_Inside*, as a consequence of the properties of First-Order Logic (c.f. (3)). However, the above two predicates are not equivalent when the region R is convex. An example demonstrating this is given in Figure 7(b)—R_2 satisfies *Definitely_Sometime_Inside*, but it does not contain an entire uncertainty area for any time point and, consequently, it does not satisfy *Sometime_Definitely_Inside*.

The relationships among the predicates are depicted in Figure 8, where the arrow denotes an implication.

Given the set of operators that we defined, one can use an extension of SQL in an ORDBMS to pose the queries that we presented in Section 1. Recall, for example:

Q3: *"Retrieve the squad cars which will* possibly *be inside the region R,* always *between 2:30AM and 2:40AM."*

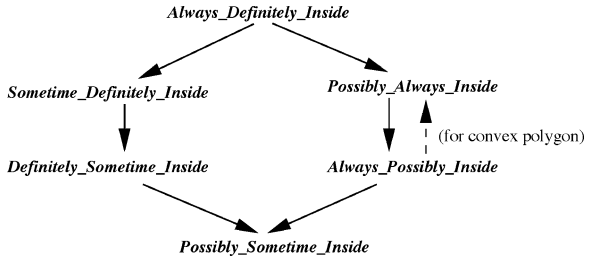


Fig. 8. Relationships among the spatio-temporal predicates.

If we assume that the dispatcher has a MOD available, the query can be stated as

```
SELECT oid
FROM  MOD
WHERE Possibly_Always_Inside(MOD.trajectory,R,2:30,2:40)
```

Even more complex query conditions can be expressed by composition of the operators. Consider, for example:

Q4: *"Retrieve all the objects which are possibly within a region R, always between the earliest*[3] *time when the object A arrives at locations* L_1 *and the latest time when it arrives at location* L_2*."*

This query can be expressed as

```
WITH Earliest(times) AS
  SELECT When_At(trajectory,L_1)
  FROM MOD
  WHERE oid = A
WITH Latest(times) AS
  SELECT When_At(trajectory,L_2)
  FROM MOD
  WHERE oid = A
SELECT M1.oid
FROM MOD as M1
WHERE Possibly_Always_Inside(M1.trajectory,R,
                             MIN(Earliest.times),
                             MAX(Latest.times))
```

In real-life applications, one cannot expect that a casual user will be familiar with the SQL syntax and it is natural to expect that there will be some form of a user interface available. We defer that discussion to Section 6.

[3]Observe that a given object may pass through a given point along its route more than once.

5. PROCESSING OF THE OPERATORS

In this section, for each of the operators introduced in Section 4, we identify the topological properties which are necessary and sufficient conditions for their truth. Subsequently, we use these topological properties to devise the processing algorithms and we analyze their complexity. It turns out that, for most of the algorithms, we can use concepts which are well studied in Computational Geometry and Motion Planning. Thus, we first present the preliminaries in the next subsection. Then we address each particular operator (and respective algorithm) in separate subsections. Finally, we conclude this section with some observations about the applicability of the processing algorithms for a special class of spatio-temporal range queries.

For clarity of presentation, we slightly change the order of the algorithms which implement the operators with respect to the order of their definitions, as given in Section 4, which was based on their "declarative similarity." Their order in this section reflects the similarity of the ideas used in their processing.

5.1 Preliminaries

We now present two important and well-known problems from computational geometry—*the red-blue intersection* [Basch et al. 2003; Bentley and Ottmann 1979; Chazelle and Edelsbrunner 1992; Palazzi and Snoeyink 1994] and the *Minkowski sum* [Agarwal et al. 2002; O'Rourke 2000], and we discuss how their instances apply to our particular problem domain. Subsequently, we introduce the concept of the *Minkowski difference*, which will be used in two of the algorithms for processing the operators.

5.1.1 *Red-Blue Intersection Problem.* The *red-blue intersection* problem is a variant of the problem to detect and report the set of all intersections among a given set of straight line segments, one of the oldest studied problems in Computational Geometry (cf. O'Rourke [2000]). In the red-blue intersection setting, the set of line segments is divided into two disjoint subsets *red* and *blue*, and one is interested only in intersections among line segments of different colors (i.e., the *"purple"* points [Basch et al. 2003]). One of the features that is common for all line segment intersection problems is that the complexity of the algorithms for solving them is *output-sensitive*, that is, besides the size of the input, it also depends on the number of the intersections. There are several known variations of the red-blue intersection problem, based on the properties of the segments in each of the input sets (e.g., allowing self-intersections among the segments of same color or not) and based on whether *counting* or actual *reporting* of the purple intersections is required. Each variation yields different complexity bounds on the processing algorithms [Agarwal 1990a, 1990b; Basch et al. 2003; Mairson and Stolfi 1988; Chazelle and Edelsbrunner 1992; Palazzi and Snoeyink 1994].

In our settings, one can clearly separate the segments of the (route of a) given trajectory, say, blue ones, from the edges of the query polygon R, say, red ones, and apply appropriately modified versions of the existing results when needed.

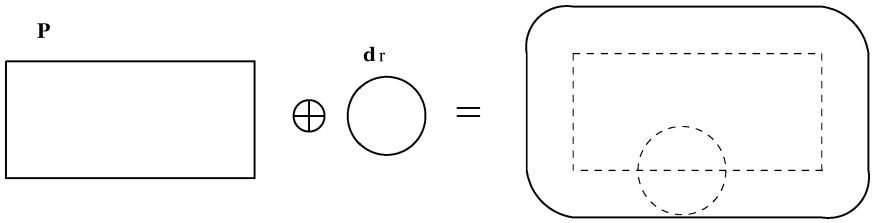


Fig. 9. Minkowski sum of a convex polygon with a disk.

5.1.2 *Minkowski Sum.* Given two sets in $\mathcal{R}^2$, say P_1 and P_2, their *Minkowski sum*, denoted by $P_1 \oplus P_2$, is defined as $P_1 \oplus P_2 = \{p_1 + p_2 \mid p_1 \in P_1, p_2 \in P_2\}$, where the summation is of vector p_1 with vector p_2 [Agarwal et al. 2002].

In our setting P_1 corresponds to the query region R and P_2 corresponds to the uncertainty disk with center at the origin and radius r, denote it d_r, which is illustrated in Figure 9.

—If R is a convex polygon having k vertices, then we can construct the Minkowski sum $R \oplus d_r$ in linear time $O(k)$ by translating/shifting each edge of R toward R's exterior, in a direction perpendicular to their original positions, and connecting the segments by circular arcs with radius r, centered at the vertices of R. Clearly, the border of the Minkowski sum will consist of an alternating sequence of k straight line segments and k circular arcs.

—If R is a simple, nonconvex polygon (subsequently also called a *concave polygon*), the Minkowski sum $R \oplus d_r$ is not necessarily simply connected any more. As the example in Figure 10 shows, extending the border of a concave polygon might lead to self-intersections of the border, which, in turn, might create holes in the extended polygon.

Before we discuss the construction of Minkowski sum for the case when R is concave, we reiterate another well-known problem of computational geometry—the *triangulation* (cf. O'Rourke [2000]). This is a special instance of the problem of *convex partitioning*, where the goal is to partition a given concave polygon into convex polygons. Quite a few algorithms have been proposed and, for the most part, the tradeoffs are between the upper-bound on the complexity versus the ease of implementation (e.g., the data structures needed) versus the minimization of the size of the output. Triangulation, on the other hand, attempts to partition a given polygon into triangles. If the polygon is convex, an obvious $O(k)$ approach is to draw the $k-3$ diagonals from a given vertex. However, this is not feasible for concave polygons. Chazelle [1991] developed an algorithm which achieves triangulation in $O(k)$ time, but for our purposes we can use a simple algorithm which runs in $O(k \log k)$ time (cf. O'Rourke [2000]). Essentially, the algorithm sorts the vertices of the polygon according to one of the axes, which takes $O(k \log k)$ time. Then, using the sorted vertices, we draw lines perpendicular to the sorting axis to achieve *trapezoidalization* of the polygon. Trapezoidalization is used to identify the monotone portions of the polygon, each of which can be triangulated in linear time, with a total of $O(k)$ triangles.

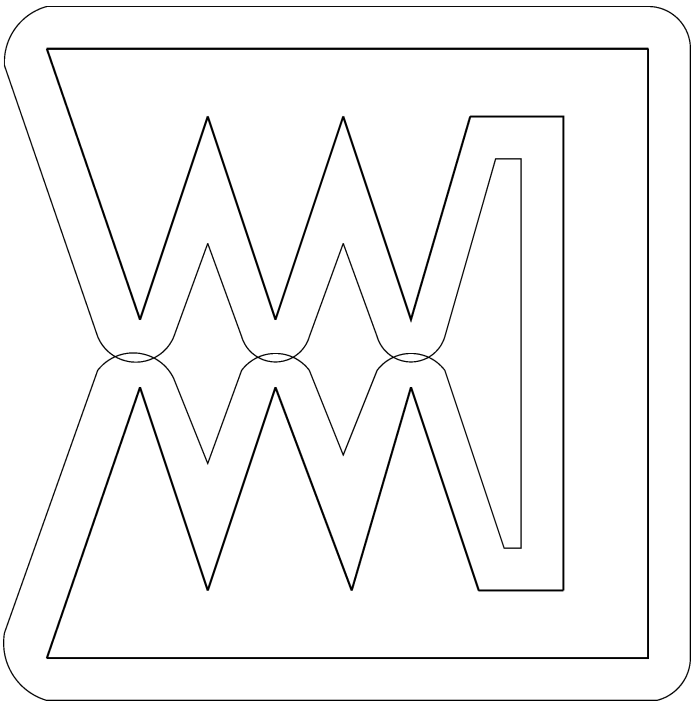


Fig. 10. Minkowski sum of a concave polygon with a disk.

Now, one can compute the Minkowski sum for each of the triangles with d_r in a straightforward manner and their boundaries are simple Jordan curves, which have the nice property that two of them intersect each other in at most two points. Consequently, the number Int_p^i of possible intersections of the boundary curves in the union of the Minkowski sum of each triangle with d_r, which exactly corresponds to the boundary of $R \oplus d_r$, is $Int_p^i \leq max(2, 6k - 24)$. Using the results in Kedem et al. [1986], one can compute the contour (boundary) of the union of the Minkowski sum of each triangle with d_r in $O(k \log^2 k)$.

Thus, in case R is bounded by a concave polygon, the Minkowski sum $R \oplus d_r$ can be calculated in $O(k \log^2 k)$.

5.1.3 *Minkowski Difference.* Now we introduce the "complementary operation" of the Minkowski sum, which we call the *Minkowski difference* of a polygon R and a disk d_r with radius r, and denote it by $R \ominus d_r$.

The set $R' = R \ominus d_r$ consists of the portion of the interior of R obtained after removing from R the region "swept" by the disk d_r as its center traverses along the edges of R. One can describe the process of constructing the Minkowski difference as moving each edge e_i of R with the same speed in a direction perpendicular to itself toward R's interior, and stopping the motion when it is at distance r from its initial position. The adjacent edges which share a reflex vertex will be connected with a circular arc with radius r, centered at their common vertex.

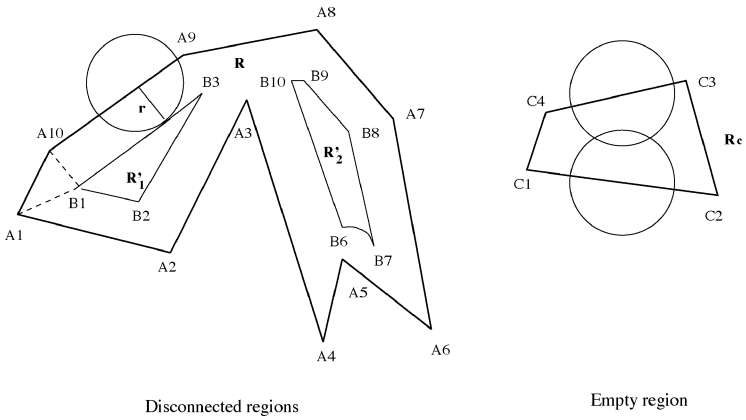


Fig. 11. Minkowski difference.

Figure 11 illustrates the process of constructing the Minkowski difference and the deformations that can occur during the process, depending on the polygon's structure and the magnitude of r:

(1) *Empty region*. As illustrated by the right portion of Figure 11, in an extreme case, for the polygon R_c the Minkowski difference may result in an empty region.

(2) *Split-event*. As illustrated by the left portion of Figure 11, the Minkowski difference of a region and a disk may result in two (or more) disjoint regions. This occurs when a reflex vertex, like A_3 in Figure 11, "splits" an edge, like $\overline{A_8A_9}$, during the motion toward R's interior.

(3) *Edge-event*. Again, as illustrated by the left portion of Figure 11, some edges may "vanish" during the process. This is the case with the edge $\overline{A_{10}A_1}$. The reason is that the radius r of the disk d_r is larger than the perpendicular distance between the edge $\overline{A_{10}A_1}$ and the point of intersection of the bisectors of the angles at A_{10} and A_1 (point B_1 in Figure 11).

The boundary of the Minkowski difference $R \ominus d_r$ of a nonconvex simple polygon consists of a sequence of line segments and circular arcs. As the example in Figure 12 shows, the boundary points between these segments and arcs (illustrated with thiner solid line) lie on the edges of the Voronoi-diagram computed for the vertices and open edges of the given polygon. This Voronoi-diagram is a superset of the well-known *medial axis* of the polygon, as illustrated with the dashed segments in Figure 12. The edges of the Voronoi-diagram are formed by

—bisectors between pairs of reflexive vertices of the given polygon,

—bisectors between pairs of edges of the given polygon,

—parabola segments determined by a reflexive vertex and an edge of the given polygon.

The Voronoi-diagram and the medial axis of a simple (not necessarily convex) polygon with k vertices can be computed in $O(k)$ time, as has been shown by Chin et al. [1999]. From the Voronoi diagram, one can extract in $O(k)$ time the

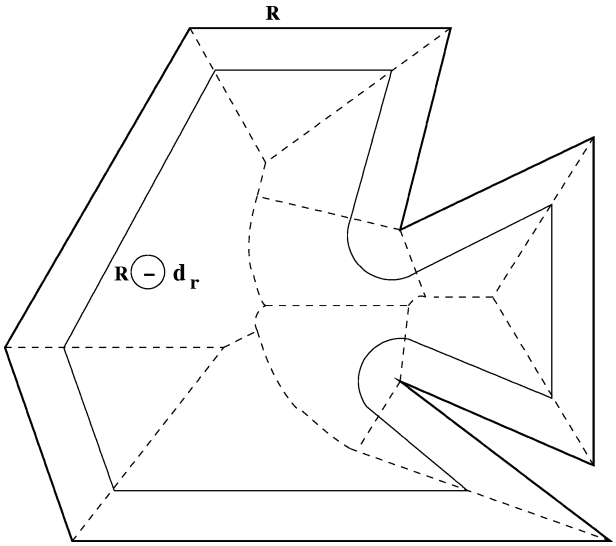


Fig. 12. Medial axis and the construction of the Minkowski difference.

Minkowski difference $R \ominus d_r$ and also determine whether it is empty. Let us point out that the algorithm presented in Chin et al. [1999] is interesting from a theoretical point of view and, as the authors themselves mentioned, is not practical. A more practical algorithm would require $O(k \log k)$ time.

If R is convex, the Minkowski difference can be constructed in a straightforward way in linear time.

5.2 Algorithms

Let t_b and t_e be two time values denoting the begin and the end time-points of the query. Taking time as a third dimension, the region R along with the query time-interval $[t_b, t_e]$ can be represented as a prism P_R in 3D space: $P_R = \{(x, y, t) \mid (x, y) \in R \wedge t_b \leq t \leq t_e\}$. P_R is called a *query-prism*. We will use VTr to denote the trajectory volume of a given uncertain trajectory $UTr = (T, r)$ between t_b and t_e and VT_{int} to denote the 3D intersection of the trajectory volume with the query prism—$VT_{int} = VTr \cap P_R$. Let T_{XY} denote the projection of the trajectory T between t_b and t_e on the XY plane (i.e., its route). Assume that the trajectory (respectively its route) has n segments[4] in the interval $[t_b, t_e]$ and that the query polygon R has k edges and is represented as a sequence of its vertices in a counter-clockwise order.

For the purpose of query processing, we assume that a 3D indexing scheme is provided by the underlying ORDBMS, similar to the ones proposed in Pfoser et al. [1999], Tayeb et al. [1998], and Vazirgiannis et al. [1998]. The insertion of a trajectory is done by enclosing, for each trajectory, the respective trajectory

[4]Observe that this is a slight abuse of terminology since, as specified in Section 2, when constructing a trajectory one may actually obtain more segments than its corresponding route.

Table I. List of Symbols Used

T	Trajectory between t_b and t_e
T_{XY}	Route of the trajectory T
r	Uncertainty radius
$UTr(T, r)$	Uncertain trajectory of T
VTr	Trajectory volume of $UTr(T, r)$
UZ	Uncertainty zone of $UTr(T, r)$
R	Query region (simple polygon)
P_R	Query prism (based in R)
VT_{int}	Intersection: $VTr \cap P_R$
VT_{sd}	Set-difference: $VTr \setminus P_R$
d_r	Disk with center in the origin and radius r
$R \oplus d_r$	Minkowski sum
$R \ominus d_r$	Minkowski difference

volume between t_i and t_{i+1} in a minimum bounding box (MBB). During the *filtering* stage we retrieve the trajectories which have at least one of their MBBs intersecting with P_R. Throughout the rest of this work we focus on the *refinement* stage of the processing. For convenience, we summarize the terms that we use and their definitions in Table I.

Now we proceed with algorithms for each of the operators. As it turns out, if the query polygon R is convex, the complexities of many algorithms can be significantly improved.

5.2.1 *Possibly_Sometime_Inside.* As a consequence of the definitions of the predicates (cf. Section 4) and VT_{int} introduced above, we have the following:

CLAIM 5.1. *The predicate Possibly_Sometime_Inside is* true *iff* $VT_{int} \neq \emptyset$

Based on Claim 5.1 the algorithm for processing the *Possibly_Sometime_Inside* predicate can be specified as follows (recall that d_r denotes a disk with radius r):

Algorithm 5.2. (Possibly_Sometime_Inside(T, R, t_b, t_e))

1. If $T_{XY} \cap (R \oplus d_r) = \emptyset$
2. return **false**;
3. else
4. return **true**;

In other words, VT_{int} is nonempty if and only if T_{XY} intersects $R \oplus d_r$. Step 1 can be verified as follows. If some arbitrary endpoint (x_i, y_i) of the route is inside $R \oplus d_r$, then clearly the intersection is not empty. Otherwise, one can simply check whether any line segment of the route intersects a side of $R \oplus d_r$. If so, the intersection is not empty.

For a concave polygon R having k vertices, the Minkowski sum $R \oplus d_r$ can be computed in $O(k \log^2 k)$ time, and the endpoints as well as the line segments of the route can be checked in $O(n \cdot k)$ time, resulting in a total time complexity of $O(n \cdot k + k \log^2 k)$.

If one can guarantee that the segments of the route do not intersect in interior points, this time complexity can be improved to $O((n+k)\log(n+k) + k \log^2 k) =$

$O(n \log n + k \log^2 k)$, by using a plane-sweep algorithm.[5] Nonintersection among the blue segments can be guaranteed using the value of the *Level* attribute. If all the segments have the same value for the *Level* attribute (as is the case, e.g., if none is an overpass), then they are guaranteed not to intersect in interior points.

In the case where R is a convex polygon, its Minkowski sum $R \oplus d_r$ does not have to be computed beforehand, but parts of it can be constructed on the fly, as needed. Then an endpoint or a line segment of the route can be checked in $O(\log k)$ time, resulting in an $O(n \log k)$ total time complexity.

5.2.2 *Always_Possibly_Inside.* The satisfaction of the *Always_Possibly_Inside* predicate can be determined based on the following:

CLAIM 5.3. *Always_Possibly_Inside(T, R, t_b, t_e) is true if and only if $VT_{int} \neq \emptyset$ for every $t \in [t_b, t_e]$.*

The proof of Claim 5.3 is, again, a straightforward consequence of the definition of the predicate.

Following is the algorithm for evaluating the predicate *Always_Possibly_Inside*:

Algorithm 5.4. (Always_Possibly_Inside(T, R, t_b, t_e))
1. If T_{XY} lies completely inside $R \oplus d_r$
2. return **true**;
3. else
4. return **false**;

Checking if the route is entirely within $R \oplus d_r$ between t_b and t_e amounts to checking whether

(1) there are no intersections between the segments of T_{XY} and $R \oplus d_r$, and
(2) an arbitrary endpoint of T_{XY} segments is in $R \oplus d_r$.

Again, the brute-force approach would yield a complexity of $O(n \cdot k + k \log^2 k)$, and if the segments of the route are guaranteed not to intersect in interior points, this time complexity again can be improved to $O((n + k) \log (n + k) + k \log^2 k) = O(n \log n + k \log^2 k)$. If the polygon R is convex, all we need to verify is that each end-point of T_{XY} is inside $R \oplus d_r$ and, due to convexity, it follows that each segment is guaranteed to be entirely within $R \oplus d_r$. An observation similar to the one used in the analysis of the predicate *Possibly_Sometime_Inside* is applicable—detecting whether a point is inside $R \oplus d_r$ can done in $O(\log k)$, and $R \oplus d_r$ can partially be constructed on the fly, as needed. Thus, the complexity of the algorithm for processing the *Always_Possibly_Inside* predicate is $O(n \log k)$ when R is convex.

[5]One may observe that the algorithms in Agarwal [1990a, 1990b], Basch et al. [2003], Chazelle and Edelsbrunner [1992], and Palazzi and Snoeyink [1994] assume that each of the inputs (red and blue) consists of straight line segments only, and in our case the red set consists of both line segments and circular arcs. The crucial observation is that there are no intersections among the same color segments. In such settings, the worst-case complexities can be retained [Basch 2004].

5.2.3 *Definitely_Always_Inside.* Recall that $VT_{int} = VTr \cap P_R$. Based on the definition of the predicate, we have the following:

CLAIM 5.5. *The predicate Definitely_Always_Inside(T,R,t_b,t_e) is true if and only if $VT_{int} = VTr$, that is, $VTr \subseteq P_R$.*

As for the implementation of the predicate, we have the following:

Algorithm 5.6. Definitely_Always_Inside(T, R, t_b, t_e)
1. For each line segment of T_{XY}
2. If the uncertainty zone of the segment is *not* entirely contained in R;
3. return **false** and exit;
4. return **true**.

Step 2 above can be processed by checking if each segment of the route is entirely contained within $R' = R \ominus d_r$. As we explained in Section 5.1.3, the time complexity of the construction of Minkowski difference $R' = R \ominus d_r$ is $O(k)$ (recall that R' may be empty). Thus, the time complexity of this algorithm is $O(n \cdot k)$.

In case the polygon R is convex, $R \ominus d_r$ is a convex polygon too, if it exists. Hence, it suffices to check if each of the endpoints of the segments of T_{XY} is inside a convex polygon with $O(k)$ edges. Thus, we get that the time complexity is $O(n \log k + k)$.

5.2.4 *Sometime_Definitely_Inside.* The truth of this predicate can be verified by using the following:

CLAIM 5.7. *Sometime_Definitely_Inside(T, R, t_b, t_e) is true if and only if VT_{int} contains an entire horizontal disk with radius r, centered at some point $(x, y, t) \in T$ ($t \in [t_b, t_e]$).*

The proof of Claim 5.7 is a straightforward consequence of the definition of the predicate and Theorem 1. An equivalent condition is that P_R contains an entire horizontal disk with radius r, centered at some point $(x, y, t) \in T$ ($t \in [t_b, t_e]$).

The implementation of the predicate *Sometime_Definitely_Inside* is specified by the following:

Algorithm 5.8. Sometime_Definitely_Inside(T, R, t_1, t_2)
1. For each straight line segment of T_{XY}
2. If R contains a circle with radius r centered at some point on the segment;
3. return **true** and exit
4. return **false**

Execution of step 2 of the Algorithm 5.8 again requires construction of the Minkowski difference $R' = R \ominus d_r$. If R' is "empty," we know that the output of Algorithm 5.8 is false. Otherwise, we need to check for the existence of an intersection between R' and T_{XY}. Checking every segment of T_{XY} against every segment of R' requires $O(nk)$. If no intersection is found, it may be the case that the entire T_{XY} is contained within R', for which it suffices to check if at

least one endpoint of an arbitrary segment of T_{XY} is inside R'. Thus, the time complexity is $O(n \cdot k)$.

In case R is convex, the complexity of Algorithm 5.8 becomes $O(n \log k + k)$.

Again, in a manner similar to Algorithm 1 and Algorithm 2, we observe that the respective complexities of Algorithms 3 and 4 can be improved even when R is concave using the *Level* attribute. Specifically, if all the route segments have the same value for the *Level* attribute, then the complexities of Algorithms 3 and 4 become $O((n + k) \log (n + k))$.

5.2.5 *Definitely_Sometime_Inside.* Let VT_{sd} denote the set-difference of VTr and P_R. Based on the definition of the predicate, we have the following:

CLAIM 5.9. *Definitely_Sometime_Inside*(T, R, t_b, t_e) *is true if and only if* $VT_{sd} = VTr \backslash P_R$ *does not contain a possible motion curve between* t_b *and* t_e.

Before we proceed with presenting the algorithm for processing the predicate *Definitely_Sometime_Inside*, we review some topological facts.

Connectivity of a topological set is commonly viewed as an existence of some *path* between every pair of points in the set, using the following definition (c.f. [Sutherland 1978]):

Definition 5.10. Given any two points a and b in a topological space S, a *path* from a to b in S is any continuous *map* $f : [0, 1] \rightarrow S$ such that $f(0) = a$ and $f(1) = b$.

In order to apply these observations in our settings and toward derivation of the algorithm for processing the *Definitely_Sometime_Inside* predicate, we make use of the following theorem:

THEOREM 5.11. $VT_{sd} = VTr \backslash P_R$ *is path-connected between* t_b *and* t_e*, if and only if there exists a possible motion curve* PMC^T *between* t_b *and* t_e *which is entirely in* VT_{sd}.

PROOF. (See Appendix.) □

The important consequence of Theorem 5.11 is that in order to ensure that the predicate *Definitely_Sometime_Inside* is *true*, it suffices to demonstrate that VT_{sd} is not path-connected. This is what we use in the algorithm which processes the *Definitely_Sometime_Inside* predicate.

Let UZ denote the uncertainty zone (the XY projection of VTr) between t_b and t_e. Let C_b and C_e denote the circles which form the boundaries of the uncertainty areas at t_b and t_e, respectively. Let UZ_{in} denote UZ with the interior of C_b and C_e removed, and the outer half-circles of C_b and C_e also removed. Let C'_b and C'_e denote the inner half-circles of C_b and C_e. Also, let l_1 and l_2 denote the left and right boundaries of UZ_{in}, with respect to the direction of the object's motion along the route. Clearly, each of l_1 and l_2 will consist of a "polyline-like" sequence of line segments and circular arcs. The concepts are illustrated in Figure 13.

An important observation is that VT_{sd} is path-connected between t_b and t_e if and only if there exists a path in UZ connecting a point of C'_b with a point of C'_e which only consists of parts of the boundary of the query polygon R lying in

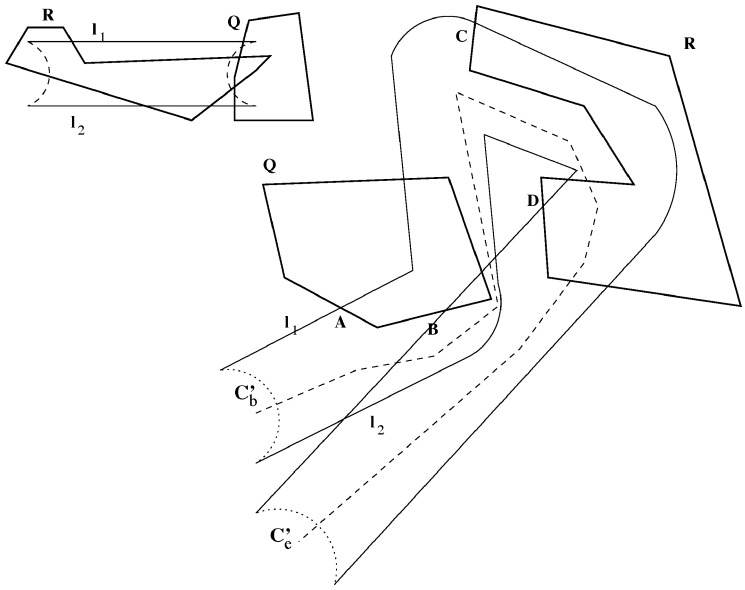


Fig. 13. Processing of the *Definitely_Sometime_Inside* predicate.

the interior of UZ and/or parts of either l_1 or l_2 which are not contained in the interior of R. Such a path cannot contain parts of both l_1 and l_2! This fact is used in the following algorithm for processing the *Definitely_Sometime_Inside* predicate.

Algorithm 5.12. Definitely_Sometime_Inside(T, R, t_b, t_e)

1. If l_1 or l_2 do not intersect the interior of R
   return **false** and exit
2. Else_If
   there exists a path in UZ between C'_b and C'_e which consists of either
   2.1 part of the boundary of R contained in the interior of UZ
   or
   2.2 an alternating sequence of parts of the boundary of R contained in the interior of UZ and parts of l_1 not contained in the interior of R
   or
   2.3 an alternating sequence of parts of the boundary of R contained in the interior of UZ and parts of l_2 not contained in the interior of R
   return **false** end exit
3. return **true**

The correctness of Algorithm 5.12 follows from Theorem 5.11 and the above mentioned observation. The condition in line 2.1 covers the case when both l_1 and l_2 are contained in R, but there is a "channel" between them in $UZ \setminus R$ which connects C'_b and C'_e. This implies an existence of a possible route completely inside $UZ \setminus R$ which, in turn, implies an existence of a possible motion curve inside $VT_{sd} = VTr \setminus P_R$. If l_1 (respectively l_2) is contained in R, but not l_2 (respectively l_1), and there is a path in $UZ \setminus R$ which connects C'_b and C'_e, then this case is covered by the condition in line 2.3 (respectively 2.2).

Figure 13 gives an illustration of the conditions stated in Algorithm 5.12. Observe that in both examples, the regions marked as Q do not satisfy the *Definitely_Sometime_Inside* predicate because there exists a path consisting of parts of l_1 and parts of the boundary of Q connecting C'_b and C'_e. The dashed polyline in the right portion of Figure 13 illustrates a possible route which invalidates the conditions for the *Definitely_Sometime_Inside* predicate. However, in both examples, the regions denoted by R satisfy the *Definitely_Sometime_Inside* predicate.

Now we turn to the complexity analysis of the Algorithm 5.12. As usual, let n denote the number of trajectory segments and k the number of vertices/edges of R. Let q denote the total number of intersections between the boundaries of UZ and R. The main difference from the previously analyzed algorithms is that now we need to detect the actual locations of the (purple) intersection points, instead of simply checking if some exist. Since l_1 and l_2 can contain line segments and circular arcs that intersect themselves, the complexity of detecting the intersections of UZ with R now becomes $O((n+k+q)2^{\alpha(n+k+q)}\log^3(n+k))$, where $\alpha()$ denotes the very slowly growing inverse of the Ackermann-function (cf. Theorem 4.1 in Basch et al. [2003] and Sharir and Agarwal [1995]).

The detected intersection points can be inserted as additional vertices into l_1, l_2 and R at no extra cost. However, after detecting the intersection points between the boundaries of UZ and R, some extra verification needs to be done.

If l_1 (respectively l_2) does not contain any intersection point, one can check in $O(k)$ time whether one of the endpoints of l_1 (respectively l_2) is outside R. If this is the case, l_1 (respectively l_2) does not intersect the interior of R, and the condition of line 1 of Algorithm 5.12 is satisfied.

If C'_b (respectively C'_e) does not contain any intersection point, one can check in $O(k)$ time whether one of the endpoints of C'_b (respectively C'_e) is inside R. If this is the case, C'_b (respectively C'_e) is contained in R and therefore no path which satisfies one of the conditions 2.1–2.3 can exist, that is, the predicate is true.

If R intersects both l_1 and l_2, and C'_b and C'_e both are not contained in R, we try to incrementally construct a path connecting C'_b with C'_e which satisfies one of the conditions 2.1–2.3. We check initially whether the common endpoint of C'_b and l_1 (respectively l_2) is outside R. If so, the path starts at this endpoint and follows l_1 (respectively l_2) until the next intersection point with the boundary of R. If both endpoints of C'_b, one common with l_1 and the other common with l_2, lie within R, there must exist at least one intersection of C'_B with the boundary of R. In this case, the path starts at such an intersection point, and we look ahead along l_1 and l_2 for the next intersection points with the boundary of R. If no such intersection points exist, we follow the boundary of R inside UZ and try to reach C'_e, that is, we check whether condition 2.1 is satisfied. If we reach C'_e, we have found a path and the predicate returns *false*, otherwise it returns *true*. If we find one (or two) intersection points when looking ahead along l_1 and l_2, we still follow the boundary of R inside UZ, until we reach one of the intersection points, say on l_1. Then the path continues along l_1 (and outside R) until the next intersection point of l_1 with the boundary of R.

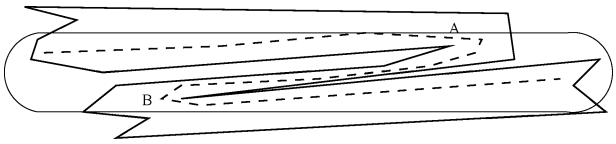


Fig. 14. Continuous curve which is not a possible motion curve.

When walking along l_1 (respectively l_2) and reaching an intersection point with the boundary of R in segment i, we look ahead along l_1 (respectively l_2) for the next intersection with the boundary of R. If such an intersection does not exist, we continue walking along the part of the boundary of R inside UZ and try to reach C'_e. If before reaching C'_e we encounter an intersection point with l_2 (respectively l_1) which belongs to a segment $s \geq i$ and at which we would leave the uncertainty segment s, we know that no path can exist which satisfies one of the conditions 2.1–2.3, that is, the predicate is true. If we reach C'_e, we have found a path and the predicate returns false. If the look ahead along l_1 (respectively l_2) returns an intersection point I_j for the jth segment of the trajectory's route, we continue walking along the part of the boundary of R inside UZ and try to reach I_j. If before reaching I_j we encounter another intersection point with l_2 (respectively l_1) which belongs to a segment s, $i \leq s \leq j$, and at which we would leave the uncertainty segment s, we know that no path can exist which satisfies one of the conditions 2.1–2.3, that is, the predicate is true. If we reach I_j, our walk continues on l_1 (respectively l_2) until the next intersection point with the boundary of R.

In the worst case, we have to spend for each of the n segments of the trajectory's route an $O(k+q)$ time to traverse the boundary of R once. Hence, the cost of the construction of the path can cost $O(n \cdot (k+q))$.

Based on the above discussion, the time complexity of Algorithm 5.12 is $O(n \cdot (k+q) + (n+k+q)2^{\alpha(n+k+q)} \log^3(n+k))$.

5.2.6 *Possibly_Always_Inside.* Observe that, due to Theorem 4.1, in the case where the query region R is bounded by a convex polygon, we can readily apply Algorithm 5.4 to process the *Possibly_Always_Inside* predicate. However, it turns out that the ease of processing the convex case has a "multiple penalty" for the case when R is concave.

The main problem which occurs when R is concave is that we may have continuous 3D curves (i.e., paths) completely inside $VT_{int} = VTr \cap P_R$ all throughout the query time $[t_b, t_e]$, which are *not* possible motion curves. This subtlety is due to the definition of a *path* as a *map*, whereas possible motion curves are defined as *functions*.

An illustration in the XY plane is given by Figure 14. The query polygon is indicated with the thicker, and the uncertainty zone is shown with the thiner solid line. The dashed curve indicates a possible route. However, a careful observation will reveal that any moving object which "obeys" the restriction of its uncertainty area cannot follow the route indicated by the dashed curve. The reason is that, even with the tolerance of the uncertainty area, the latest time

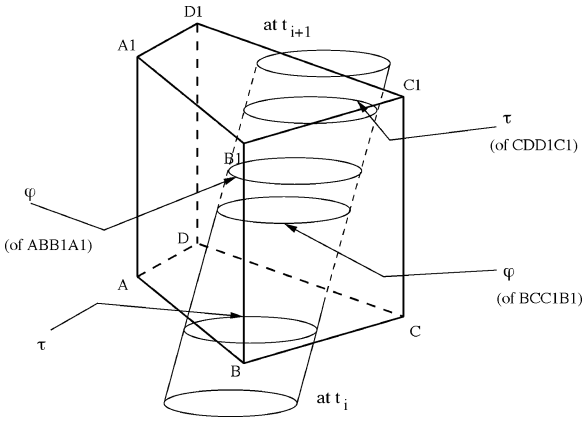


Fig. 15. Critical times of the sides of prism and a trajectory volume.

at which the object can be at point B is much earlier than the earliest time at which the object can reach point A.

In 3D terms, this means that the volume $VT_{int} = VTr \cap P_R$ is connected by a 3D curve which has two (or more) points (x_i, y_i, t) and (x_j, y_j, t) which, as mentioned, violates the very definition of a possible motion curve as a *function* $f_{PMC^T} : Time \rightarrow R^2$. Although there may be a *path* from the start to the destination point, the object may need to travel "back in time" to follow that path.

To develop a criterion which we will use for evaluating the *Possibly_Always_Inside* predicate, we need to introduce some notation. Let v_i denote the ith vertex of the region R, and let e_i denote the edge between v_i and v_{i+1} (modulo k). Also, let s_i denote the ith side of the query prism P_R. The *critical times* of the segment of a trajectory volume between t_j and t_{j+1} with respect to the side s_i of the prism P_R, based in a polygon R at $t = t_j$ and with height $(t_{j+1} - t_j)$, is defined as follows:

Definition 5.13. τ_{ij}—*begin_intersection_time* is the time value in $[t_j, t_{j+1}]$ such that for every $t \in [t_j, \tau_{ij})$ the uncertainty area of the trajectory at t and s_i have no intersection points and at τ_{ij} they have exactly one intersection point. If the uncertainty area at t_j intersects s_i, then $\tau_{ij} = t_j$. If the uncertainty area does not intersect s_i for any $t \in [t_j, t_{j+1}]$, we set τ_{ij} to some large positive value, say $\tau_{ij} = \infty$.

φ_{ij}—*end_intersection_time* is the time value in $[t_j, t_{j+1}]$ such that for every $t \in (\varphi_{ij}, t_{j+1}]$ the uncertainty area of the trajectory at t and s_i have no intersection points and at φ_{ij} they have exactly one intersection point. If the uncertainty area at t_{j+1} intersects s_i, then $\varphi_{ij} = t_{j+1}$. If the uncertainty area does not intersect s_i for any $t \in [t_j, t_{j+1}]$, we set φ_{ij} to some large negative value, say $\varphi_{ij} = -\infty$.

We will use τ_j to denote the minimal τ_{ij} and φ_j to denote the maximal φ_{ij}. Thus, τ_j and φ_j correspond to the "global" begin_intersection and end_intersection times of the prism with respect to the jth segment of the trajectory volume. The concepts introduced in Definition 5.13 are illustrated in Figure 15. Observe that the τ-times are the same for the sides ABB_1A_1 and BCC_1B_1 of the

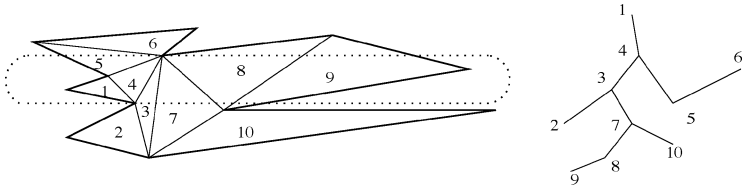


Fig. 16. Triangulation of the concave polygon and its dual graph.

prism, and this is the global τ value for the prism. However, they have different φ-times. On the other hand, the side DAA_1D_1 has $\tau = \infty$ and $\varphi = -\infty$. Meanwhile, the side CDD_1C_1 has its definite τ value, but $\varphi = t_{j+1}$, and that is the global φ for the entire prism. Observe that, as a consequence of Definition 5.13, a particular side of the prism will have $\varphi_{ij} = -\infty$ if and only if $\tau_{ij} = \infty$. Also, let us point out that, as a special case, we may have $\tau_{ij} = t_j$ and $\varphi_{ij} = t_{j+1}$ when the side s_i is tangent to the segment of *VTr* between t_j and t_{j+1}

The calculations used in obtaining the critical times are presented in the Appendix.

Our algorithm for processing the *Possibly_Always_Inside* predicate first triangulates the query polygon R into $O(k)$ triangles. As we have seen in Section 5.1.2, triangulation of a simple polygon can be achieved in $O(k)$ time [Chazelle 1991], but a more practical approach would use $O(k \log k)$ (c.f. O'Rourke [2000]).

The concepts are illustrated in Figure 16, where the thick solid lines indicate the polygon and the thin solid lines depict the diagonals used for the triangles. Since each triangle has three edges, it follows that each node in the dual graph will have at most three edges incident to it. Clearly, the vertical prism based in triangle 10 will have $\tau = \infty$ and $\varphi = -\infty$ for all three of its sides.

Observe that for each triangle, we can always label its edges in such a manner that the edge which is shared among two adjacent triangles has the same label in both triangles.

Now we can specify the algorithm for processing the *Possibly_Always_Inside* predicate when the region R is a concave polygon.

Algorithm 5.14. – Possibly_Always_Inside(T, R, t_b, t_e)

1. Triangulate the polygon R and construct its dual graph, denote it D_R.
2. Label each edge of D_R with the label of the side that is common to the triangles in R corresponding to edge's endpoints in D_R.
3. Let S_1 denote the set of nodes of D_R whose corresponding triangles intersect the uncertainty area at t_b.
4. For each segment j of the trajectory, between t_b and t_e
5. For each triangle Δ_i from R
6. Calculate the critical times for each side of the prism based in Δ_i
7. Label the vertices of the dual graph with the triplet $((\tau_{l_1 j}, \varphi_{l_1 j}), (\tau_{l_2 j}, \varphi_{l_2 j}), (\tau_{l_3 j}, \varphi_{l_3 j}))$ denoting the critical times of each side of the prism based in an edge of the triangle, where l_1, l_2 and l_3 are the labels of each edge of the corresponding triangle in R.
8. end_for
9. If at t_j (the beginning time of the segment) the uncertainty area is outside the polygon

10. return **false** and exit.
11. Let S_{j+1} be the set of all the nodes that can be reached
by a path in the graph such that :
11.1 it *starts* at a node in S_j and *ends* at a node with $\varphi_j \geq t_{j+1}$ AND
11.2 each next node along the path has $\tau_j \leq \varphi_j$ of the previous node AND
11.3 for each edge along the path with label l_k, its end-points (nodes in the graph) have $\tau_{l_k j} \neq \infty$ (equivalently, $\varphi_{l_k j} \neq -\infty$)
12. If S_{j+1} is empty, return **false** and exit.
13. end_for
14. return **true**.

The correctness of Algorithm 5.14 can be demonstrated as follows. If the condition in line 9 is satisfied, then we cannot have a possible motion curve which is inside P_R at the beginning of the jth segment of the trajectory volume. Hence, the output is *false*. Otherwise, we can have (possibly) uncountably many of them which are inside P_R at the t_j. The condition in line 11.1 simply states that the motion in the jth uncertainty segment has to continue at t_j at the same triangle where it has ended in the previous uncertainty segment and its motion has to finish in the uncertainty area of t_{j+1}. If this cannot be achieved, then the object cannot have a possible motion curve throughout the entire $[t_j, t_{j+1}]$. The condition in the second conjunct (line 11.2) states that the transition should occur between prisms which have adjacent sides and without having a need to "travel back in time." In other words, one can construct a curve which intersects the sequence of prisms, each consecutive pair of which has adjacent side, in a manner which is monotonically increasing in time. The last conjunct, line 11.3 states that, if one needs to be able to "travel" from one triangular prism within P_R to an adjacent one, it should be in a "smooth" manner, by entering the next one via its common side with the previous one. Otherwise, we have a discontinuity in the possible motion curves which exists between the two consecutive triangular prisms and, consequently, such a path is not allowed. Thus, if the set S_{j+1} is not empty, it means that there exists a path monotonic in t which starts at t_b and enters every next triangular prism through the side which is common with the previous one along the way, until t_{j+1}.

As for the complexity of Algorithm 5.14, we observe that line 11 is the most expensive one. We may have $O(k^2)$ pairs of nodes to check for the existence of a path, the length of which is also $O(k)$, which yields $O(k^3)$. If the trajectory has n segments between t_b and t_e, this brings a bound of $O(nk^3)$. The penalty for calculating the critical times per segment of a trajectory volume is $O(k)$, which yields overall of $O(nk)$. However, this is dominated by $O(nk^3)$ which, in turn, dominates the $O(k \log k)$ spent for the triangulation. Thus, the upper bound of the running time of Algorithm 5.14 is $O(nk^3)$.

5.3 Circular Query Regions

We now present a special case when the query region R is bounded by a circle.[6] There is nothing challenging about the processing algorithms, as we

[6]A circle is a one-dimensional set of points. We use the term *disk* to denote the two-dimensional closed region bounded by a circle.

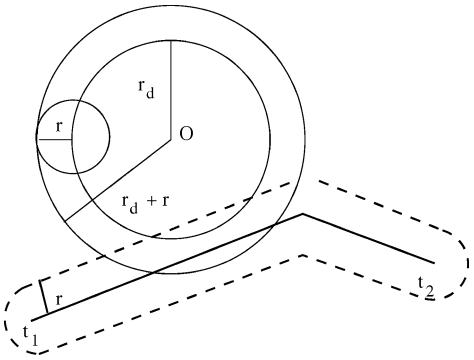


Fig. 17. *Minkowski sum* of two circles.

will demonstrate shortly. However, the reason that we address this case is that it may have (declarative) practical significance. One may now pose a variant of spatio-temporal range queries like:

"Retrieve all the objects that will be possibly/definitely WITHIN DISTANCE *of 2 miles from the hospital H,* sometime/always *between t_b and t_e."*

Observe that the answer to this query will consist of all the trajectories which will *possibly*/*definitely* intersect the disk centered at the (location of the) hospital H with radius 2 miles, *sometime*/*always* between t_b and t_e. Thus, for this syntactic variant we can apply our operators, and the only modification is that the query region will now be bounded by a circle instead of a closed polyline.

There are two main observations with respect to the processing of the operators. First, the disk *is* a convex entity and, viewed in 3D within some time-interval $[t_b, t_e]$, it yields a vertical cylinder. Second, we do not have the notion of a *side* of a polygon. This affects the complexity of the processing algorithms.

Let $D(O, r_d)$ denote the disk centered at O with the radius r_d representing the query region R. As before, we use r to denote the uncertainty radius of the trajectory. If we construct the *Minkowski sum* of $R = D(O, r_d)$ with the disk centered at the origin and having the same radius r as the uncertainty, the result is another disk $D(O, (r_d + r))$. However, this is now done in constant time.

As illustrated in Figure 17, if a trajectory satisfies *Possibly_Sometime_Inside* between the times t_b and t_e, with respect to the disk $D(O, r_d)$, its route will intersect the disk centered at O with radius $r_d + r$.

Observe that each variant of the *Within_Distance* queries with respect to a point-location can be processed by using the algorithms for the predicates that we already described. Following are the complexity results when R is a disk.

—*Possibly_Sometime_Inside(T, R, t_b, t_e)*. The algorithm still needs to verify if $T_{XY} \cap D(O, (r_d + r)) \neq \emptyset$. However, now we only consider each of the route segments once per query circle, which yields a complexity of $O(n)$. Clearly, using this predicate, one can verify if a given trajectory T satisfies *Possibly_Sometime_Within_Distance (T, O, r_d, t_b, t_e)*, where O is the center and r_d is the distance from the center.

—*Possibly_Always_Inside(T, R, t_b, t_e)*. Due to the "convexity" of the circle, the algorithm needs to verify whether each of the segments' endpoints between t_b and t_e is inside $D(O,(r_d + r))$, which can be done in $O(n)$ time. Recall that, due to Theorem 4.1, this is also sufficient for verifying the *Always_Possibly_Inside* predicate. Thus, we can use it to evaluate both *Possibly_Always_Within_Distance (T, O, r_d, t_b, t_e)*, and *Always_Possibly_Within_Distance (T, O, r_d, t_b, t_e)*.

—*Definitely_Always_Inside(T, R, t_b, t_e)*. It suffices to check if the circle with radius r (the uncertainty) centered at the endpoint of each segment of the trajectory is contained within $R = D(O, r_d)$. The complexity is, once again, $O(n)$. Obviously, we can use this to evaluate whether the trajectory T with the given uncertainty r is *Definitely_Always_Within_Distance* r_d from the circle centered at O.

—*Sometime_Definitely_Inside(T, R, t_b, t_e)*. This predicate is true if and only if $T_{XY} \cap D(O, r_d - r) \neq \emptyset$. Thus, in order to satisfy the predicate, we need to check if there exists a segment of T_{XY} which intersects $D(O, r_d - r)$, which can be done in $O(n)$ time. This can be used to process the *Sometime_Definitely_Within_Distance (T, O, r_d, t_b, t_e)* predicate.

As for the last predicate, we have the following interesting observation, as a specific property of the disk query region:

THEOREM 5.15. *If the query region is a disk, then* Sometime_Definitely_Inside *is true if and only if* Definitely_Sometime_Inside *is true.*

PROOF. Assume that the query region R is bounded by a disk $D(O, r_d)$. Observe that if $r_d < r$, the theorem is trivially true because none of the predicates is satisfied.

($\Rightarrow$) If *Sometime_Definitely_Inside* is satisfied then it follows, from the definitions of the predicates and the tautology (3) (cf. Section 4), that *Definitely_Sometime_Inside* is also satisfied.

($\Leftarrow$) Assume that *Definitely_Sometime_Inside* is satisfied. Let C_Q denote the query cylinder, based at the query disk $D(O, r_d)$ at the vertical plane $t = t_b$, and with a height $t_e - t_b$.

Assume that $VTr \cap C_Q$ does not contain a full disk with radius r for any $t \in [t_b, t_e]$. Then we can construct a possible motion curve between t_b and t_e which "walks" around the outer boundry of C_Q and is entirely within $VTr \backslash C_Q$. This, however, contradicts the assumption that *Definitely_Sometime_Inside* is satisfied. Hence, there must exist some $t \in [t_b, t_e]$ at which $VTr \cap C_Q$ contains a full disk with radius R, which is sufficient to satisfy the *Sometime_Definitely_Inside* predicate. □

One last observation is that the *Within_Distance* variant can be readily used for a generalization of the query point to a region. Our operators can be easily extended to process queries of the type:

"Retrieve all the objects which will possibly/definitely *be within distance of 1 mile from the region R,* sometime/always *between 3:00 and 3:15 PM."*

Table II. Complexity Results

Operator	Complexity of algorithm	Type of region
Possibly_Sometime_Inside	$O(n \cdot k + k \log^2 k)$	Concave
	$O(n \log k)$	Convex
	$O(n)$	Disk
Possibly_Always_Inside	$O(nk^3)$	Concave
	$O(n \log k)$	Convex
	$O(n)$	Disk
Always_Possibly_Inside	$O(n \cdot k + k \log^2 k)$	Concave
	$O(n \log k)$	Convex
	$O(n)$	Disk
Definitely_Sometime_Inside	$O(n \cdot (k+q) + (n+k+q)2^{\alpha(n+k+q)} \log^3(n+k))$	Concave
	$O(n \cdot (k+q) + (n+k+q)2^{\alpha(n+k+q)} \log^3(n+k))$	Convex
	$O(n)$	Disk
Sometime_Definitely_Inside	$O(n \cdot k)$	Concave
	$O(n \log k + k)$	Convex
	$O(n)$	Disk
Definitely_Always_Inside	$O(n \cdot k)$	Concave
	$O(n \log k + k)$	Convex
	$O(n)$	Disk

5.4 Summary of the Complexity Results

We conclude this long section with the summary of the complexity results for our operators, presented in Table II, grouped by the "nature" of the query region R. As usual, n is the number of the trajectory segments in the query time-interval, k is the number of the vertices/sides of the query region R, and q denotes the number of intersections. We reiterate that in the case where the segments of the T_{XY} do not intersect each other, the factor $O(nk)$ can be reduced to $O((n+k)\log(n+k))$.

6. SYSTEM ASPECTS

In this section we present some of the implementation issues related to the database server that are of practical interest when designing a system for modeling, tracking, and querying moving objects. Describing the software architecture and all the implementation details of such a system is beyond the scope of this work. Instead, we present some experimental observations that are relevant to the work presented in the previous sections, based on our experiences with the DOMINO project [Trajcevski et al. 2002a; Wolfson et al. 2002].

As we indicated in Section 3, one cannot expect a casual user of the system (e.g., a dispatcher in a transportation company) to be familiar with the SQL syntax and to enter queries, to be familiar with translating points into the map-based reference coordinate systems, etc.

Figure 18 illustrates the GUI part of the DOMINO project which implements our operators. It represents a visual tool which, in this particular example, shows the answer to the query: "Retrieve the trajectories which *possibly* intersect the region *sometime* between 12:15 and 12:30." The figure shows three trajectories in Cook County, Illinois, and the query region (polygon) represented by the shaded area. The region was drawn by the user on the screen when entering

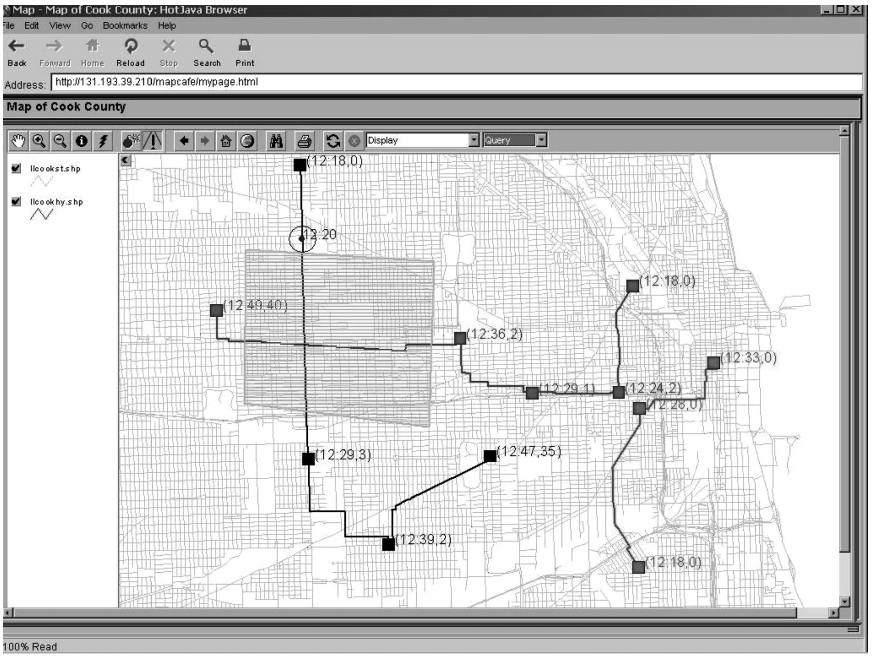
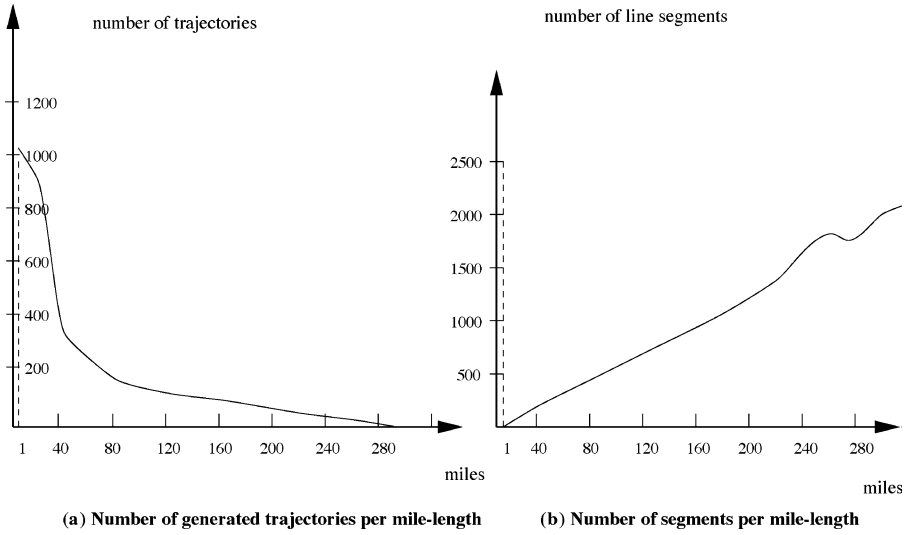


Fig. 18. Visualization of Possibly_Sometime_Inside.

the query, simply by drag-and-drop of the mouse. The query itself was selected from a pull-down menu, and the user entered the relevant time-parameters in the text boxes. Each trajectory shows the route with planned stops along it (indicated by dark squares). It also shows the expected time of arrival and the duration of the job (i.e., the stay) at each stop. Observe that only one of the trajectories satisfies the predicate *Possibly_Sometime_Inside* with respect to the polygon. It is the one with the circle labeled *12:20*, the earliest time at which the object could enter the query polygon. The other two trajectories fail to satisfy the predicate, each for a separate reason. One of them will not intersect the polygon ever (i.e., the polygon is not on the route). Although the other trajectory's route intersects the polygon, the intersection will occur at a time which is not within the query time-interval [12:15, 12:30].

As a part of the DOMINO project, we wanted to get some realistic estimates about the size of the trajectory. Following the procedure presented in Section 2, we constructed 1141 trajectories based on the electronic map of 18 counties around Chicagoland. The map size is 154.8 MB and has 497,735 records, each representing a city-block.

The trajectories were constructed by randomly choosing a pair of endpoints, and connecting them by the shortest path in the map (shortest in terms of the *Drive_Time*). Our results are depicted in Figure 19. The length of the routes was between 1 and 289 miles, and the left graph represents how many routes were within a given mileage range. The right graph represents the number of segments per trajectory, as a function of the length of the route and one can notice the linear dependency between the "storage requirements" (number



(a) Number of generated trajectories per mile-length **(b) Number of segments per mile-length**

Fig. 19. Number of segments in real-map trajectories.

of segments) and the length of a route. The average number of segments per mile turned out to be 7.2561. An important observation is that, assuming that a trajectory point (x, y, t) uses 12 bytes and that each vehicle from a given fleet (e.g., a metropolitan delivery company) drives a route of approximately 100 miles, we need $\approx$10 kbytes of storage per trajectory. Thus, the storage requirements for all the trajectories of a fleet of 1000 vehicles is $\approx$10 MB which means that the trajectories of the entire fleet can be kept in main memory.

As for the uncertainty aspect, as part of our experiments, we have conducted three actual drive-tests on the trajectories generated form the Chicagoland maps. Each of the vehicles was equipped with a GPS connected to a PDA. Initially, the team members in each vehicle entered the addresses of the start-point, stop-points, and endpoint of the trip. After transmitting them to the server, each PDA was returned the sequence of the trajectory segments. The approximate length of the trajectories was 25 miles and they were tailored to partially enter periods of both regular traffic and rush hour. The uncertainty threshold r and the update policy were stored in each PDA. Along the trip, whenever the GPS-reported point was further than r from the expected location, a location-update was generated and sent to the server. Subsequently, the server would generate the new future-portion of the trajectory and transmit it back to the PDA in the vehicles. We observed[7] that, for the value of $r = 0.2mi$, the average number of updates per mile was approximately 5. For a company like Federal Express, which monitors its fleet based on a region, there are approximately 100 vehicles in the downtown Chicago area. If the average speed is 30 miles/h then each vehicle takes about 2 min/mile, which is five updates every 2 min. For the total fleet of 100 vehicles, this means that the server receives

[7]A methodology for generating realistic pseudotrajectories and an extensive set of experiments for comparing different update policies is presented in Wolfson and Yin [2003].

Operator	Oracle Spatial		C++ (DLL)	
	$T_s + T_c$ (s)	T_c (s)	$T_s + T_c$ (s)	T_c (s)
Possibly_Sometime_Inside	0.011132	0.000063	0.176633	0.000791
Possibly_Always_Inside	0.011178	0.000109	0.182893	0.000654
Definitely_Always_Inside	0.011176	0.000107	0.181296	0.000076
Definitely_Sometime_Inside	0.011277	0.000208	0.185915	0.000336
Sometime_Definitely_Inside	0.012233	0.001166	0.193573	0.000710

Fig. 20. Experimental comparison of implementation alternatives.

approximately 4.2 updates/s which, if all the trajectories are kept in main memory, is an acceptable load.

We conclude this section with a set of experimental observations which illustrate some more tradeoffs involved and their impact on the choices that one has to make, when incorporating data management into a real application. Namely, when implementing the operators, we had a choice of using the host language for implementing the processing algorithms (cf. Section 5) and relying on the ORDBMS for the storage (and other queries) or using the advanced features that the commercially available ORDBMSs offer. We compared a C++ based implementation with the one using the PL/SQL of Oracle and its spatial extender (e.g., MDSYS.SDO_GEOMETRY).

The platform used for the performance evaluation was Win2000 Professional on Intel Pentium 4, 1.90-GHz processor, with 512 MB of RAM, Service Pack 2. We used Oracle 9.0.1 with DBMS_PROFILER Package as an ORDBMS. Since we did not use any index, the experiments consisted of simply selecting three to four trajectories and a convex query region, and executing the operators. To measure the execution time in the C++ implementation, the API high-resolution timers available for Win2000 through Kernel32.dll (available via http://msdn.microsoft.com/library/) were used.[8] The external processes in C++ used OCCI to connect to Oracle.

Figure 20 illustrates the average values of three experimental comparisons. We have separately monitored two time values: (1) $T_s + T_c$—the time it takes to complete the process of retrieving the selected trajectory and filtering the points between *begin_time* and *end_time* of the query and to complete the processing of the predicate; and (2) T_c—the actual computational time of determining if a particular operator is *true* or *false* for the given set of trajectory points and the query polygon.

As can be observed, the T_s portion is a major overkill for an implementation which uses C++ based processing. Having an index will only improve the T_s factor for the Oracle-based implementation. Even for the T_c component, the Oracle implementation performed better than the C++ one, for three (i.e., four, since *Possibly_Always_Inside* is equivalent to *Always_Possibly_Inside* for convex

[8]We did not consider the overhead of the calls to this API per se, as it was not significant and did not have an impact on the conclusions.

query regions) out of six predicates. One of the reasons for it is the time spent in constructing the instances of the necessary data types in C++ used in the algorithms. When the operators provided by the Oracle ORDBMS are used, these are subject to internal optimization by the database engine.

It seems that a very promising avenue and, at the same time, a very challenging topic, would be to actually incorporate existing code like, for example, the *CGAL* library of computational geometry algorithms (http://www.cs.ruu.nl/CGAL) as an extender to the commercially available ORDBMSs. At this time, we view it as a future goal.

7. RELATED WORK

Linguistics, modeling, and querying issues in moving objects databases have been addressed from several perspectives. Sistla et al. [1997] introduced the MOST model for representing moving objects (similar to Saltenis et al. [2000]) as a function of (location, velocity_vector). The underlying query language is nonstandard, and is based on the Future Temporal Logic (FTL). Similar issues are addressed in [Vazirgiannis et al. 1998]. A trajectory model similar to ours was given in [Vazirgiannis and Wolfson 2001], where the authors extended range queries with new operators for special cases of spatio-temporal range queries. However, there was no treatment of the uncertainty aspect of the moving object's location. A series of works [Erwig et al. 1998; Forlizzi et al. 2000; Güting et al. 2000, 2003] addressed the issue of modeling and querying moving objects by presenting a very comprehensive framework of abstract data types and a rich algebra of operators. The works were targeted toward providing a formal foundation for modeling and querying known motions in the past. Since the uncertainty is more important for future trajectories, it was not addressed in these works.

Uncertainty issues in moving object databases have been addressed before. [Wolfson et al. 1998, 1999] introduced a cost-based approach to determine the size of the uncertainty area (r in this article). However, linguistic issues and querying aspects were not addressed in these articles. A formal quantitative approach to the aspect of uncertainty in modeling moving objects was presented in Pfoser and Jensen [1999]. The authors limited the uncertainty to the past of the moving objects and the error may become very large as time approaches *now*. It was a less "collaborative" approach than ours in the sense that there is no clear notion of the motion *plan* is given by the trajectory. Uncertainty of moving objects was also treated in Sistla et al. [1998] in the framework of modal temporal logic. The difference from the present work is that here we treat the uncertainty in traditional range queries. A recent result [Cheng et al. 2003] gave probabilistic estimates of the answer to a few categories of queries over uncertain values of dynamic data. However, in terms of range queries, the work was limited to one-dimensional intervals.

A large body of work in moving objects databases has been concentrated on indexing in primal [Pfoser et al. 1999; Saltenis et al. 2000; Saltenis and Jensen 1999, 2002; Tayeb et al. 1998] or dual space [Agarwal et al. 2000; Kollios et al. 1999a, 1999b; Pasquale et al. 2003]. Theodoridis et al. [1999a, 1999b] presented

specifications of *what* an indexing of moving objects needs to consider, and generation of spatial datasets for benchmarking data. Along these lines was the recent work presented in Wolfson and Yin [2003]. These results will be useful in studying the most appropriate access method for processing the operators introduced in this article.

On the commercial side, there is a plethora of related GIS products [ESRI 1996; Geographic Data Technology Co. 2000; U.S. Dept. of Commerce 1991]; maps with real-time traffic information [Intelligent Transportation Systems 2000] and GPS devices and management software. IBM's DB2 Spatial Extender [Davis 1998], Oracle's Spatial Cartridge [Oracle Corporation 2000], and Informix Spatial DataBlade [Team 1999] provided several 2D spatial types (e.g., line, polyline, polygon, ...), and included a set of predicates (e.g., *intersects, contains*) and functions for spatial calculations (e.g., *distance*). However, the existing commercial products still lack the ability to model and query spatio-temporal contexts for moving objects.

8. CONCLUSIONS AND FUTURE WORK

We have proposed a model for representing moving objects under realistic assumptions of location uncertainty. We also introduced a set of operators which can be used to pose queries in that context. The model and the operators combine spatial, temporal, and uncertainty constructs, and have been implemented as part of our DOMINO project. We presented the processing algorithms for the proposed operators and analyzed their complexity. We also outlined some important practical observations, based on our experience with the DOMINO system. Let us point out that an interesting approach that we explored is to model the location uncertainty as a rectangle. This would capture the scenario where the moving object is known to be on the highway, but its exact lane is not known.

In terms of future work, an interesting problem is the management of the storage space needed for the trajectories. Namely, one can apply ideas similar to line simplification (cf. [Weibel 1997]) to reduce the size of a given trajectory. However, this will impose some extra imprecision when processing spatio-temporal queries and a choice of an appropriate metric is essential in guaranteeing bounds on the query errors [Cao et al. 2003]. Currently, we are investigating the problem of optimizing the simplification when updates to the motion plan (future trajectories) are present.

Another important problem is related to automatic updates of the trajectories which are affected by abnormal traffic conditions [Trajcevski et al. 2002b]. Consider, for example, the following query: "Retrieve all the objects which will possibly be within 2 miles from the United Center, sometime between 7:00 PM and 7:20 PM." If the query was posed at 6:00 PM and an accident occurred on some road segment at 6:30 PM, certain trajectories will have to be updated and, consequently, the answer set for the query may have to be recalculated. Currently, we are utilizing triggers to automatically update the answer set to such queries.

A particularly challenging problem is the one of query processing for spatio-temporal databases. We will investigate how to incorporate an indexing schema

within the existing ORDBMSs (cf. Chen et al. [1999]; Kornacker [1999]), and develop and experimentally test a hybrid indexing schema which would pick an appropriate access method for a particular environment. Along the lines of the recent results in Prabhakar et al. [2002], we will also tackle the problem of query optimization. For example, for Boolean combinations of the operators introduced in this work, one can process each operator separately and combine the results at the end. However, it can be shown that often this procedure can be significantly improved by considering the global query.

APPENDIX

A.1 Proof of Theorem 5.11

PROOF. ($\Rightarrow$) If there exists some PMC^T between t_b and t_e which is entirely inside VT_{sd}, then that PMC^T is the witness of path-connectedness of VT_{sd}.

($\Leftarrow$) The proof is "by construction," that is, it demonstrates how to construct a possible motion curve, from a given (existing) *path*. Without loss of generality, we will restrict our discussion to one segment of the trajectory (respectively, one segment of trajectory volume).

Assume that $VT_{sd} = VTr \backslash P_R$ is path-connected, between t_b and t_e. Then, there must exist some 3D curve, call it $\mathcal{C}$, which is entirely in $VT_{sd} = VTr \backslash P_R$, for every $t \in [t_b, t_e]$. If $\mathcal{C}$ is monotonic in the *time*-dimension, then it is the desired possible motion curve.

Assume not. Then, there must exist some point on $\mathcal{C}$, denote it $A(x_A, y_A, t_A)$, at which $\mathcal{C}$ has a local extremum with respect to the time-dimension (i.e., all the points of $\mathcal{C}$ within some ε-neighborhood of A have their time values smaller than t_A) and $t_A < t_e$. Since VT_{sd} is path-connected, there must exist another point on $\mathcal{C}$, call it $B(x_B, y_B, t_B)$, such that $t_B > t_A$. Let $A_1(x_A, y_A)$ and $B_1(x_B, y_B)$ denote the XY projections of A and B, respectively, and denote by $\overline{A_1B_1}$ the line-segment between A_1 and B_1. Also, let $\Pi_{X,Y}(\mathcal{C})$ denote the XY projection of $\mathcal{C}$. Clearly, both A_1 and B_1 are on $\Pi_{X,Y}(\mathcal{C})$. $\Pi_{X,Y}(\mathcal{C})$ is inside the uncertainty zone of the trajectory's route, but outside the region R. The construction of the portion of the possible motion curve between A and B can now be explained as follows.

Let $N(x_N, y_N)$ denote a point on $\overline{A_1B_1}$. Draw the line l_N which is perpendicular to $\overline{A_1B_1}$ through N and let $M_1(x_M, y_M)$ denote the *closest* point of intersection of l_N with $\Pi_{X,Y}(\mathcal{C})$. The vertical line (perpendicular to the XY plane) through M_1 will intersect $\mathcal{C}$ at some $M(x_M, y_M, t_M)$. Consider the point $M_c(x_M, y_M, t_{MN})$ for which

$$t_{MN} = t_A + \frac{\sqrt{(x_N - x_A)^2 + (y_N - y_A)^2}}{\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}} \cdot (t_B - t_A). \tag{4}$$

Clearly, $t_A \leq t_{MN} \leq t_B$. Thus, from the points of $\mathcal{C}$ between A and B, we can construct a curve which will be inside VT_{sd} (recall that $\Pi_{X,Y}(\mathcal{C})$ is in the uncertainty zone but outside R), and the time values of its points will monotonically increase between t_A and t_B. Since we assumed that between t_b and t_A the time values of the points along $\mathcal{C}$ are monotonically increasing too, we have actually

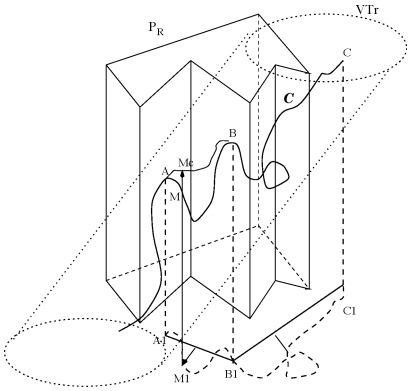


Fig. 21. Construction of a possible motion curve from a path.

constructed a possible motion curve which is entirely within VT_{sd} between t_b and t_B. By repeating the construction in a recursive manner from the point B on $\mathcal{C}$, we will obtain the desired possible motion curve between t_b and t_e.

The concepts that we introduced and used in this proof are illustrated in Figure 21. Let us point out that the reason we needed the *closest* point of intersection between l_N and $\Pi_{X,Y}(\mathcal{C})$ is the uniqueness of the constructed possible motion curve (i.e., we want to handle situations like the ones depicted in the right portion of Figure 21).

As a last comment, observe that as an extreme case, two points along the path, say W and Z, may be connected by some curve which is entirely in some horizontal plane. In this case, one can always find an ε-neighborhood around W which has a point, say S, on the path, with a lower time value, and apply the same argument as above between S and Z. This completes the proof. □

A.2 Calculating the Critical Times

When evaluating the *Possibly_Always_Inside* predicate for a nonconvex, simple polygon as query region, we need to identify the set of critical points.

Let $B_{11}B_{12}B_{21}B_{22}$ denote a vertical rectangle corresponding to a side of some prism. Let A_j and A_{j+1} denote the endpoints of the jth trajectory segment between the times t_j and t_{j+1}.

The general case of the time $t \in [t_j, t_{j+1}]$ being a critical time in the jth trajectory segment, occurs when the intersection of the uncertainty area at t with the rectangle $B_{11}B_{12}B_{21}B_{22}$ is a single 3D point.

Part (a) of Figure 22 illustrates the concepts introduced in Definition 5.13 in Section 5. Observe that we have two *critical* time values, t_1 and t_2. Also note that the time $t_3(t_1 < t_3 < t_2)$ is not a critical one because at t_3 the uncertainty area has more than one intersection point with $B_{11}B_{12}B_{21}B_{22}$ (infinitely many of them). For a given vertical rectangle and a segment of a trajectory volume, the number of significant times may be 0, 1, 2, or infinitely many, as we demonstrate below.

Since the query region is represented as a polygon in the XY plane, each edge of the polygon is part of a line. with an equation of the form $a \cdot x + b \cdot y + c = 0$.

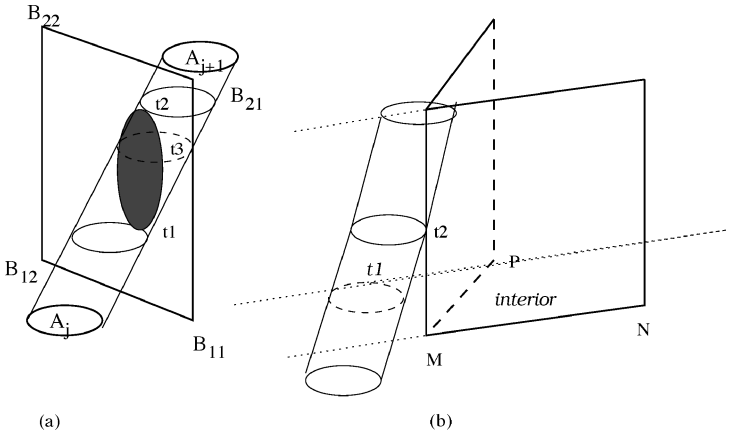


Fig. 22. Computing the times of intersection.

Hence, each edge of the polygon, when considered for a given time interval will represent a vertical 3D rectangle.

As we stated in Section 3, at each time t, the horizontal circle which is the boundary of the trajectory volume is defined as: $(x - (x_i + v_i^x \cdot t))^2 + (y - (y_i + v_i^y \cdot t))^2 = r^2$.

Substituting for y (or for x, if $b = 0$) from the equation of the line defining the edge, we have

$$\left(x - \left(x_i + v_i^x \cdot t\right)\right)^2 + \left((m \cdot x + b) - \left(y_i + v_i^y \cdot t\right)\right)^2 = r^2. \tag{5}$$

This yields an equation in x and t of the form

$$K \cdot x^2 + x \cdot (R + S \cdot t) + U \cdot t^2 + V \cdot t + W = 0, \tag{6}$$

where K, R, S, U, V, W are constants (in terms of the other constants from Equation (5)). Solving for x, as a parameter of t, we get

$$x_{1,2} = \frac{-(R + S \cdot t) \stackrel{+}{-} \sqrt{(R + S \cdot t)^2 - 4 \cdot K \cdot (U \cdot t^2 + V \cdot t + W)}}{2 \cdot K}. \tag{7}$$

According to Definition 5.13, a time t is critical iff we have a unique solution of Equation (7) in x. In order to achieve it, all we need is to set the discriminant to 0,

$$(R + S \cdot t)^2 - 4 \cdot K \cdot (U \cdot t^2 + V \cdot t + W) = 0, \tag{8}$$

and solve it for t. Equation (8) is quadratic in t and, in general, it will have two solutions. However, several cases may arise with respect to the values of the solutions (and, implicitly, due to the different possibilities for the values of the constants in the Equation 7):

—If both of them are complex, that is, the discriminant of Equation (7) does not change its sign, then

(1) if the discriminant is positive, we always have two solutions for x in Equation 7 and, therefore, there is no unique time;

(2) if the discriminant is negative, then we have no real solution for x in Equation (7) and, again, we do not have any critical times.

—If both of the solutions of the Equation (8) are real, then

(1) both of them are in the interval $[t_i, t_{i+1}]$, which yields two critical times;
(2) one of them is outside the interval $[t_i, t_{i+1}]$, which yields only one critical time;
(3) both of them are outside the interval $[t_i, t_{i+1}]$, which yields no critical times.

—If the left-hand side of Equation (8) is identically equal to zero, for every $t \in [t_i, t_{i+1}]$, then we have infinitely many critical times. The geometric interpretation of this case is that the 3D vertical rectangle is tangential to the trajectory volume.

Part (b) of Figure 22 illustrates another issue which needs to be considered. The actual time in which the trajectory volume will "touch" the vertical plane defined by the line $\overline{MN}$ is t_1. However, since the valid edge is between M and N, we need to take t_2 as the significant time point, because the intersection at t_1 is outside the XY boundaries of the segment $\overline{MN}$. Thus, when calculating the significant times, we need to substitute the values obtained as solutions to Equation (8) back into Equation (7), and check if the solutions for x are within the bounds of the particular line segment. If not, we have to substitute the boundary values for x back into Equation (3), and solve it for t (in which case, again, we may get zero, one, or two critical times).

ACKNOWLEDGMENTS

The authors are indebted and extremely grateful to the anonymous referees for their insightful comments, both of a technical and structural nature, which improved the quality of this work. We wish to thank Pankaj Agarwal for pointing to us a way to significantly simplify the algorithms for processing the *Possibly_Sometime_Inside* and *Always_Possibly_Inside* operators for convex query region and Julien Basch for his insights on some of the complexity results. We also thank Damian Roqueiro for his help in providing some experimental data.

REFERENCES

AGARWAL, A. K., ARGE, L., AND ERICKSON, J. 2000. Indexing moving points. In *Proceedings of the 19th International ACM Conference on Principles of Database Systems (PODS) Conference*. ACM Press, New York, NY, 175–186.

AGARWAL, P. K. 1990a. Partitioning arrangements of lines: 1. An efficient deterministic algorithm. *Discrete Computat. Geom. 5*, 449–483.

AGARWAL, P. K. 1990b. Partitioning arrangements of lines: 2. Applications. *Discr. Computat. Geom. 5*, 533–573.

AGARWAL, P. K., FLATO, E., AND HALPERIN, D. 2002. Polygon decomposition for efficient construction of minkowski sums. *Computat. Geom. 21*, 1-2, 39–61.

BASCH, J. 2004. Personal communication.

BASCH, J., GUIBAS, L. J., AND RAMKUMAR, G. D. 2003. Reporting red-blue intersections between two sets of connected segments. *Algorithmica 35*, 1, 1–20.

BENTLEY, J. L. AND OTTMANN, T. A. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput. 28*, 9, 643–647.

BRUNDICK, F. S. AND HARTWIG, G. W. 1997. Model-based situational awareness. In *Proceedings of the Joint Service Combat Identification Systems Conference (CISC-97)*.

CAO, H., WOLFSON, O., AND TRAJCEVSKI, G. 2003. Spatio-temporal data reduction with deterministic error bounds. In *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing*.

CAREY, M., CHAMBERLIN, D., NARAYANAN, S., VANCE, B., DOOLE, D., RILEAU, S., SWEGARMAN, R., AND MATTOS, N. 1999. O-O what have they done to DB2. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*. 542–554.

CHAMBERLAIN, S. 1995. Model-based battle command: A paradigm whose time has come. In *Proceedings of the Symposium on C2 Research and Technology, NDU*.

CHAZELLE, B. 1991. Triangulating a simple polygon in linear time. *Discr. Computat. Geom. 6*, 485–524.

CHAZELLE, B. AND EDELSBRUNNER, H. 1992. An optimal algorithm for intersecting line segments in the plane. *J. Assoc. Comput. Mach. 39* 1, 1–54.

CHEN, W., CHOW, J., FUH, Y., GRANDBOIS, J., JOU, M., MATTOS, N., TRAN, B., AND WANG, Y. 1999. High level indexing of user-defined types. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*. 554–564.

CHENG, R., KALASHNIKOV, D., AND PRABHAKAR, S. 2003. Evaluating probabilistic queries over imprecise data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 551–562.

CHIN, F. Y. L., SNOEYINK, J., AND WANG, C. A. 1999. Finding the medial axis of a simple polygon in linear time. *Discr. Computat. Geom. 21*, 3, 405–420.

DAVIS, J. R. 1998. *Managing Geo-Spatial Information within the DBMS*. IBM DB2 Spatial Extender (IBM White Paper).

DREYFUS, S. E. 1969. An appraisal of some shortest-path algorithms. *Operat. Res. 17*, 3.

ERWIG, M., SCHNEIDER, M., AND GÜTING, R. H. 1998. Temporal and spatio-temporal datasets and their expressive power. In *Proceedings of the Advances in Database Technologies (ER'98, Workshop on Spatio-Temporal Management)*. Lecture Notes in Computer Science, vol. 1552. Springer-Verlag, Berlin, Germany, 454–465.

ESRI. 1996. *ArcView GIS:The Geographic Information System for Everyone*. Environmental Systems Research Institute Inc., St. Paul, MN.

FORLIZZI, L., GÜTING, R. H., NARDELLI, E., AND SCHNEIDER, M. 2000. A data model and data structures for moving objects databases. In *Proceedings of the ACM SIGMOD Internation Conference on Management of Data*. ACM Press, New York, NY, 319–330.

GAEDE, V. AND GÜNTHER, O. 1998. Multidimensional access methods. *ACM Comput. Surv. 11*, 4.

GENESERETH, M. R. AND NILSSON, N. J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Francisco, CA.

Geographic Data Technology Co. 2000. *GDT Maps*. Geographic Data Technology Co., Lebanon, NH. Website: http://www.geographic.com.

GÜTING, R. H., BÖHLEN, M. H., ERWIG, M., JENSEN, C., LORENTZOS, N., SCHNEIDER, M., AND VAZIRGIANNIS, M. 2000. A foundation for representing and queirying moving objects. *ACM Trans. Database Syst. 25*, 1, 1–42.

GÜTING, R. H., BÖHLEN, M. H., ERWIG, M., JENSEN, C. S., LORENTZOS, N., SCHNEIDER, M., AND VIQUEIRA, J. R. R. 2003. Spatio-temporal models and languages: An approach based on data types. In *Spatio-Temporal Databases: The Chorochronos Approach*, M. Koubarakis et al., Eds. Lecture Notes in Computer Science, vol. 2520. Springer-Verlag, Berlin, Germany, 117–177.

HARTWIG, G. W., BRUNDICK, F. S., AND KOTHENBEUTEL, C. S. 1996. Tactically significant route planning. Tech. Rep. ARL-TR-1139. Army Research Lab, Aberdeen Proving Ground, MJ.

HIGHTOWER, J. AND BORRIELO, G. 2001. Location systems for ubiquitous computing. *IEEE Comput. Mag. 34*, 8 (Aug.), 57–66.

Intelligent Transportation Systems. 2000. *ITS maps*. Intelligent Transportation Systems. http://www.itsonline.com.

KEDEM, K., LIVNE, R., PACH, J., AND SHARIR, M. 1986. On the union of Jordan-regions and collision-free translational motion amidst polygonal obstacles. *Discr. Computat. Geom. 1*, 59–71.

KOLLIOS, D., GUNOPULOS, D., AND TSOTRAS, V. J. 1999a. On indexing mobile objects. In *Proceedings of the 18th Internations ACM PODS Conference on Principles of Database Systems*. ACM Press, New York, NY, 261–272.

KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. J. 1999b. Nearest neighbour queries in a mobile environment. In *Spatio-Temporal Database Management*. In *Proceedings of the Inernational Workshop STDBM '99*. Lecture Notes in Computer Science, vol. 1678. Springer-Verlag, Berlin, Germany, 119–134.

KORNACKER, M. 1999. High-sperformance extensible indexing. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*. 699–708.

MAIRSON, H. G. AND STOLFI, J. 1988. Reporting and counting intersections between two sets of line segments. In *Theoretical Foundations of Computer Science, NATO ASI.*, Vol. F40. Kluwer Academic Publishers, Norwell, MA.

Oracle Corporation. 2000. *Oracle8i: Spatial Cartridge User's Guide and Reference, Release 8.0.4*. Oracle Corporation. Redwood Shores, CA. Available online at http://technet.oracle.com/docs/products/oracle8/doc-index.htm.

O'ROURKE, J. 2000. *Computational Geometry in C*. Cambridge University Press, Cambridge, U.K.

PALAZZI, L. AND SNOEYINK, R. 1994. Counting and reporting red/blue segment intersections. *Graph. Models Image Process. 56*, 4, 304–310.

PASQUALE, A. D., FORLIZZI, L., JENSEN, C. S., MANOLOPOULOS, Y., NARDELLI, E., PFOSER, D., PROIETTI, G., SALTENIS, S., THEODORIDIS, Y., AND TZOURAMANIS, T. 2003. Access methods and query processing techniques. In *Spatio-Temporal Databases: The Chorochronos Approach*, M. Koubarakis et al., Eds. Lecture Notes in Computer Science, vol. 2520. Springer-Verlag, Berlin, Germany, 203–263.

PFOSER, D. AND JENSEN, C. 1999. Capturing the uncertainty of moving objects representation. In *Proceedings of the International Symposium on Advances in Spatial Databases (SSD)*. 111–132.

PFOSER, D., THEODORIDIS, Y., AND JENSEN, C. 1999. Indexing trajectories of moving point objects. Tech. rep. 99/07/03. Dept. of Computer Science, University of Aalborg, Aalborg, Denmark.

PITOURA, E. AND SAMARAS, G. 2001. Locating objects in mobile computing. *IEEE Trans. Knowl. Data Eng. 13*, 4, 571–592.

PRABHAKAR, S., XIA, Y., KALASHNIKOV, D., AREF, W., AND HAMBRUSCH, S. 2002. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput. 51*, 10, 1124–1140.

SALTENIS, S. AND JENSEN, C. 1999. R-tree based indexing of general spatio-temporal data. Tech. rep. TR-45. TimeCenter. Available online at www.cs.auc.dk/TimeCenter/pub.html.

SALTENIS, S. AND JENSEN, C. 2002. Indexing of moving objects for location-based services. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 463–472.

SALTENIS, S., JENSEN, C. S., LEUTENEGGER, S. T., AND LOPEZ, M. A. 2000. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM Press, New York, NY, 331–342.

SHARIR, M. AND AGARWAL, P. K. 1995. *Davenport-Schinzel Sequences and Their Geometric Applications*. Campbridge University Press, Cambridge, U.K.

SISTLA, A., WOLFSON, P., CHAMBERLAIN, S., AND DAO, S. 1998. Querying the uncertain positions of moving objects. In *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada, Eds. Lecture Notes in Computer Science, vol. 1399. Springer-Verlag, Berlin, Germany, 310–337.

SISTLA, A. P., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. 1997. Modeling and querying moving objects. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 422–432.

SUTHERLAND, W. A. 1978. *Introduction to Metric and Topological Spaces*. Oxford University Press, Oxford, U.K.

TANSEL, A., CLIFFORD, J., JAJODIA, S., SEGEV, A., AND SNODGRASS, R. 1993. *Temporal Databases: Theory and Implementation*. Benjamin Cummings Publishing Co., San Francisco, CA.

TAYEB, J., ULUSOY, O., AND WOLFSON, O. 1998. A quadtree-based dynamic attribute indexing method. *Comput. J. 41*, 3, 185–200.

TEAM, I. D. 1999. Informix datablade technology: Transforming data into smart data. Informix Press, Menlo Park, CA.

THEODORIDIS, Y., SELLIS, T., PAPADOPOULOS, A. N., AND MANOLOPOULOS, Y. 1999a. Specifications for efficient indexing in spatiotemporal databases. In *Proceedings of the International Conferece on*

Statistical and Scientific Database Management (SSDBM). IEEE Computer Society Press, Los Alamitos, CA, 123–132.

THEODORIDIS, Y., SILVA, J. R. O., AND NASCIMENTO, M. A. 1999b. On the generation of spatiotemporal datasets. In *Proceedings of the International Symposium on Large Spatial Databases*. 147–164.

TRAJCEVSKI, G., WOLFSON, O., CAO, H., LIN, H., ZHANG, F., AND RISHE, N. 2002a. Managing uncertain trajectories of moving objects with DOMINO. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*. 218–225.

TRAJCEVSKI, G., WOLFSON, O., XU, B., AND NELSON, P. 2002b. Real-time traffic updates in moving object databases. In *Proceedings of the MDDS Workshop, in conjunction with Database and Expert Systems Applications (DEXA))*. 698–704.

U.S. Dept. of Commerce. 1991. *Tiger/Line Census Files: Technical Documentation*. U.S. Dept. of Commerce, Washington, D.C.

VAZIRGIANNIS, M., THEODORIDIS, Y., AND SELLIS, T. 1998. Spatiotemporal composition and indexing for large multimedia applications. *Multimed. Syst. J. 6*, 4.

VAZIRGIANNIS, M. AND WOLFSON, O. 2001. A spatiotemporal model and language for moving objects on road networks. In *Proceedings of the Symposium on Spatial and Temporal Databases (SSTD)*.

WANT, R. AND SCHILIT, B. 2001. Expanding the horizons of location aware computing. *IEEE Comput. Mag. 34*, 8 (Aug.), 31–34.

WEIBEL, R. 1997. Generalization of spatial data: Principles and selected algorithms. In *Algorithmic Foundations of Geographic Information Systems*, M. J. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, Eds. Lecture Notes in Computer Science, vol. 1340. Springer-Verlag, Berlin, Germany, 99–152.

WOLFSON, O., CAO, H., LIN, H., TRAJCEVSKI, G., ZHANG, F., AND RISHE, N. 2002. Management of dynamic location information in domino. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*.

WOLFSON, O., CHAMBERLAIN, S., DAO, S., JIANG, L., AND MENDEZ, G. 1998. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 588–596.

WOLFSON, O., SISTLA, A. P., CHAMBERLAIN, S., AND YESHA, Y. 1999. Updating and querying databases that track mobile units. *Distrib. Parall. Databases 7*, 3, 257–387.

WOLFSON, O. AND YIN, H. 2003. Accuracy and resource consumption in tracking and location prediction. In *Proceedings of the Symposium on Spatial and Temporal Databases (SSTD)*, 325–343.

Received June 2002; revised June 2003, January 2004; accepted February 2004