Querying the Uncertain Position of Moving Objects

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

Abstract

In this paper we propose a data model for representing moving objects with uncertain positions 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frequently (which would impose a serious performance and wireless-bandwidth overhead), or, the answer to queries is outdated.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.

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. Our model also allows for uncertainty in the values of dynamic attributes. 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 [12] 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 [12] 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.

*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

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 provide two different kinds of semantics— called *may* and *must* semantics respectively. Because of possible uncertainty in the values of dynamic attributes, these two different semantics may produce different results. For example, consider the query "Retrieve all air-planes with in 5 miles of airplane A". Under the "may" semantics, the answer is the set of all airplanes that are possibly with in 5 miles of A. Under the "must" semantics, this will be the set of all airplanes which are definitely with in 5 miles of A. These two values coincide if there is no uncertainty in the position of A.

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".

We consider the problem of evaluating "may" queries for the case of moving objects. We show that, in general, this evaluation problem is computationally hard for the case when there is uncertainity and the objects are moving freely in two dimensional space. For the case when the objects are moving on well defined routes and when there is uncertainty in their speeds, we show that the evaluation problem of "may" queries, that use restricted FTL formulas, is efficiently solvable; we do this by reducing this problem to the evaluation problem for the deterministic case, i.e. when there is no certainty. Finally, we give an efficient algorithm for evaluating queries in the deterministic case (for relational databases, this method translates the FTL formula in to a sequence of SQL queries).

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. The principles for incorporating dynamic attributes on top of existing DBMS's are outlined.
- Adaptation of FTL as a query language in MOST. Two different semantics for queries, called "may" and "must" semantics, are provided.
- An efficient algorithm is devised for processing queries specified in an important subclass of FTL when all dynamic variables are deterministic, i.e. when there is no uncertainty in their values at any instant of time.
- Efficient algorithm for processing an important subclass of "may" queries when objects are traveling on well defined routes and when there is uncertainty in their speeds. This algorithm is obtained by reducing it to the deterministic case.

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

- We show how the proposed algorithms can be implemented on top of an existing DBMS.

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.

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 attribute POSITION denoting the position of the object. Depending on the coordinate system one might be using, the POSITION attribute may have sub-attributes POSITION.X, POSITION.Y and POSITION.Z denoting the x-,y- and z-coordinates of the object. 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 ann 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 an object starting from the origin and moving along the x axis in two dimensional space at 50 to 60 miles per unit time. Then its position at any point in time is given by its x and y coordinates, where the value of its y-coordinate is a static attribute and has valuezero, and the value of its x-coordinate is a dynamic attribute that changes with time. Its x-coordinate has value 1 some time in the time interval between $\frac{1}{60}$ and $\frac{1}{50}$ units of time, has value 2 some time in the interval between $\frac{2}{60}$ and $\frac{2}{50}$, and so on (equivalently, after t units of time, the x-coordinate value of the object's position is somewhere between $50t$ and $60t$). We represent the values of a dynamic attribute at various times by a sequence of value-time pairs of the form $(u_1, t_1), ..., (u_i, t_i), (u_{i+1}, t_{i+1}), ...(u_n, t_n)$ where $t_1 < t_2 < ... < t_i < t_{i+1}$.... Such a sequence indicates that, for each i such that $0 \leq i < n$, the attribute has value u_i during the right open interval $[t_i, t_{i+1})$, and has value u_n at time t_n. Usually, $t_1, ..., t_n$ denote the time instances when the attribute value changes implicitly. Note that, here we are assuming that the attribute takes discrete values.

Formally, a *dynamic attribute* A is represented by three sub-attributes, *A.initialvalue*, *A.updatetime*, and *A.function*(denoted as $A.f$). Here $A.f$ is a multi-valued function which takes as argument a sequence of value-time pairs and returns a set of possible value-time pairs of the dynamic attribute. Intuitively, the returned value-time pairs denote the different ways in which the input sequence can be extended to denote values of the variable in future. At time *A.updatetime* the value of A is given by *A.initialvalue* and its value until the next update is defined inductively as follows. Let

$v = (u_1, t_1), ..., (u_n, t_n)$ be a sequence of value-time pairs denoting the values of A up to time t_n, where $u_1 = A.initialvalue$ and $t_1 = A.updatetime$, and let (u_{n+1}, t_{n+1}) be any element in the set C returned by $A.f$ on input v. Then, the sequence $(u_1, t_1)...., (u_n, t_n), (u_{n+1}, t_{n+1})$ denotes a possible sequence of values of A up to time t_{n+1}. Note that by extending v with different elements from C we get different extensions; these different extensions may give different values to A at the same time denoting the uncertainty in its value.

For any $t \geq A.updatetime$, we define the set of *possible values* of A at time t as follows. Let $(u_1, t_1), ..., (u_n, t_n), (u_{n+1}, t_{n+1})$ be a sequence of value-time pairs generated inductively as defined above such that $t_n \leq t < t_{n+1}$. Then u_n is a possible value of A at time t. The set of all possible values of A at time t is defined by considering all possible such sequences. Thus, we see that the possible values at a future time is defined inductively as function of its values at previous times. The need for such a definition will be be clear when we define database histories in the next subsection. Also, it is to be noted that the uncertainty in the possible value of a dynamic attribute is indicated by the fact that $A.f$ returns a set of values.

As an example, let $o.POSITION.X$ be the dynamic variable denoting the x-coordinate of an object o moving in the direction of the x-axis at a speed ranging between the values l and h. Let u be a sequence of value-time pairs ending with (u_n, t_n). Then the function $o.POSITION.X.f$, on input u, will output the set of values $\{(u_n + 1, t_n + \epsilon) : \frac{1}{h} \leq \epsilon \leq \frac{1}{l}\}$ denoting that the x-coordinate increases by 1 at any time between $\frac{1}{h}$ and $\frac{1}{l}$ time units after t_n.

A dynamic variable A is called *deterministic* if the set of values returned by $A.f$ is a singleton set. An explicit update of a dynamic attribute may change any of its sub-attributes, except for $A.updatetime$.

There are two possible interpretations of $A.updatetime$, corresponding to valid-time and transaction-time (see [14]). 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 gives the range of possible values 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 possible current x-positions of object o" depends on the value of the dynamic attribute $o.POSITION.X$ 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 may 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.

We assume that there is a special variable called *time_stamp*. A *database state* is a mapping that associates a set of objects of the appropriate type to each object class and a time value to the *time_stamp* variable. The value of the *time_stamp* variable in a database state gives the time when that database state was created, i.e. the update that created the database state. In any database state, the value of a dynamic attribute A is given by the values of its three sub-attributes $A.initialvalue$, $A.updatetime$ and $A.f$. For any object o, we let $o.A$ denote the attribute A of o; if A has a sub-attribute B then we let $o.A.B$ denote the value of the sub-attribute. We denote the value of the attribute A of an object o in a state s by $s(o.A)$.

Let s be a database state, and t be any time value greater than or equal to $s(time\_stamp)$. A *possible database state* s' corresponding to s at time t is a mapping that associates a set of objects of the appropriate type to each object class and the value t with the variable $time\_stamp$ satisfying the following properties: for each object class C the set of objects assigned by s' to C is same as the set of objects assigned by s; for each object o present in s the value of an attribute A of o in s' is defined as follows; if A is a static attribute then $s'(o.A) = s(o.A)$; if A is a dynamic attribute then s' treats A as atomic and assigns it a value $s'(o.A)$ which belongs to the set of possible values of A as defined in the previous subsection. For a database state s, and any time $t \geq s(time\_stamp)$, there can be more than one possible database state corresponding to s at time t. However, if all the dynamic attributes are deterministic, i.e. no uncertainty, then there can only be one possible database state corresponding to s at time t.

A *trace* is a finite sequence $s_0, .., s_i, ..., s_n$ of database states such that for every $i > 0$, $s_i(time\_stamp) > s_{i-1}(time\_stamp)$, i.e. the values of the $time\_stamp$ are strictly increasing. For any $i > 0$, we say that the attribute A of object o is updated in the state s_i if o is present in both s_i and s_{i-1} and $s_{i-1}(o.A) \neq s_i(o.A)$. We say that object o is created in state s_i if o is present in s_i but not in s_{i-1}. If o is created in s_i then for every dynamic attribute A of o, $s_i(o.A.updatetime) = s_i(time\_stamp)$. Similarly, if a dynamic attribute A of an object o is updated in state s_i, then $s_i(o.A.updatetime) = s_i(time\_stamp)$.

Let $\sigma = (s_0, ..., s_i, ...s_n)$ be a trace. A *possible database history* h (briefly, a *database history*) corresponding to σ is an infinite sequence $v_0, v_1, ..., v_j, ...$ of possible database states such that the following properties are satisfied: (i) for all $j > 0$, $v_j(time\_stamp) > v_{j-1}(time\_stamp)$, and $v_j(time\_stamp)$ increases unboundedly with j; (ii) for every i such that $0 \leq i \leq n$, there exists an k such that $s_i(time\_stamp) = v_k(time\_stamp)$; (iii) for each $j \geq 0$, v_j is a possible database state corresponding to s_i where i is the maximum integer such that $0 \leq i \leq n$ and $s_i(time\_stamp) \leq v_j(time\_stamp)$.

Note that, in a history, the value of $time\_stamp$ is monotonically increasing; usually, we assume that these time values denote the instances when the state changes either due to an explicit update, or due to implicit change in the value of a dynamic attribute.

There can be many possible database histories corresponding to a trace. However, if all the dynamic variables are deterministic then there is only one history corresponding to a trace.

Consider the example of an object o moving in the x-direction with a speed ranging between 50 and 60 miles per unit time starting with the x-coordinate equals 0 at time 0. Assume that its x-coordinate is represented by the dynamic attribute $o.POSITION.X$. We assume that the x and y coordinates are integer values. The database trace corresponding to this example has only one element consisting of the initial state. Now consider any sequence of possible database states $v_0, v_1, ..., v_i, ...$ Then $v_0(o.POSITION.X) = 0$, and for any $i > 0$, $v_i(o.POSITION.X) = i$ and $v_{i-1}(time\_stamp) + \epsilon$ where ϵ is any value between $\frac{1}{60}$ and $\frac{1}{50}$. Here the lowest value and highest values of ϵ correspond to the cases when the object moves at maximum and minimum speeds respectively.

Consider a database trace denoting the various updates on a given database up to the current time t. Let h be a database history corresponding to this trace. The finite sequence consisting of possible database states in h with a lower time-stamp than t is called the *past database-history*, and the inifinite sequence consisting of all possible states in h with a time-stamp higher than the current time t is called the *future database-history*. Each state in the future history is identical to the state at time t, except possibly for the values of the dynamic attributes.

We would like to emphasize at this point that the database history is an abstract concept, introduced solely for providing formal semantics to our temporal query language, FTL. We do not maintain store the database history any where.

2.3 Object Positions

As indicated earlier, we assume that we have a database denoting information about spatial and other objects. The class of spatial objects has a subclass of moving objects. There may be other subclasses of spatial objects such as polygons etc. All the moving objects are assumed to be point objects, and they have a dynamic attribute called POSITION. If a moving object is moving in 2-dimensional space then we assume that it has sub-attributes POSITION.X and POSITION.Y each of which itself is a dynamic attribute (similarly, objects moving in 3-dimensional space will have three sub-attributes). On the other hand, objects may be moving on well defined routes (such as high ways etc.) and in that case, the position of the object is given by its distance when measured from a fixed point on the route in a particular direction; this distance will be considered as a dynamic attribute.

Although we have used a general multi-valued function to represent the values of a dynamic attribute, one can use one of the following two schemes for representing positions of moving objects. In the first scheme, for an object moving on a well defined route, we specify the motion of the object by two numbers denoting the upper and lower bounds on the speed; for an object moving freely in the two dimensional space, its motion is specified by giving the speed bounds in the X and Y directions. In the second scheme, for an object moving on a route, we specify its distance, from the initial position, by two functions of time that give upper and lower bounds on the distance at any future time; for an object moving freely in two dimensional space, we use pairs of functions for each of the X,Y directions.

2.4 Three types of MOST queries

A *query* is a function which takes as input a database trace and a time value, and outputs a set of values. In our query language, the user can use temporal operators and can refer to the current as well as future possible database states. We define the semantics of a query by referring to the possible histories of the database. We define two different kinds of semantics of a query, called *may* and *must* semantics. In our model, we 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 histories on which the query is evaluated, and on the time when they are evaluated (in contrast, in traditional databases the situation is simpler). For each of these types of queries, we may use either of the two semantics. Which of the semantics to be used can be explicitly specified by the user or the query processor may retrieve answers under the both the semantics and output both the answers. An instantaneous query is a function of the set of current possible database states, and a continuous query is an instantaneous query evaluated continuously at each instance in the future.

Formally, the value of an *instantaneous* query at time t is defined using the set of possible histories starting at t, i.e. the time when the query is entered. As indicated earlier, the value depends upon the kind of semantics used, may or must semantics. t is usually the time when the query is entered. For example, the query $Q =$ "Display the motels within 5 miles of all the current possible positions of vehicle x", when considered as an instantaneous query returns a set of motels, presented to the user immediately after the query is evaluated. Since there may be an uncertainty in the current position, the set of motels returned depends upon the kind of semantics used. Under the "may" semantics the result is the set of motels with in 5 miles of any possible current position. Under the "must" semantics the result is the set of motels which are with in 5 miles of every possible current position.

Observe that an instantaneous query may refer to all possible future histories. For example, "Display the motels that I reach within 3 minutes" refers to all the histories, and within each history it refers to states with a time-stamp between now and three minutes later. Under "may" semantics it will output the set of of motels reached in three minutes in any of the possible future histories; under "must" semantics it will output the set of motels that will be reached in three minutes in every possible future history.

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 an instantaneous query pertains to a predefined (but very large) fixed 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 speed of the object 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 defined as a sequence of instantaneous queries at each future time $t' \geq t$, where the instantaneous query at t' has two arguments (i) the database trace as of t' and (ii) the time value t; note that the semantics of this instantaneous query is defined using the possible histories with respect to the database trace 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 possible histories. These histories may differ for the 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 POSITION.X 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.

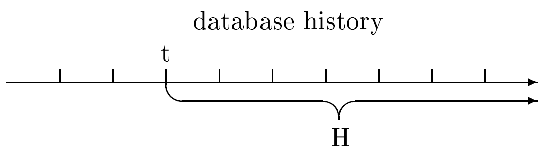

Figure 1: database history
(a) An *instantaneous* query at time t is defined with respect to the set of possible future histories H_t (i.e. the future history beginning at t).
(b) A *continuous* query at time t is a sequence of instantaneous queries at each time $t' \geq t$.
(c) A *persistent* query at time t is a sequence of instantaneous queries, all at time t. The queries are evaluated at each time $t' \geq t$ when 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 [12] 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 $[x \leftarrow t]$.

In our system, a query is specified by the following syntax:
RETRIEVE <target-list> WHERE <semantic-spec> <condition>.
Here <condition> is an FTL formula in which all the free variables are object variables. The specification <target-list> is a list of attributes of all object variables appearing free in the condition part. The clause <semantic-spec> can be one of the two key words **may** or **must**, and it specifies the semantics to be used in processing the query. We call a query to be a "may" query if its semantic clause is the key word "may", otherwise the query is called a "must" query.

For example, the following query retrieves the pairs of objects o and n such that, on all future histories, 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) in all possible future histories:

RETRIEVE o,n
WHERE **must** $DIST(o,n) \leq 5$
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 $[x \leftarrow RETRIEVE(o)]$ 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.

Let Q be an instantaneous query specified at time t using the syntax given at the end of the last subsection. Let the FTL formula f denote the condition part of Q, and let T denote the target list of Q. We define the semantics based on the ¡semantic-spec¿ clause in Q. Let σ be the database trace denoting the sequence of updates up to t. Let H be the set of all possible future database histories corresponding to σ as of now, i.e. as of time t. For any $h \in H$, let F_h be the set of all evaluations ρ to the free variables in f such that f is satisfied at the beginning of h with respect to the evaluation ρ. Let R_h denote the set of all tuples t obtained by applying some evaluation in F_h to the target list T, i.e. $R_h = \{\rho(T) : \rho \in F_h\}$. Let $May\_Answer(Q) = \bigcup_{h \in F_h} R_h$ and $Must\_Answer(Q) = \bigcap_{h \in F_h} R_h$. If Q is a "may" query, then we define the semantics of Q, i.e. the answer to Q, to be $May\_Answer(Q)$, and if Q is a "must" query its semantics is defined to be $Must\_Answer(Q)$. Thus, it is easy to see that the answer computed for the "may" query indicates possibility with respect to at least one of the future possible histories, while the answer computed with for a "must" query denotes definiteness of the result. Both these answers coincide when all the dynamic attributes are deterministic, i.e. H contains a single history.

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 [12]). **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.

The following query retrieves all the objects o of type "civilian" that may enter a restricted area P within three units of time from the current instance.

(I) RETRIEVE o
WHERE **may** ($o.type =$"civilian" $\wedge$ $P.type =$"restricted"$\wedge$
Eventually_within_c $INSIDE(o, P)$)

The following query retrieves all the civilian objects o that definitely (i.e. must) enter a restricted area P within three units of time, and stay in P for another 2 units of time.

(II) RETRIEVE o
WHERE **must** ($o.type =$"civilian" $\wedge$ $P.type =$"restricted"$\wedge$
Eventually_within_3 ($INSIDE(o, P) \wedge$
Always_for_2 $INSIDE(o, P)$))

The following query retrieves all the objects o that may 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 **may** (**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

Earlier in subsection 2.3, we have indicated two different ways for representing the positions of moving objects. In the reminder of this paper, we use the first of these schemes. For an object o moving on a route, we assume that $o.ubs$ and $o.lbs$, respectively, denote the upper and lower bounds on the speed of the object and that these bounds are positive ; we also assume that the attribute $o.route$ gives the identity of the route on which the object is traveling. We say that an object o is moving freely in 2-dimensional space if its velocities in the x and y directions are independent. For such an object o, we let $o.X.ubs$ and $o.X.lbs$ denote the upper and lower bound speeds in the direction of the x-axis, and $o.Y.ubs$ and $o.Y.lbs$ represent the corresponding speeds in the direction of the y-axis; each of these speeds can be positive or negative. (Note that for an object that moves on a route, the direction of its motion is determined by the route and its speed will give its state of motion at that point; on other hand for an object moving freely in 2-dimensional space we need to know its speeds in both the x and y directions). For a moving object, any of the above sub-attributes can be explicitly updated.

In this subsection, we consider the problem of evaluating queries in the MOST model. An FTL formula f is said to be a *restricted conjunctive* formula, if it has no negations appearing in it, the only temporal operators appearing in it are **until**, **until_within_c** and **Eventually_within_c**, and the *time_stamp* or the *time* variable does not appear in it; the last condition implies that for every query q that appears on the right hand side of an assignment in f (i.e. as in $[x \leftarrow q]$) the value returned by q at any time is independent of the time when it is evaluated and is only a function of the values to the free variables in q and the current positions of the objects. This condition also ensures that satisfaction of a non-temporal predicate when an object is at a particular position depends only on the position of the object but not the time when it reached the position. Also, note that f does not contain the **nexttime** operator.

The following theorem shows that the problem of evaluating a "may" query whose condition part is a conjunctive FTL formula is PSPACE-hard when the objects are moving freely in 2-dimensional space. This theorem is proved by exhibiting a straightforward reduction from the model-checking problem for conjunctive formulas which is a known PSPACE-hard problem [10].

THEOREM 1: Given a MOST database D modeling objects moving freely in 2-dimensional space, and given a "may" query whose condition part is given by a conjunctive FTL formula containing one free moving object variable, the problem of evaluating the query is a PSPACE-hard problem. □

Now, we consider the problem of evaluating "may" queries where the objects are moving on routes. Consider a query Q whose condition part is given by a conjunctive formula f with one free moving object variable o. Now consider an object, say $o1$, whose speed is in the range $[l, u]$. There are many possible histories corresponding to the varying speeds of $o1$. Let h be the possible history corresponding to the case where the object moves with the highest speed u at all times. Intuitively, it seems to be the case that if there is a possible history h' such that h' satisfies f at the first state with respect to the evaluation where the variable o is assigned object $o1$, then f is also satisfied at the beginning of h with respect to the same evaluation. This is due to the following properties: (a) in both the histories object $o1$ goes through the same positions (possibly at different times), (b) all the time bounds in the formula f are only upper bounds, and if these bounds are met when the object is moving at a lower speed then they will definitely be met when the object is moving at a higher speed, and (c) time does not appear any where else in the formula; this ensures that satisfaction of a non-temporal predicate at a particular time only depends on the position of the object but not the time when it reached the position.

Now, we have the following theorem.

THEOREM 2: Let f be a conjunctive FTL formula with one free object variable o ranging over moving objects, $o1$ be an object moving on a route with speed in the range $[l, u]$, ρ be an evaluation in which o is mapped to the object $o1$, and h be a history in which $o1$ is moving with the maximum speed u. Then, f is satisfied at the beginning of some possible history with respect to the evaluation ρ iff it is satisfied at the beginning of h with respect to ρ.

Proof: Let h' be any possible history that satisfies f at the beginning with respect to the evaluation ρ. For each $i \geq 0$, let s_i and t_i denote the i^{th} states in h and h' respectively. Since, in a history a new state is added whenever the position of any object changes, it is the case that the distance of any object in successive states of a history either remains unchanged or changes by 1. Hence, we can divide a history in to a sequence of sub-sequences $B_0, B_1, ..., B_i, ...$ of successive states such that ,for each $i \geq 0$, the distance of object $o1$ in any two states of B_i is same, and its distance in a state in B_i differs from a state in B_{i+1} by 1. Let $B_0, B_1, ...B_i, ...$ be the sequence of sub-sequences corresponding to h; similarly, let $C_0, C_1, ..., C_i, ...$ be such a sequence corresponding to h'. Since, in both the histories $o1$ starts from the same initial position, it is the case that for each $i \geq 0$, the distance of $o1$ in any state in B_i equals its distance in any state in C_i. For each $i \geq 0$, we say that every state in B_i *corresponds* to every state in C_i and vice versa. Let g be a subformula of f. Now, by a simple induction on the length of g, we show that

(*) If g is satisfied at t_i in h' and s_j is any state in h that corresponds to t_i then g is also satisfied at s_j in h'.

The proof is as follows. If g is an atomic formula then (*) holds because the satisfaction of g, with respect to an evaluation, only depends on the position of object o_1, and it is independent of the time. The non-trivial case in the induction is when g is of the form g_1 **until_within_c** g_2 where c is a positive constant. Assume that g is satisfied at t_i in h'. This implies that there exists some $i' \geq i$ such that g_2 is satisfied at $t_{i'}$, and for all k, $i \leq k < i'$, g_1 is satisfied at t_k; further more, the difference in the value of the *time_stamp* variable in the states $t_{i'}$ and t_i is bounded by c. Clearly, there is a state $s_{j'}$ in h that appears after s_i and that corresponds to $t_{i'}$; furthermore, every state appearing between s_j and $s_{j'}$ corresponds to some state appearing between t_i and $t_{i'}$. By induction, we see that g_2 is satisfied at $s_{j'}$, and g_1 is satisfied at s_j and at all states appearing after s_j but before $s_{j'}$. Also, the distance traversed by $o1$ from state s_j to $s_{j'}$ is same as that between t_i and $t_{i'}$. Since, in history h, $o1$ is traveling at a higher speed, it is the case that difference in the values of *time_stamp* in state $s_{j'}$ and s_j is smaller than between $t_{i'}$ and t_i. From all this, we see that the formula g_1 **until_within_c** g_2 is also satisfied at state s_j in h. The other cases in the proof are straightforward. □

Theorem 2 shows that, in order to answer the "may" queries whose condition part is a restricted conjunctive formula with a single free variable that ranges over moving objects, it is enough if we consider the single history where the objects are moving at the maximum speed. This corresponds to the deterministic case.

In the reminder of this section we present an algorithm for evaluating FTL queries for the case when the objects are moving at constant speeds on different routes. Our algorithm works for class of queries given by *conjunctive* formulas, and for the case when all the dynamic variables are deterministic. A conjunctive formula is an FTL formula without negation and without the **nexttime** operator and without any reference to the *time_stamp* variable. Even though conjunctive formulas can not explicitly refer to the *time_stamp* variable, one can express real-time properties using the real time temporal operators. Note that the class of conjunctive formulas is superset of the class of restricted conjunctive formulas.

In practice, most queries are indeed expressed by conjunctive queries. For instance, all the example queries we use in this paper are such. One of the main reasons 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 2.4), having $k+2$ attributes. The first k attributes give an instantiation ρ to the variables, and the last two attributes give 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 FTL query processing algorithm

Let $f(x_1, x_2, ..., x_k)$ be a conjunctive FTL formula with free variables $x_1, x_2, ..., x_k$ such that the variable *time_stamp* is also not referenced in it. 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. 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 $g(x_1, ..., x_k)$ be a subformula containing free variables $x_1, ..., x_k$. The relation R_g will have $(k+2)$ attributes; the first k attributes correspond to the k variables; the last two attributes in each tuple specify the beginning and ending of a time interval; we call this as the interval of the tuple. Each tuple in R_g denotes an instantiation ρ of values to the free variables in g and an interval I (specified by the last two columns) during which the formula g is satisfied with respect to ρ.

The algorithm computes R_g, inductively, for each subformula g in increasing lengths of the subformula. To do this it executes a sequence of one or more SQL queries whose result will be the desired relation R_g. We only describe how to generate these SQL queries. 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 $R(x_1, ..., x_k)$ 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 (for this example, if we assume that all objects are point objects, and that x_1 ranges over moving objects, and x_2 ranges over stationary objects, and that we have a relational database containing information about the the routes and speeds of moving objects and about the positions of statinary objects on the routes, then we can write an SQL query that computes a relation denoting the the ids of objects and the time intervals during which the predicate R is satisfied). 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. It is faily easy to see how we can write a single SQL query that computes R_g from R_{g_1} and R_{g_2}.

- 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 $p+2, q+2$ be the number of columns in R_1 and R_2 respectively. First, we compute another relation S from R_1 as follows. We define a *chain* in R_1 to be a set T of tuples in R_1 that give same values to the first p columns and such that the following property is satisfied: if l denotes the lowest value of the left end points of all intervals of tuples in T and u denotes the highest value of the right end points of these tuples ,then every time point in the interval $[l, u]$ is covered by an interval of some tuple in T (i.e., the interval $[l, u]$ is the union of all the intervals in T); we define T to be a *maximal chain* if no proper super set of it is a chain. The relation S is obtained by having one tuple corresponding to each maximal chain T in R_1 whose first p columns have the same values as those in T and whose interval is the interval $[l, u]$ as defined above. For example, if a maximal chain has three tuples with intervals $[10, 20]$, $[15, 30]$ $[11, 40]$ then these will be represented by a single tuple whose interval is $[10, 40]$.

  The resulting relation S satisfies the following property. For any two tuples $t, t' \in S$, if t, t' match on the first p columns (i.e. columns corresponding to the variables), then their intervals will be disjoint 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;

  The following SQL query computes S from R_1. For any tuple t, we let $t.l$ and $t.u$ denote the left and right end points of the interval of t.

```
SELECT(< list >, t1.l, t2.u)
FROM R1 t1, R1 t2
WHERE COND-B AND
        NOT EXISTS (
        SELECT t3
        FROM R1 t3, R1 t4
        WHERE COND-C AND
                        NOT EXISTS (
                          SELECT t5
                          FROM R1 t5
                            WHERE COND-D ))
```

  In the above query, the $< list >$ in the target list is the list of the first p attributes of $t1$. COND-B specifies that $t1$ and $t2$ give identical values to the first p columns and that $t1.l \leq t2.u$, and there is no other tuple whose interval contains $t2.u + 1$ or $t1.l - 1$; the later condition guarantees maximality of the chain. The WHERE clause of the outermost query states that $t1.l$ and $t2.l$ denote the left and right ends of a chain. This is indicated by stating that there are no tuples $t3$ and $t4$ whose intervals intersect with the interval $[t1.l, t2.u]$, and such that $t3.u < t4.l$ and such that there is a gap between $t3.u$ and $t4.l$; COND-C specifies the first of the two conditions; the existence of a gap between $t3.u$ and $t4.l$ is indicated by the inner most subquery starting with the clause "NOT EXISTS"; this subquery states that there is no tuple $t5$ whose interval intersects with the interval $[t3.u, t4.l]$; COND-C states the later condition. COND-B,COND-C and COND-D also specify that the first p columns of $t1$ thru $t5$ match.

Observe that if t_1, t_2 are any two tuples belonging to S and R_2, respectively, such that their intervals intersect, and $t_1.l \leq t_2.l$, and their values on common columns match, then g is satisfied throughout the interval $[t_1.l,\ t_2.u]$. Now, the relation R_g is computed from S and R_2 as follows. Let A be the union of all column names in S and R_2 that correspond to variables. The relation R_g will contain $|A| + 2$ columns. For each $t_1 \in S$ and $t_2 \in R_2$ that satisfy the above properties, the relation R_g will contain a tuple t such that $t.l = t_1.l, t.u = t_2.u$, and the first $|A|$ columns of t contain the corresponding values from t_1 or t_2. It is fairly straightforward to write a SQL query that computes R_g from S and R_2.

- Let $g = g_1$ **until_within_c** g_2 and R_1, R_2 be the relations corresponding to g_1 and g_2 respectively. Let S be the relation computed from R_1 as given in the previous case (i.e. the case for "until"). Let $t_1 \in S$ and $t_2 \in R_2$ be tuples that match on common columns and such that their intervals intersect and such that $t_1.l \leq t_2.l$. Let $d = \max\{t1.l, t_2.l - c\}$. It should be easy to see that g is satisfied throughout the interval $[d,\ t2.u]$ with respect to the evaluation given by columns corresponding to variables in t_1 and t_2. For every such tuples t_1 and t_2, there will be a tuple t in R_g with $t.l = d$, $t.u = t_2.u$ and such that the variable columns in t have the same values as in t_1 or t_2. It should be easy to write a SQL query that computes R_g from S and R_2.

- Let $g = g_1$ **until_after_c** g_2. Recall that g is satisfied at some point if g_2 is satisfied at some point which is at least c time units later and until then g_1 is satisfied. Let R_1, R_2, S, t_1 and t_2 be as in the previous case. Let $e = \min\{t_1.u, t_2.u\}$. Also assume that $t_1.l \leq e - c$. Now, it is easy to see that g is satisfied through out the interval $[t_1.l,\ e - c]$. Corresponding to each t_1, t_2 satisfying the above conditions, the relation R_g will have a tuple t such that $t.l = t_1.l$, $t.u = e - c$ and the variable columns in t have the same values as the corresponding columns in t_1 or t_2. We can easily write an SQL query that computes R_g from S and R_2.

- 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 $p + 3$ columns 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 two columns specify a time interval. Each tuple t in Q denotes the value of the atomic query q during the interval specified by the last two columns, 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 four columns; the first column gives the object id, the third and fourth columns give 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.

4 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).

4.1 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 its sub-attributes; two of the sub-attributes are $A.initialvalue$ and $A.updatetime$; the other subattributes specify how the attribute value changes over time. In case of when A is the position of a moving object, the other subattributes may be the upper and lower bounds on the speed, or upper and lower bound functions of time that denote the possible positions of the object at any time t.

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.

Recall that in section 4, we considered the problem of evaluating "may" queries in a MOST database system modeling the motion of objects. There we had shown that when objects are moving on well defined routes, and when there is uncertainty in their speeds, the evaluation problem for "may" queries, whose condition part is a restricted conjunctive formula, can be reduced to the deterministic case where the objects are traveling at their maximum speeds. We also presented a method for processing "may" queries for the the deterministic case when the condition part is given by an arbitrary conjunctive FTL formula f. This method, inductively, computes a relation R_g corresponding to each subformula g of f. For the case when g has no temporal operators, we assumed that the relation R_g are computed by some routines. The computation of R_g, for the case when g contains temporal operators, is achieved by translation in to SQL queries that refer to previously computed relations corresponding to smaller subformulas. Thus we can implement the above method on top of an existing DBMS that supports SQL provided we have a method for computing the relations R_g for the case when g has no temporal operators. The method that we outline below can be employed for this purpose also.

In this subsection, we address the problem of evaluating "may" queries whose condition part has no temporal operators and when there is uncertainty in the values of dynamic attributes. Our method applies to any type of uncertainty (i.e. it is not limited to the case of moving objects whose speeds are specified to lie between two bounds). Our method can be employed, as specified in the previous paragraph, to process non-temporal subformulas in the algorithm of section 4.

Now consider any "may" query whose condition part is non-temporal. If the query does not contain a reference to a dynamic 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 sub-attributes of A from the DBMS; and the MOST system computes the current range of possible values 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 do as follows. Let $p_1, ..., p_k$ be all such atoms. We first write F as $(F' \wedge p_1) \vee (F'' \wedge \neg p_1)$. We can repeat the above procedure for other atoms also to rewrite F into an equivalent condition of the form $(F_1 \wedge G_1)$ $(F_2 \wedge G_2)$ $... \wedge (F_r \wedge G_r)$ where the clauses $F_1, F_2, ..., F_r$ do not contain any dynamic attributes, and each clause G_i is a condition involving the atoms $p_1, ..., p_k$. In the worst case, r may be as much as 2^k. However, by identifying terms with common subexpressions, in practice, we can get r to be much smaller. As explained earlier, corresponding to each F_i, we create a query Q_i whose WHERE clause is F_i; the condition G_i is evaluated on each tuple in the result of Q_i by computing the current values of the dynamic attributes mentioned in Q_i. All these results are combined to obtain the answer to the main query.

4.2 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.

4.3 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.

5 Comparison to relevant work

One area of research that is relevant to the model and language presented in this paper is temporal databases [9, 13, 15]. 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 ([9] uses interpolation functions to some extent). In contrast, here we assume that dynamic attributes change continuously, and consequently the temporal data model is different than the data model presented in this paper. Thus, it is also not clear if and how temporal extensions to deal with imcomplete information (see [4, 7] are applicable to our context. Additionally, 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 [5] 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 [2, 3, 1]) domain, and to the spatial domain (see [6]). 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.

Another body of relevant work is location-dependent software systems (e.g. [8, 16, 2]). 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 ([12]) 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.

In [11] we considered the same issues as here, but we did not deal with imprecision; namely, a dynamic attribute of an object has a unique value at a particular time, rather than a set of possible values.

6 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. There can be uncertainty in the value of the dynamic variables. Such variables are represented by sub-attributes that specify their values over time. For moving objects, these sub-attributes specify upper and lower bounds on the speeds of the objects; or they give a pair of functions of time, and at any time the variable may have any value in the range whose lower and upper bounds are specified by the two functions. 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.

References

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

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

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

[4] C. Dyreson and R. Snodgrass. Valid-time indeterminacy. *International Conf. on Data Eng.*, Apr. 1993.

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

[6] 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.

[7] Y.-C. P. S. Gadia, S. Nair. Incomplete information in relational temporal databases. *Eighteenth VLDB*, Aug. 1992.

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

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

[10] A. P. Sistla and E. M. Clarke. Complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3), July 1985.

[11] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. *Thirteenth International Conference on Data Engineering*, April 1997.

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

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

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

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

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