Modeling and Querying Moving Objects

A. Prasad Sistla* Ouri Wolfson† Sam Chamberlain‡ Son Dao§

Abstract

In this paper we propose a data model for representing moving objects in database systems. It is called the Moving Objects Spatio-Temporal (MOST) data model. We also propose Future Temporal Logic (FTL) as the query language for the MOST model, and devise an algorithm for processing FTL queries in MOST.

1. Introduction

Existing database management systems (DBMS's) are not well equipped to handle continuously changing data, such as the position of moving objects. The reason for this is that in databases, data is assumed to be constant unless it is explicitly modified. For example, if the salary field is 30K, then this salary is assumed to hold (i.e. 30K is returned in response to queries) until explicitly updated. Thus, in order to represent moving objects (e.g. cars) in a database, and answer queries about their position (e.g., How far is the car with license plate RWW860 from the nearest hospital?) the car's position has to be continuously updated. This is unsatisfactory since either the position is updated very frequently (which would impose a serious performance and wireless-bandwidth overhead), or, the answer to queries is outdated. Furthermore, it is possible that due to disconnection, an object cannot continuously update its position.

In this paper we propose to solve this problem by representing the position as a function of time; it changes as time passes, even without an explicit update. So, for example, the position of a car is given as a function of its motion vector (e.g., north, at 60 miles/hour). In other words, we consider a higher level of data abstraction, where an object's motion vector (rather than its position) is represented as an attribute of the object. Obviously the motion vector of an object can change (thus it can be updated), but in most cases it does so less frequently than the position of the object.

*Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607

†Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607 CESDIS, NASA Goddard Space Flight Center, Code 930.5, Greenbelt, MD 20771

‡Army Research Laboratory, Aberdeen Proving Ground, MD

§Hughes Research Laboratories, Information Sciences Laboratory, Malibu, CA

In this paper we propose a data model called Moving Objects Spatio-Temporal (or MOST for short) for databases with dynamic attributes, i.e. attributes that change continuously as a function of time, without being explicitly updated. In other words, the answer to a query depends not only on the database contents, but also on the time at which the query is entered. Furthermore, we explain how to incorporate dynamic attributes in existing data models and what capabilities need to be added to existing query processing systems to deal with dynamic attributes.

Clearly, our proposed model enables queries that refer to future values of dynamic attributes, namely future queries. For example, consider an air-traffic control application, and suppose that each object in the database represents an aircraft and its position. Then the query Q = “retrieve all the airplanes that will come within 30 miles of the airport in the next 10 minutes” can be answered in our model. In [14] we introduced a temporal query language called Future Temporal Logic (FTL). The language is more natural and intuitive to use in formulating future queries such as Q. Unfortunately, due to the difference in data models, the algorithm developed in [14] for processing FTL queries does not work for MOST databases. Therefore, in this paper we develop an algorithm for processing an important subclass of FTL queries for MOST databases.

The answer to future queries is usually tentative in the following sense. Suppose that the answer to the above query Q contains airplane a. It is possible that after the answer is presented to the user, the motion vector of a changes in a way that steers a away from the airport, and the database is updated to reflect this change. Thus a does not come within 30 miles of the airport in the next 10 minutes. Therefore, in this sense the answer to future queries is tentative, i.e. it should be regarded as correct according what is *currently* known about the real world, but this knowledge (e.g. the motion vector) can change.

Continuous queries is another topic that requires new consideration in our model. For example, suppose that there is a relation MOTELS (that resides, for example, in a satellite) giving for each motel its geographic-coordinates, room-price, and availability. Consider a moving car issuing a query such as ”Display motels (with availability and cost) within a radius of 5 miles”, and suppose that the query is continuous, i.e., the car requests the answer to the query to be continuously updated. Observe that the answer changes with the car movement. When and how often should the query be reevaluated? Our query processing algorithm facilitates a single evaluation of the query; reevaluation has to occur only if the motion vector of the car changes.

We assume that there is a natural, user-friendly way of entering into the database the current position and motion vector of objects. For example, a point on a screen may represent the car's current position[1], and the driver may draw around it, on the touch-sensitive screen, a circle with a radius of 5 miles; then s/he may name the circle C and indicate that C moves as a rigid body having the motion vector of the car. This way the driver specifies a circle and its motion vector, and the car's computer can create a data representation of the moving object. The computer can automatically update the motion vector of C when it senses a changes in speed or direction. In other applications, such as air-traffic-control, there may be other means of entering objects and their motion vector.

Generally, a query in our data model involves spatial objects (e.g. points, lines, regions, polygons) and their motion vector. Some examples of queries are: "Retrieve the objects that will intersect the polygon P within 3 minutes", or, "Retrieve the objects that will intersect P within 3 minutes, and have the attribute PRICE $\leq$ 100", or, "Retrieve the objects that will intersect P within 3 minutes, and stay in P for 1 minute", or "Retrieve the objects that will intersect P within 3 minutes, stay in the polygon for 1 minute, and 5 minutes later enter another polygon Q".

For performance considerations, in answering queries of this type, we would like to avoid examining each moving object in the database. In other words, we would like to index dynamic attributes. The problem with a straightforward use of spatial indexing is that since objects are continuously moving, the spatial index has to be continuously updated, an unacceptable solution. Therefore, we introduce one possible method of indexing dynamic attributes, which guarantees logarithmic (in the number of objects) access time.

In summary, in this paper we introduce the MOST data model whose main contributions are as follows.

- A new type of attributes called dynamic attributes. A method of indexing dynamic attributes is introduced. The principles for incorporating dynamic attributes on top of existing DBMS's are outlined.
- Adaptation of FTL as a query language in MOST. An efficient algorithm is devised for processing queries specified in an important subclass of FTL.

The rest of this paper is organized as follows. In section 2 we introduce the MOST data model and discuss the types of queries it supports in terms of database histories. In section 3 we define the FTL query language, i.e. its syntax and semantics in the context of MOST; we also demonstrate the language using examples, and we introduce an algorithm for processing FTL queries. In section 4 we discuss a method of indexing dynamic attributes. In section 5 we discuss several issues related to implementation of the MOST data model, including: MOST on top of existing DBMS's, queries issued by moving objects, and distributed query processing. In section 6 we compare our work to relevant literature, and in section 7 we discuss future work.

[1]this position may be supplied, for example, by a Geographic Positioning System (GPS) on board the car.

2. The MOST data model

The traditional database model is as follows. A *database* is a set of object-classes. A special database object called *time* gives the current time at every instant; its domain is the set of natural numbers, and its value increases by one in each clock tick. An *object-class* is a set of attributes. For example, MOTELS is an object class with attributes Name, Location, Number-of-rooms, Price-per-room, etc.

Some object-classes are designated as *spatial*. A spatial object class has three attributes called X.POSITION, Y.POSITION, Z.POSITION, denoting the object's position in space. The spatial object classes have a set of spatial methods associated with them. Each such method takes spatial objects as arguments. Intuitively, these methods represent spatial relationships among the objects at a certain point in time, and they return true or false, indicating whether or not the relationship is satisfied at the time. For example, INSIDE(o,P) and OUTSIDE(o,P) are spatial relations. Each one of them takes as arguments a point-object o and a polygon-object P in a database state; and it indicates whether or not o is inside (outside) the polygon P in that state. Another example of a spatial relation is WITHIN-A-SPHERE(r, o_1 , ... ,o_k). Its first argument is a real number r, and its remaining arguments are point-objects in the database. WITHIN-A-SPHERE indicates whether or not the point-objects can be enclosed within a sphere of radius r.

There may also be methods that return an integer value. For example, the method DIST(o_1, o_2) takes as arguments two point-objects and returns the distance between the point-objects.

To model moving objects, in subsection 2.1 we introduce the notion of a dynamic attribute, and in subsection 2.2 we relate it to the concept of a database history. In subsection 2.3 we discuss three different types of queries that arise in this model.

2.1 Dynamic attributes

Each attribute of an object-class is either static or dynamic. Intuitively, a static attribute of an object is an attribute in the traditional sense, i.e. it changes only when an explicit update of the database occurs; in contrast, a dynamic attribute changes over time according to some given function, even if it is not explicitly updated. For example, consider a moving object whose position in two-dimensional space at any point in time is given by values of the x, y coordinates. Then each one of the object's coordinates is a dynamic attribute.

Formally, a *dynamic attribute* A is represented by three sub-attributes, $A.value$, $A.updatetime$, and $A.function$, where $A.function$ is a function of a single variable t that has value 0 at $t = 0$. The *value* of a dynamic attribute depends on the time, and it is defined as follows. At time $A.updatetime$ the value of A is $A.value$, and until the next update of A the value of A at time $A.time + t_0$ is given by $A.value + A.function(t_0)$. An explicit update of a dynamic attribute may change its value sub-attribute, or its function sub-attribute, or both sub-attributes.

In addition to querying the value of a dynamic attribute, a user can query each sub-attribute independently. Thus, the user can ask for the objects for which $X.POSITION.function = 5 \cdot t$, i.e. the objects whose speed in the X direction is 5.

There are two possible interpretations of $A.updatetime$, corresponding to valid-time and transaction-time (see [8]). In the first interpretation, it is the time at which the update occurred in the real world system being modeled, e.g. the time at which the vehicle changed its motion vector. In this case, along with the update, the sensor has to send to the database $A.updatetime$. In the second interpretation, $A.updatetime$, is simply the time-stamp when the update was committed by the DBMS. In this paper we assume that the database is updated instantaneously, i.e. the valid-time and transaction-time are equal.

When a dynamic attribute is queried, the answer returned by the DBMS consists of the value of the attribute at the time the query is entered. In this sense, our model is different than existing database systems, since, unless an attribute has been explicitly updated, a DBMS returns the same value for the attribute, independently of the time at which the query is posed. So, for example, in our model the answer to the query: "retrieve the current x-position of object o" depends on the value of the dynamic attribute X.POSITION at the time at which the query is posed. In other words, the answer may be different for time-points t_1 and t_2, even though the database has not been explicitly updated between these two time-points.

In this paper we are concerned with dynamic attributes that represent spatial coordinates, but the model can be used for other hybrid systems, in which dynamic attributes represent, for example, temperature, or fuel consumption.

2.2 Database histories

In existing database systems, queries refer to the current database state, i.e. the state existing at the time the query is entered. For example, the query can request the current price of a stock, or the current position of an object, but not future ones. Consequently, existing query languages are *nontemporal*, i.e. limited to accessing a single (i.e. the current) database state. In our model, the database implicitly represents future states of the system being modeled (e.g. future positions of moving objects), therefore we can envision queries pertaining to the future, rather than the current state of the system being modeled. For example, a moving car may request all the motels that it will reach (i.e. come within 500 yards of) in the next 20 minutes. To interpret this type of queries, i.e. queries referring to dynamic attributes, we need the notion of a database history, i.e. a sequence of database states.

A *database state* is a mapping that associates a set of objects of the appropriate type to each object class. Each database state has an associated *time stamp*. In the state, the value of a dynamic attribute is taken to be the value of the attribute at the time t = *time stamp*. Queries are interpreted over database histories. A *database history* is an infinite sequence of database states, one for each clock tick, according to a fixed global clock. Thus, the time stamps along the database history are strictly increasing. Furthermore, the value of an attribute A of an object may be different in two consecutive database states for one of two reasons. First, A was explicitly updated, and second, A is a dynamic variable that was not explicitly updated but its value is different at the consecutive clock ticks.

At a particular point in time t, the database states with a lower time-stamp than t are called the *past database-history*. However, the history also contains an infinite number of states in the *future database-history*, i.e. states with a time-stamp higher than the current time t. Each state in the future history is identical to the state at time t, except for the value of the dynamic attributes. The value of a dynamic attribute A in a future state with time-stamp t' is taken to be the value of A at time t'. This value is computed according to the $A.value$ and $A.function$ at time t. Thus, although the database contents are identical throughout the future database-history, the values of a dynamic attribute may not be identical.

We would like to emphasize at this point that the database history is an abstract concept, introduced solely for the providing formal semantics to our temporal query language, FTL. The database history does not consume space, since we do not save information about the history.

2.3 Three types of MOST queries

A *query* is a predicate over the database history (rather than a predicate over a single database state, as in traditional databases). The *answer* to a query is defined when the predicate is satisfied, and it consists of the set of instantiations of the variables that satisfy the predicate. In our model we need to distinguish between three types of queries; instantaneous, continuous and persistent. The same query may be entered as instantaneous, continuous and persistent, producing different results in each case. These types differ depending on the history on which the query is evaluated, and on the evaluation time. In contrast, in traditional databases the situation is simpler. There are two types of queries, instantaneous and continuous. An instantaneous query is a predicate on the current database state, and a continuous query is a predicate on each one of the future states.

Formally, an *instantaneous* query at time t is a query evaluated on the infinite history starting at t, i.e. the time when the query is entered. t is usually the time when the query is entered. For example, the query Q = "Display the motels within 5 miles of my position", when considered as an instantaneous query returns a set of motels presented to the user immediately after the query is evaluated.

Observe that an instantaneous query may refer to the future history, and it may refer to more than one database state. For example, "Display the motels that I will reach within 3 minutes " refers to all the states with a time-stamp between now and three minutes later.

Obviously, since an instantaneous query is evaluated on an infinite history, its answer may be infinite. For example, the query: "Display the tuples (motel,reaching-time) representing the motels that I will reach, and the time when I will do so" may have an infinite answer. To cope with this situation we will assume in this paper that a continuous query expires after a predefined (but very large) amount of time. There are other ways of dealing with this problem (they involve a finite representation of infinite sets), but these are beyond the scope of this paper.

To motivate the second type of query, assume that a satisfactory motel is not found as a result of the instantaneous query Q, since, for example, the price is too high for the value. However, the answer to Q changes as the car moves, even if the database is not updated. Thus, the traveler may wish to make the query continuous, i.e. request the system to regard it as an instantaneous query being continuously reissued at each clock tick (while the car is moving), until cancelled (e.g. until a satisfactory motel is found). Formally, a *continuous* query at time t is a sequence of instantaneous queries, one for each point in time $t' > t$ (i.e. the query is considered on the infinite history starting at time t'). If the answer to a continuous query is presented to the user on a screen, the display may change over time, even if the database is not updated.

Clearly, continuously evaluating a query would be very inefficient. Rather, when a continuous query is entered our processing algorithm evaluates the query once, and returns a set of tuples. Each tuple consists of an instantiation ρ of the predicate's variables (i.e. an answer to the query when considered in the noncontinuous sense) and a time interval $begin$ to end. The tuple $(\rho, begin, end)$ indicates that ρ is in the answer of the instantaneous queries from time $begin$ until the time end. The set of tuples produced in response to a continuous query CQ is called $Answer(CQ)$.

Obviously, an explicit update of the database may change a tuple in $Answer(CQ)$. For example, it is possible that the query evaluation algorithm produces the tuple $(o, 5, 7)$, indicating that o satisfies the query between times 5 and 7. If the motion vector of o is updated before time 5, the tuple may need to be replaced by, say $(o, 6, 7)$, or it may need to be deleted. Therefore, a continuous query CQ has to be reevaluated when an update occurs that may change the set of tuples $Answer(CQ)$. In this sense $Answer(CQ)$ is a materialized view. However, a continuous query in our model is different than a materialized view, since the answer to a continuous query may change over time even if the database is not updated.

Finally, the third type of query is a persistent query. Formally, a *persistent* query at time t is a sequence of instantaneous queries on the infinite history starting at t. Observe that, in contrast to a continuous query, the different instantaneous queries comprising a persistent query have the same starting point in the history. These histories may differ for different instantaneous queries due to database updates executed after time t.

To realize the need for persistence, consider the query R = "retrieve the objects whose speed in the direction of the X-axis doubles within 10 minutes". Suppose that the query is entered as persistent at time 0. Assume that for some object o, at time 0 the value of the dynamic attribute X.POSITION changes according to the function $5t$ (recall, t is time, i.e. the speed is 5). At time 0 no objects will be retrieved, since for each object, the speed is identical in all future database states; only the location changes from state to state. Suppose further that after one minute the function is explicitly updated to $7t$, and after another minute it is explicitly updated to $10t$. Then, the speed in the X direction has changed from 5 at time 0 to 10 at time 2, and hence, at time 2 object o should be retrieved as an answer to R. But if we consider the query R as instantaneous or continuous o will never be retrieved, since starting at any point in time, the speed of o is identical in all states of the future database history. When entered as persistent, the query R is considered as a sequence of instantaneous queries, all operating on the history that starts at time 0. At time 2 this history reflects a change of the speed from 2 to 4 within two minutes, thus o will be retrieved at that time.

In summary, the three types of queries are illustrated in the following figure.

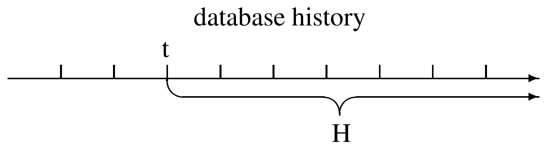

Figure 1. database history
(a) An *instantaneous* query at time t is evaluated on the history H_t (i.e. the future history beginning at t).
(b) A *continuous* query at time t is a sequence of instantaneous queries at each time $t' \geq t$.
(c) A *persistent* query at time t is a sequence of instantaneous queries, all at time t. The queries are evaluated at each time $t' \geq t$ the database is updated.

In contrast to continuous queries, the evaluation of persistent queries requires saving of information about the way the database is updated over time, and we postpone the subject of persistent query evaluation to future research. Observe that persistent queries are relevant even in the absence of dynamic variables. In [14] we developed an algorithm for processing FTL persistent queries. Unfortunately, that algorithm does not work when the queries involve dynamic variables.

Observe that continuous and persistent queries can be used to define temporal triggers. Such a trigger is simply one of these two types of queries, coupled with an action and possibly an event.

3 The FTL language

In this section we first motivate the need for our language (subsection 3.1), then we present the syntax (3.2) and semantics (3.3) of FTL. In subsection 3.4 we demonstrate the language through some example, and in subsection 3.6 we present our query processing algorithm.

3.1 Motivation

A regular query language such as SQL or OQL can be used for expressing temporal queries on moving objects, however, this would be cumbersome. The reason is that these languages do not have temporal operators, i.e. keywords that are natural and intuitive in the temporal domain. Consider for example the query Q: "Retrieve the pairs of objects o and n such that the distance between o and n stays within 5 miles until they both enter polygon P".

Assume that for each predicate G there are functions $begin\_time(G)$ and $end\_time(G)$ that give the beginning and ending times of the first time-interval during which G is satisfied; also assume that "now" denotes the current time. Then the query Q would be expressed as follows.

RETRIEVE o,n
FROM Moving-Objects
WHERE begin_time($DIST(o, n) \leq 5$) $\leq$ now
and end_time($DIST(o, n) \leq 5$) $\geq$
begin_time($INSIDE(o, P)) \wedge INSIDE(n, P)$).

At the end section 3.2 we show how the query Q is expressed in our proposed language, FTL. Clearly, the query in FTL is simpler and more intuitive. The SQL and OQL queries may be even more complex when considering the fact that the spatial predicates may be satisfied for more than one time interval. Thus, we may need the functions $begin\_time1$ and end_time1 to denote the beginning and ending times of the first time interval, $begin\_time2$ and end_time2 to denote the beginning and ending of the second time interval, etc.

3.2 Syntax

The FTL query language enables queries pertaining to the **future** states of the system being modeled. Since the language and system are designed to be installed on top of an existing DBMS, the FTL language assumes an underlying nontemporal query language provided by the DBMS. However, the FTL language is not dependent on a specific underlying query language, or, in other words, can be installed on top of any DBMS. This installation is discussed in section 4.1.

The formulas (i.e. queries) of FTL use two basic future temporal operators Until and Nexttime. Other temporal operators, such as Eventually, can be expressed in terms of the basic operators. The symbols of the logic include various *type* names, such as relations, integers, etc. These denote the different types of object classes and constants in the database. We assume that, for each $n \geq 0$, we have a set of n-ary *function* symbols and a set of n-ary *relation* symbols. Each n-ary function symbol denotes a function that takes n-arguments of particular types, and returns a value. For example, + and * are function symbols denoting addition and multiplication on the integer type. Similarly, $\leq, \geq$ are binary relation symbols denoting arithmetic comparison operators. The functions symbols are also used to denote *atomic* queries, i.e. queries in the underlying nontemporal query language (e.g. OQL). We assume that all atomic queries retrieve single values. For example, the function "RETRIEVE (o.height) WHERE o.id = 100" denotes the query that retrieves the height of an object whose id is 100. Atomic queries can have variables appearing in them. For example, "RETRIEVE (o.height) WHERE o.id = y" has the variable y appearing free in it; for a given value to the variable y, it retrieves the height of the object whose id is given by y.

Functions of arity zero denote constants and relations of arity zero denote propositions.

The formulas of the logic are formed using the function and relation symbols, the object classes and variables, the logical symbols $\neg, \wedge$, the assignment quantifier $\leftarrow$, square brackets $[,]$ and the temporal modal operators Until and Nexttime. In our logic, the assignment is the only quantifier. It binds a variable to the result of a query in one of the database states of the history. One of the advantages of using this quantifier rather than the First Order Logic (FOL) quantifiers is that the problems of safety are avoided. This problem is more severe when database histories (rather than database states) are involved. Also, the full power of FOL is unnecessary for the sequence of database states in the history. The assignment quantifier allows us to capture the database atomic query values at some point in time and relate them to atomic query values at later points in time.

A *term* is a variable or the application of a function to other terms. For example, $time + 10$ is a term; if x, y are variables and f is a binary function, then $f(x, y)$ is a term; the query "RETRIEVE o.height WHERE o.id = y" specified above is also a term. Well formed formulas of the logic are defined as follows. If $t_1, ..., t_n$ are terms of appropriate type, and R is an n-ary relational symbol, then $R(t_1, ..., t_n)$ is a well formed formula. If f and g are well formed formulas, then $\neg f$, $f \wedge g$, f Until g, Nexttime f and $([x \leftarrow t]f)$ are also well formed formulas, where x is a variable and t is a term of the same type as x and may contain free variables; such a term t may represent a query on the database. A variable x appearing in a formula is *free* if it is not in the scope of an assignment quantifier of the form $[x \leftarrow t]$.

For example, the following query retrieves the pairs of objects o and n such that the distance between o and n stays within 5 miles until they both enter polygon P (the FTL formula is the argument of the WHERE clause):

RETRIEVE o,n
WHERE $DIST(o,n) \leq 5$
Until $(INSIDE(o,P)) \wedge INSIDE(n,P)$

3.3 Semantics

Intuitively, the semantics are specified in the following context. Let s_0 be the state of the database when a query f is entered. The formula f is evaluated on the history starting with s_0.

We define the formal semantics of our logic as follows. We assume that each type used in the logic is associated with a domain, and all the objects of that type take values from that domain. We assume a standard interpretation for all the function and relation symbols used in the logic. For example, $\leq$ denotes the standard less-than-or-equal-to relation, and $+$ denotes the standard addition on integers. We will define the satisfaction of a formula at a state on a history with respect to an evaluation, where an *evaluation* is a mapping that associates a value with each variable. For example, consider the formula $[x \leftarrow RETRIEVE(o)]\mathsf{Nexttime}RETRIEVE(o) \neq x$, that is satisfied when the value of some attribute of o differs in two consecutive database states. The satisfaction of the subformula $RETRIEVE(o) \neq x$ depends on the result of the atomic query that retrieves o from the current database, as well as on the value of the variable x. The value associated with x by an evaluation is the value of o in the previous database state.

The definition of the semantics proceeds inductively on the structure of the formula. If the formula contains no temporal operators and no assignment (to the variables) quantifiers, then its satisfaction at a state of the history depends exclusively on the values of the database variables in that state and on the evaluation. A formula of the form f Until g is satisfied at a state with respect to an evaluation ρ, if and only if one of the following two cases holds: either g is satisfied at that state, or there exists a future state in the history where g is satisfied and until then f continues to be satisfied. A formula of the form Nexttime f is satisfied at a state with respect to an evaluation, if and only if the formula f is satisfied at the next state of the history with respect to the same evaluation. A formula of the form $[x \leftarrow t]f$ is satisfied at a state with respect to an evaluation, if and only if the formula f is satisfied at the same state with respect to a new evaluation that assigns the value of the term t to x and keeps the values of the other variables unchanged. A formula of the form $f \wedge g$ is satisfied if and only if both f and g are satisfied at the same state; a formula of the form $\neg f$ is satisfied at a state if and only if f is not satisfied at that state.

In our formulas we use the additional propositional connectives $\vee$ (*disjunction*), $\Rightarrow$ (*logical implication*) all of which can be defined using $\neg$ and $\wedge$. We will also use the additional temporal operators Eventually and Always which are defined as follows. The temporal operator Eventually f asserts that f is satisfied at some future state, and it can be defined as $true$ Until f. Actually, in our context a more intuitive notation is often **later** f, but we will use the traditional Eventually f. The temporal operator Always f asserts that f is satisfied at all future states, including the present state, and it can be defined as $\neg$ Eventually $\neg f$. We would like to emphasize that, although the above context implies that f is evaluated at each database state, our processing algorithm avoids this overhead.

3.4 Examples

In this subsection, we show how to express some queries in FTL. For expressive convenience, we also introduce the following real-time (i.e. bounded) temporal operators. These operators can be expressed using the previously defined temporal operators and the $time$ object. (see [14]). **Eventually_within_c** (g) asserts that the formula g will be satisfied within c time units from the current position. **Eventually_after_c** (g) asserts that g holds after at least c units of time. **Always_for_c** (g) asserts that the formula holds continuously for the next c units of time. The formula (g **until_within_c** h) asserts that there exists a future instance within c units of time where h holds, and until then g continues to be satisfied. Each of our example queries have the form "RETRIEVE <target-list> WHERE <condition>". Here <condition> is given by a FTL formula.

The following query retrieves all the objects o that enter the polygon P within three units of time, and have the attribute $PRICE \leq 100$.

(I) RETRIEVE o
WHERE $o.PRICE \leq 100 \wedge$ **Eventually_within_c** $INSIDE(o,P)$

The following query retrieves all the objects o that enter the polygon P within three units of time, and stay in P for another 2 units of time.

(II) RETRIEVE o
WHERE **Eventually_within_3**
$((INSIDE(o,P) \wedge$
Always_for_2 $INSIDE(o,P))$

The following query retrieves all the objects o that enter the polygon P within three units of time, stay in P for two units of time, and after at least five units of time enter another polygon Q.

(III) RETRIEVE o
WHERE **Eventually_within_3**
[$(INSIDE(o, P) \wedge$ **Always_for_2** $(INSIDE(o, P)) \wedge$
Eventually_after_5 $INSIDE(o, Q)$]

3.5 Algorithm for evaluation of MOST queries

In this subsection, we present an algorithm for evaluating FTL queries in the MOST model. Our algorithm works for a subset of queries given by *conjunctive* formulas. A conjunctive formula is an FTL formula without negation. In practice, most queries are indeed expressed by conjunctive queries. For instance, all the example queries we use in this paper are such. The reason for the restriction to conjunctive formulas is safety (i.e. finiteness of the result); negation may introduce infinite answers. The handling of negation can be incorporated in the algorithm, but this is beyond the scope of this paper. An additional restriction of the algorithm is that it works only for continuous and instantaneous queries (i.e. not for persistent queries).

For a query CQ specified by the formula f with free variables $(x_1, ..., x_k)$ the algorithm returns a relation called $Answer(CQ)$ (this relation was originally discussed in subsection 2.2), having $k + 1$ attributes. The first k attributes give an instantiation ρ to the variables, and the last attribute gives a time interval during which the instantiation ρ satisfies the formula.

The system uses this relation to answer continuous and instantaneous queries as follows. For a continuous query CQ, the system presents to the user at each clock-tick t, the instantiations of the tuples having an interval that contains t. So, for example, if $Answer(CQ)$ consists of the tuples (2, (10,15)), and (5, (12,14)), then the system displays the object with $id = 2$ between clock ticks 10 and 15, and between clock-ticks 12 and 14 it also displays the object with $id = 5$.

For an instantaneous query, the system presents to the user the instantiations of the tuples having an interval that contains the current clock-tick.

The complete algorithm is presented in the appendix.

4 Indexing dynamic attributes

In this section we address the issue of indexing dynamic attributes. The objective is to enable answering queries of the form "Retrieve the objects that are currently in the polygon P" without examining all the objects. The problem with a straight-forward use of spatial indexing is that since objects are continuously moving, the spatial index has to be continuously updated, an unacceptable solution.

We introduce a possible method of indexing dynamic attributes. Due to space limitations we only outline the main ideas in the method. The method plots all the functions representing the way a dynamic attribute A changes with time. Thus, the x-axis represents time, and the y-axis represents the value of A. For the sake of simplicity we assume that the functions are linear. However, the ideas can be extended to nonlinear functions.

This method is adapted from [5], although the model there is different. We use a spatial index (see [9] for a survey of spatial access indexes) for each dynamic attribute A. Spatial indexes use a hierarchical recursive decomposition of space, usually into rectangles; the id of each object o is stored in the records of representing the rectangles crossed by the $A.function$ of o.

Suppose now that the instantaneous query "Retrieve the objects for which currently $4 < A < 5$ " is entered at time 1:00am. Then, using the index we retrieve the records representing the rectangles that intersect the rectangle $4 < A < 5$ and $1 - \epsilon < t < 1 + \epsilon$. For each object id in these records we check whether "currently" $4 < A < 5$.

An update of $o.A$ at time t involves updating the records representing rectangles ending after time t; o is removed from the records representing rectangles crossed by the old function-line, and it is added to the records representing rectangles crossed by the new function-line.

For an object moving in 2-dimensional space, the above scheme can be mimicked using an index of 3-dimensional space, with the third dimension being, obviously, time.

Observe that spatial indexing is limited to finite space. Thus, in order to use this scheme we have to consider the time dimension starting at 0 and ending at some time-point T. Consequently, the index needs to be reconstructed every T time units. Choosing an appropriate value for T is an important future-research question.

Now suppose that the query "Retrieve the objects for which currently $4 < A < 5$ " is entered at time 1:00am as a continuous query, CQ. Then, using the index we retrieve the records representing the rectangles that intersect the rectangle $4 < A < 5$ and $1 < t < T$. We construct the set $Answer(CQ)$ by examining each object id o in these records, and determining the time intervals when $4 < o.A < 5$.

5 Discussion

In this section we first discuss the implementation of our proposed data model on top of existing DBMS's (subsection 5.1), then we discuss architectural issues, particularly the implications of disconnection and memory limitations of computers on moving objects (5.2), and various query processing strategies in a mobile distributed system (5.3).

5.1 Implementing MOST on top of a DBMS

Our system proposed in this paper (including an FTL language interpreter) can be implemented by a software system, called MOST, built on top of an existing DBMS. Such a system can add the capabilities discussed in this paper to the DBMS as follows. We store each dynamic attribute A as three DBMS attributes $A.value$, $A.updatetime$, and $A.function$. Any query posed to the DBMS is first examined (and possibly modified) by the

MOST system, and so is the answer of the DBMS before it is returned to the user. In the rest of this subsection we sketch the modifications to queries and answers of the underlying DBMS. For simplicity our exposition will assume the relational model and SQL for the underlying DBMS. However, the same ideas can be extended to object-oriented model.

If the query does not contain a reference to a dynamic attribute nor does it contain temporal operators, the query is simply passed to the DBMS and the answer returned to the user.

Now assume that the query contains references to dynamic attributes, but not temporal operators. We will distinguish between references in the SELECT and WHERE clauses. If the query contains a reference to a dynamic attribute A only in the SELECT clause (i.e. in the target list), then the MOST system modifies the query as follows. Instead of A, the query retrieves the attributes $A.value$, $A.updatetime$, and $A.function$ from the DBMS; and the MOST system computes the value of A for each retrieved object before returning it to the user.

Assume now that the WHERE clause is F, which is a boolean combination of atoms (for example, an atom may be $A > 5$). Consider first the case where there is only a single atom p that refers to dynamic attributes in F. Before passing the original query Q to the DBMS the MOST system replaces Q by two queries, Q_1 and Q_2. The transformation is based on the following equivalence. $F = (F' \wedge p) \vee (F'' \wedge \neg p)$, where F' is F with p replaced by true and F'' is F with p replaced by false. Q_1 and Q_2 are defined as follows. The target list of Q_1 and Q_2 consists of the target list of Q, plus the subattributes of the dynamic attributes in p. The FROM clause of Q_1 and Q_2 is identical to that of Q. The WHERE clause of Q_1 is F' and that of Q_2 is F''. Q_1 and Q_2 are submitted to the underlying DBMS, and the results are processed as follows before returning them to the user. The atom p is evaluated on each tuple in the result of Q_1, and the atom $\neg p$ is is evaluated on each tuple in the result of Q_2. (To do these evaluations the MOST system computes the current values of the dynamic attributes appearing in p using the retrieved sub-attributes.) The tuples that do not satisfy the respective atoms are eliminated, and the projection of the union of the resulting tuples on the original target list is returned to the user. If the WHERE clause has multiple atoms referencing dynamic attributes then we can give a function $EVAL(Q)$ that performs the above procedure recursively, each time eliminating one of the atoms containing a dynamic variable. If the original query has k atoms refering to a dynamic variable then, in the worst case, this might mean evaluating upto 2^k queries that do not contain dynamic variables. However, if k is small this may not be a serious problem.

Observe that the above procedure does not use indexing of the dynamic attributes. In other words, the results of Q_1 and Q_2 are examined in their entirety. If indexing on the dynamic attributes is available, then we can modify the above procedure as follows. Instead of evaluating the atoms p and $\neg p$ on each tuple retrieved by Q_1 and Q_2 respectively, we retrieve the tuples that satisfy p and $\neg p$ respectively. Then we join the relation returned by Q_1 with the relation that satisfies p; similarly, we join the relation returned by Q_2 with the relation that satisfies $\neg p$. Observe that in order for this procedure to produce correct results, we must ensure that F' and p are satisfied for the same tuple in the cartesian product of the FROM relations. We ensure this by including in the target list of all four queries, a key of each relation in the FROM clause. The above method can extended to nested SQL queries as well.

Now consider temporal operators. Note that the procedure in the appendix given for processing FTL formulas can be modified to take advantage of the query processing capabilities of the DBMS. This is done as follows. In the given FTL formula f, we identify the maximal non-temporal subformulas. Let g be any such subformula. Note that g may contain some free variables. As given in the appendix, corresponding to g we compute a relation G that contains the set of all evaluations to the free variables in g that satisfy g. We compute this relation G by using the decomposition method for non-temporal queries described above. All the relations computed in this fashion are combined using the procedure in the appendix, according to the structure of the formula f.

5.2 Continuous queries from moving objects

Consider a centralized DBMS equipped with the MOST capability. Suppose that a continuous query CQ is issued from a moving object M. M may or may not be one of the objects represented in the database. After the centralized DBMS computes the set $Answer(CQ)$, there are two approaches of transmitting it to M, immediate and delayed.

In the *immediate* approach, the whole set is transmitted immediately after being computed. For each tuple $(S, begin, end)$, the computer in M is presenting S between times $begin$ and end. However, remember that explicit updates of the database may result in changes to $Answer(CQ)$. If so, the relevant changes are transmitted to M.

The immediate approach may have to be adjusted, depending on the memory limitations at M. For example, M's memory may fit only B tuples, and the set $Answer(CQ)$ may be larger. In this case, the set $Answer(CQ)$ needs to be sorted by the $begin$ attribute, and transmitted in blocks of B tuples.

The *delayed* approach of transmitting the set $Answer(CQ)$ to M is the following. Each tuple $(S, begin, end)$ in the set is transmitted to M at time $begin$. The computer at M immediately displays S, and keeps it on display until time end.

Of course, intermediate approaches, in which subsets of $Answer(CQ)$ are transmitted to M periodically, are possible.

The choice between the immediate and delayed approaches depends on several factors. First, it depends on the probability that an update to $Answer(CQ)$ can be propagated to M (i.e. that M is not disconnected) before the effects of the update need to be displayed. Second, it depends on the frequency of updates to $Answer(CQ)$, and the cost of propagating these updates to M.

5.3 Distributed query processing

Assume now that each object represented in the database is equipped with a computer, and the database is distributed among the moving objects. In particular, assume that the distribution is such that each object resides in the computer on the moving vehicle it represents, but nowhere else. This is a reasonable architecture in case there are very frequent updates to the attributes of the moving object. For example, if the motion vector of the object changes frequently, then these changes may only be recorded at the moving object itself, rather than transmitting each change to other moving objects or to a centralized database.

Assume that each query is issued at some moving object. We distinguish between three types of MOST queries. The first, called *self-referencing query*, is a predicate whose truth value can be determined by examining only the attributes of the object issuing the query. For example, “Will I reach the point (a,b) in 3 minutes” or, “When will I reach the point (a,b)” are self-referencing queries. Clearly, self-referencing queries can be answered without any inter-computer communication.

The second type of queries, called *object queries*, is a predicate whose truth value can be determined for an object independently of other objects. For example, “Retrieve the objects that will reach the point (a,b) in 3 minutes” is an object query; for each object we can determine whether or not it satisfies the predicate, independently of other objects. To answer an object query, a mobile computer needs to be able to communicate with the other mobile computers. Assuming this capability, there are two ways to processing such a query issued from mobile object M. First is to request that the object of each mobile computer be sent to M; then M processes the query. Second is to send the query to all the other mobile computers; each computer C for which the predicate is satisfied sends the object C to M. The second approach is more efficient since it processes the query in parallel, at all the mobile computers. The second approach is also more efficient for continuous queries. In this case, the remote computer C evaluates the predicate each time the object C changes, and transmits C to M when the predicate is satisfied. Using the first approach C would have to transmit C to M every time the object C changes.

The third type of query, called *relationship query*, is a predicate whose truth value can only be determined given two or more objects. For example, the query “Retrieve the objects that will stay within 2 miles of each other for at lease the next 3 minutes” is a relationship query. The most efficient way to answer a relationship query is to send all the objects to a central location. The most natural location is the computer issuing the query. When a relationship query is presented at mobile computer M, it requests the objects from all other mobile computers. Then M processes the query.

6 Comparison to relevant work

One area of research that is relevant to the model and language presented in this paper is temporal databases [12, 10, 11]. The main difference between our approach and the temporal database works is that, by and large, those works assume that the database varies at discrete points in time, and between updates the values of database attributes are constant ([12] uses interpolation functions to some extent). In contrast, here we assume that dynamic attributes change continuously. Also, temporal languages other than FTL can be used to query MOST databases, but any other processing algorithm will have to be modified to handle dynamic attributes.

Another relevant area is constraint databases (see [6] for a survey). In this sense, our dynamic attributes can be viewed as a constraint, or a generalized tuple, such that the tuples satisfying the constraint are considered in the database. Constraint databases have been separately applied to the temporal (see [3, 4, 1]) domain, and to the spatial domain (see [7]). However, the integrated application for the purpose of modeling moving objects has not been considered. Furthermore, this integrated application has not been considered since the model is different than ours, thus perhaps inappropriate for modeling moving objects. The main difference is that in constraint databases **all** the tuples (or objects) that satisfy the constraint (in our case the values of the function at all time-points) are considered to be in the database simultaneously. In contrast, in our model these values are not in the database at the same time; at any point in time a different value is in the database.

Methods in object oriented systems are also relevant to our model. In an object-oriented system, the value of a dynamic attribute may be computed by a method (i.e. a program stored with the data) using the sub-attributes of a dynamic attribute. However, in this case, as far as the DBMS is concerned the method is a black-box, and the only way to answer a query such as ”retrieve the objects that will intersect a polygon P at some time between now and 5pm” is to evaluate the query at every point in time between now and 5pm. In contrast, in our model we “open” the black box, i.e. expose to the DBMS the way the dynamic attribute changes. Thus the DBMS can currently compute which objects will intersect the polygon in the future. Additionally, the object oriented approach is not able to utilize indexing on dynamic variables.

Another body of relevant work is location-dependent software systems (e.g. [13, 15, 3]). There are three differences between that work and the our work presented in this paper. First, although independent of a *particular* database management system our work pertains to incorporation of mobility in database systems. Second, our work pertains to situations where the mobile clients are aware not only of their current location, but also of their movement, i.e. their future location. Indeed for airplanes and cars moving on the highway, this is often the case. Third, in our model the answer to a query depends not only on the location of the client posing the query, but also on the time at which the query is posed.

In our earlier work ([14]) we introduced FTL for specifying trigger conditions in active databases. The algorithm presented there does not work in the MOST model, since it can only deal with static attributes.

7 Conclusion and future work

In this paper we introduced the the MOST data model for representing moving objects. It has two main aspects. First is the novel notion of dynamic attributes, i.e. attributes that change continuously as time passes without being explicitly updated. They are represented by functions of time. Therefore a user can query future states of database values. This motivates the second aspect of our data model, namely the query language, FTL. It enables the specification of future queries, i.e. queries that refer to future states of the database.

In support of the new data model, in this paper we developed algorithms for processing queries specified in FTL, we discussed a method of indexing dynamic attributes, and we discussed methods for building the capabilities of MOST on top of existing database management systems. We also identified several types of queries arising in the new data model, namely instantaneous, continuous and persistent queries. We also discussed issues of query processing in a mobile and distributed environment.

In the future, we intend implement the MOST data model on top of an existing DBMS, e.g. Sybase. We intend to further explore various processing methods for the three types of queries, particularly in mobile and distributed environments. We intend to experimentally compare various mechanisms for indexing dynamic attributes.

8 Appendix: The FTL query processing algorithm

Let $f(x_1, x_2, ..., x_k)$ be a query with free variables $x_1, x_2, ..., x_k$ respectively. We assume that the system has a set of objects O. Some of these objects are stationary and the others are mobile. The positions (i.e. the X, Y and Z coordinates) of the stationary objects are assumed to be fixed, while the positions of the mobile objects are assumed to be dynamic variables. We also assume that we have a dynamic database variable *time* whose value at any instance gives the actual value of the time at that instance. Without loss of generality we assume that the time when we are evaluating the query is zero. The current database state reflects the positions of objects as of this time, and furthermore, we assume that for each dynamic variable we have functions denoting how these variables change over time. As a consequence, the values of static variables at any time is the same as their value at time zero, and the values of dynamic variables at any time in the future are given by the functions which are stored in the database. Thus, the future history of the database is implicitly defined.

For each subformula g of f (including f itself), our algorithm computes a relation R_g. Let $g(x_1, ..., x_l)$ be a subformula containing free variables $x_1, ..., x_l$. The relation R_g will have $(l+1)$ attributes, the first l attributes correspond to the l variables, and the last attribute denotes a time interval. Each tuple in R_g denotes an instantiation ρ of values to the free variables in g and an interval I during which the formula g is satisfied with respect to ρ. The intervals corresponding to different tuples that give identical values to the corresponding variables will be non-overlapping, and furthermore these intervals will not even be consecutive; the non-consecutiveness of the intervals means that there is a non-zero gap separating intervals in tuples that give identical values to corresponding variables; if the relation R_g does not satisfy this non-consecutiveness property then we can modify it to satisfy this property as follows; we simply replace all tuples that give same values to the corresponding variables and whose intervals are consecutive with a single tuple whose interval is the union of the intervals of the corresponding intervals.

The algorithm computes R_g, inductively, for each subformula g in increasing lengths of the subformula. After the termination of the algorithm, we will have the relation R_f corresponding to the original formula f.

The base case in our algorithm is when g is an atomic predicate $R(x_1, ..., x_l)$ such as a spatial relation etc. In this case, we assume that there is a routine, which for each possible relevant instantiation of values to the free variables in g, gives us the intervals during which the relation R is satisfied. Clearly, this algorithm has to use the initial positions and functions according to which the dynamic variables change. For example, if R is the predicate $DIST(x_1, x_2) \leq 5$, then the algorithm gives, for each relevant object pair o_1, o_2, the time intervals during which the distance between them is ≤ 5. We assume that the relation given by the atomic predicates are all finite. For cases where these relations are infinite in size, we need to use some finite representations for them and work with these representations; this is beyond the scope of this paper and will be discussed in a later paper.

For the case when g is not an atomic predicate, we compute the relation R_g inductively based on the outer most connective of g as given below.

Let $g = g_1 \wedge g_2$. In this case, let R_1, R_2 be the relations computed for g_1 and g_2 respectively, i.e $R_i = R_{g_i}$ for $i = 1, 2$. For a given instantiation ρ, if g_1 is satisfied during interval I_1 and g_2 is satisfied during I_2 then g is satisfied during the interval $I_1 \cap I_2$. The relation R for g is computed by joining the relationships R_1 and R_2 as follows: the join condition is that common variable attributes should be equal and the interval attributes should intersect; the retrieved tuple copies all the variable values, and the interval in the tuple will be the intersection of the of the intervals of the joining tuples.

Let $g = g_1$ Until g_2, and let R_1 and R_2 be the relations corresponding to g_1 and g_2 respectively. Let $p+1, q+1$ be the number of columns in R_1 and R_2 respectively. Let t_1 and t_2 be any pair of tuples in R_1 and R_2 respectively, such that they match on columns corresponding to the same variables. Let T_1 be the set of tuples in R_1 having the same values as t_1 in the first p columns; and similarly let T_2 be the set of tuples in R_2 having the same values as t_2 in its first q columns. Let I_1 and I_2 be the set of all intervals appearing in T_1 and T_2, respectively. Note that no two intervals in I_1 are consecutive, i.e. it is not the case that one of them starts immediately after the other. Similarly, no two intervals in I_2 are consecutive. We say that an interval $[l_1\ u_1]$ is *compatible* with another interval $[m_1\ n_1]$ if $m_1 \leq (u_1+1)$ and $n_1 \geq u_1$, i.e the two intervals either overlap or they are consecutive. Suppose we have an interval $[l_1\ u_1]$ in I_1 which is compatible with an interval $[m_1\ n_1]$ in I_2. Then, it should be easy to see that the formula g is satisfied throughout the interval $[l_1\ n_1]$ with respect to the instantiation of variables given by the tuples t_1 and t_2. A chain s is a sequence of intervals $[l_1 u_1], [m_1 n_1], , [l_2 u_2], [m_2 n_2], ..., [l_k u_k], [m_k n_k]$ such that for each $i = 1, ..., k$, $[l_i\ u_i]$ is an interval in I_1 and $[m_i\ n_i]$ is an interval in I_2, and the interval $[l_i\ u_i]$ is compatible with the interval $[m_i\ n_i]$, and for $i < k$ the interval $[m_i\ n_i]$ is compatible with the interval $[l_{i+1}\ u_{i+1}]$. For the chain s as given above, let $interval(s)$ denote the interval $[l_1\ n_k]$. It is easy to see that the formula g is satisfied throughout $interval(s)$ with respect to the instantiations of the variables as given by the tuples t_1 and t_2.

We say that a chain s is *maximal* if it is not a subsequence of any other chain. All the maximal chains can be computed by sorting the sets I_1 and I_2 individually and running a modified merge algorithm. Let S be the set of all proper, maximal sequences. Let A be the union of all column names in R_1 and R_2 that correspond to variables. The relation R for g will contain $|A| + 1$ columns. For each $s \in S$, the relation R will contain a tuple whose first $|A|$ columns contain the corresponding values from t_1 or t_2, and whose $(|A| + 1)st$ column will contain the value $interval(s)$. For every pair of joining tuples t_1 in R_1 and t_2 in R_2, the relation R for g will have the above tuples. In the worst case, this algorithm may run in time proportional to the product of the sizes of R_1 and R_2 respectively.

Let $g = [y \leftarrow q]\ g_1$, and let R_1 be the relation corresponding to g_1. The atomic query q may have some free variables. For example, q may be $height(o)$ denoting the height attribute of the object given by the variable o. We assume that the value of q is given by a relation Q with $p+2$ where the first p columns correspond to the free variables in q, the $(p+1)st$ column is the value of q and the last column is a time interval. Each tuple t in Q denotes the value of the atomic query q during the interval specified by the last column, and for the the instantiation of free variables specified by the first p columns; the value of the query is given by the $p + 1$st column. In above example, Q will have three columns; the first column gives the object id, the third column gives an interval and the second column gives the height of the object during this interval. Now the relation R for g is obtained by joining Q and R_1 where the join condition requires that columns corresponding to common variables should be equal, the column corresponding to the y variable in R_1 should be equal to the $(p+1)$st column of Q, and the time intervals should intersect. For two joining tuples t_1 in R_1 and t_2 in Q, in the output tuple we copy all variable columns from t_1 and t_2 excepting the one corresponding to variable y, and the time interval in the output tuple will be the intersection of the time intervals in t_1 and t_2.

References

[1] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, Aug. 1989.

[2] S. Acharya, B. Badrinath, T. Imielinski, and J. Navas. A www-based location dependent information service for mobile clients. *Rutgers Univ. TR*, July 1995.

[3] M. Baudinet, M. Niezette, and P. Wolper. On the representation of infinite data and queries. *ACM Symposium on Principles of Database Systems*, May 1991.

[4] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. *ACM Symposium on Principles of Database Systems*, March 1988.

[5] G. Hjaltason and H. Samet. An indexing scheme for time dependent data in clinsys. *unpublished manuscript*.

[6] P. Kanellakis. Constraint programming and database languages. *ACM Symposium on Principles of Database Systems*, May 1995.

[7] J. Paradaens, J. van den Bussche, and D. V. Gucht. Towards a theory of spatial database queries. *ACM Symposium on Principles of Database Systems*, 1994.

[8] R. Snodgrass and I. Ahn. The temporal databases. *IEEE Computer*, Sept. 1986.

[9] H. Samet. *The design and analysis of spatial data structures*. Addison Wesley, 1990.

[10] R. Snodgrass. The temporal query language tquel. *ACM Trans. on Database Systems*, 12(2), June 1987.

[11] R. Snodgrass and ed. Special issue on temporal databases. *Data Engineering*, Dec. 1988.

[12] A. Segev and A. Shoshani. Logical modeling of temporal data. *Proc. of the ACM-Sigmod International Conf. on Management of Data*, 1987.

[13] B. Schilit, M. Theimer, and B. Welch. Customizing mobile applications. *USENIX Symposium on Location Independent Computing*, Aug. 1993.

[14] P. Sistla and O. Wolfson. Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(3), June 1995.

[15] G. Voelker and B. Bershad. Mobisaic: An information system for a mobile wireless computing environment. *Workshop on Mobile Computing Systems and Applications*, 1994.

[16] J. Clifford and T. Isakowitz. On the Semantics of Temporal Variable Databases. *Proceedings of Fourth International Conference on Extending Database Technology, Cambridge, England*, 1994