An Object-Oriented Approach for Temporal Data

Liwu Li

School of Computer Science, University of Windsor, Canada

There is a growing need for easier access to temporal data. Recent developments of object-oriented data models represent the most promising approach to modeling complex aspects of the real world. Here, we present an object-oriented data model for supporting the storage and access of temporal information. The model consists of two parts. In the first part, we present the definition of a class for time object. The class is general enough so that it can be applied for various applications. Another part of the data model integrates time objects with entity objects and their attributes by following the pervasive three-dimensional metaphor of time. Particularly, the model incorporates with object-oriented technique all the three primary dimensions, which are time, entity, and attribute, of the metaphor.

Keywords: Class declaration, object-oriented database, object-oriented programming, temporal data.

1. Introduction

Time is ubiquitous in our daily life and in data processing. We have seen an increasing effort to introduce time into database technology. Particularly, the relational data model, the entity-relationship data model, and the object-oriented data model have been targeted for incorporating time as a new dimension of data. In recent years, computers are more accessible for many new applications in different areas like CAD and CASE as well as in business management. It is obvious that the wider range of applicat ions of computer and database technology proposes a wider range of demands, which include the storage and management of temporal information.

An important issue in designing a data model for temporal information is the representation of time. We should integrate a variety of formats of time to satisfy the various applications. Many new database applications are based on knowledge. A database system may need to understand and reason about the dynamic aspects of the real world (Maiocchi et al. 1992). A temporal data model must closely reflect the models of intelligent activities.

In this paper, we propose an object-oriented data model for modeling temporal information. The data model consists of two parts. First, we design time classes for encapsulating basic time objects like time point and time interval and other features of time. Then, we integrate the time objects with entities and attributes so that we can realize the three-dimensional metaphor of time. The data model is an object-oriented and temporal one. We study how to answer temporal queries for databases that are based on the data model.

We shall use the basic object-oriented syntax of C++ to describe code. It is easy for a reader to use other object-oriented programming languages such as Smalltalk (Goldberg and Robson 1989) or Eiffel (Meyer 1992) to encode the classes and functions declared here.

2. Preliminary Materials

2.1. Object-Oriented Data Model and Database Languages

Object-oriented concepts have been implemented in computer programming languages (Goldberg and Robson 1989, Ellis and Stroustrup 1990, Meyer 1992). The core concepts (Kim 1990) are:

• *object* and *class*. In an object-oriented database, real world entities are represented as *objects*, each of which has a system-wide unique identifier. A group of similar objects are described by a *class*. An object belonging to a class is called an *instance* of the class.

• *encapsulation* of attributes and methods. A class encapsulates the attributes and methods of its instances as its members. The *interface* of a class consists of some of the methods. Other members are hidden from the outside of the instances.

• *class inheritance*. The members defined in a class can be inherited (reused) by another class. The former class is called a superclass and the latter a subclass. The inheritance relationships between the classes of an object-oriented database form an acyclic directed graph, which is called a class *hierarchy*.

2.2. Allen's Model of Action and Time

Three of the characteristics of time intervals are important for reasoning temporal knowledge and modeling actions and processes (Allen 1983, p. 833). First, it allows imprecision so that relative temporal knowledge such as a time interval A before another time interval B can be represented. Second, it allows one to vary the granularity (grain) of reasoning. For instance, day and year can be used as different temporal granularities. Third, it supports temporal *persistence*, endorsed by default reasoning.

Allen's temporal logic is a typed first-order predicate calculus. Three types are used in Allen's model (Allen 1984):

• type TIME-INTERVAL for defining time intervals;

• type PROPERTY for defining propositions that hold during time intervals;

• terms that correspond to objects (real world entities) in the domain.

To describe a fact that a property p holds during a time interval t, the predicate HOLDS is used in the proposition

$$\text{HOLDS}(p, t).$$

There is a set of mutually exclusive primitive relations that can hold between time intervals. The temporal relations between two time intervals t_1 and t_2 described by Allen (Allen 1983, Allen 1984) are:

• $\text{DURING}(t_1, t_2)$: t_1 is contained within t_2;

• $\text{STARTS}(t_1, t_2)$: t_1 shares the same beginning as t_2 but ends before t_2;

• $\text{FINISHES}(t_1, t_2)$: t_1 shares the same end as t_2 but begins after t_2 begins;

• $\text{BEFORE}(t_1, t_2)$: t_1 ends before t_2 begins;

• $\text{OVERLAP}(t_1, t_2)$: t_1 begins before t_2 begins and ends before t_2 ends, t_2 begins before t_1 ends;

• $\text{MEETS}(t_1, t_2)$: t_1 ends when t_2 begins;

• $\text{EQUAL}(t_1, t_2)$: t_1 and t_2 are the same.

Allen's model of action and time defines a set of axioms for reasoning the occurrences of events and describing actions and processes. The axioms define the behavior of the above relations. The predicate IN is defined with the above relations by the logical equivalence

$$\text{IN}(t_1, t_2) \iff ((\text{DURING}(t_1, t_2) \vee (\text{STARTS}(t_1, t_2) \vee (\text{FINISHES}(t_1, t_2)).$$

2.3. Temporal Data Cube and Its Internal View

The pervasive spatial metaphor of time, which is a cube of temporal data, has been refined in Ariav's paper (Ariav 1986) to define a temporally oriented data model, TODM. The model is a consistent set of temporally oriented data constructs, operations, and constraints. The following three principles have been used by Ariav to extend the relational data model:

• *temporal completeness*. The complete set of recorded events, which brings about the state transitions of a historical database and describes the enterprise over the period of time covered by the database.

• *temporal density*. The state of a historic database between events can be determined by the persistency of the latest recorded event.

• *temporal isomorphism*. A historical database evolves in an order and pace with respect to the events.

The temporal path through which a database, which may not be a historical database, has progressed can be described by a three-dimensional cube.

3. Time Classes

Usually, the representation of a time point is relative to a reference point, which may be an assumed origin of time. For example, the years are counted with respect to the year of Lord. Therefore, we can have both a negative time point like the year 1 BC and a positive time point like AD 1995. Another property of the representation of time point is that a time grain or granularity must be used to determine the distance between the time point and the reference point. For instance, the difference between Sunday and Tuesday can be internally described as $2 - 0 = 2$ in a computer program. Without the granularity of day, the value 2 is meaningless or confusing.

We assume all the time points are relative to the same temporal reference point. The representation of the unknown unique temporal reference point may be refined to any finer granularity level, say a minute, a second, even a nanosecond. It may also be abstracted to any coarser granularity level say a year, a decade, or even a century.

In a temporal database system, we have to deal with different grains, or granularities, of time. We assume an enumeration type Granularity, the constant enumerators defined by the type may include second, minute, hour, day, month, year, etc.

3.1. Time Point vs. Time Interval

We shall include both time point and time interval as fundamental objects in the proposed temporal data model. The integration of both forms of time will enhance not only the expressive power of a database but also the flexibility of reasoning about temporal information. In the real world, we often have situations where the only reasonable index for an activity is an instantaneous time point. For example, we often hear the phrase "the official startin g time" for a scheduled activity. We may not be able to distinguish one tenth of a second. However, using a time interval between the starting and ending points of the first tenth of a second as the formal interpretation of "the official starting time" is obviously not acceptable psychologically and technically.

For temporal information, there may be uncertainty about the distinction between a time point and a time interval. Sometimes, we do not know whether a user of the temporal database will treat the time associated with an event as a time point or as a time interval. With a robot as example, the action of taking a picture by the robot may be interpreted as happening instantaneously by a user of the robot. It may also be interpreted as lasting for a while by the robot builder. If the action has to be recorded i n a database, a time object should be assigned with the action. But, the type of the time object is uncertain when the database is designed. A possible solution for a temporal data model is to provide support for converting a time point to a time interval, and *vice versa*.

Another reason for the conversion between time points and intervals is related to abstraction. Abstraction is used to control the complexity of representation. For example, we may need to describe an activity of writing a report started at 8:00 am of Monday, pausing an hour for lunch break, and continuing till 4:00 PM. The activity is often described as "writing a report on Monday." On the other hand, we may need to provide details for the abstraction so that the "t ime point" or the time interval Monday should be replaced by several time intervals.

3.2. A First Abstraction of Time Object

In object-oriented programming, the abstract view of an object is its interface. The data stored in the object can be accessed only through the interface. Thus, the private data and other implementation details can be *hidden*. Here, we present both the time point and the time interval as instances of an abstract data type (ADT). We consider what functions should be included in the interface of the ADT. The result is an interface for a class called TIME.

An intrinsic property of a time object is its *length*. In our daily life, we often take the length of a time point as equal to zero by default. This is no longer true in database area. As presented before, a time point at a coarse granularity may be interpreted as a disguised time interval at a fine granularity. For instance, the date 90/6/11 may be used as an abstraction of a time interval, which includes all the minutes between the minutes 90/6/11/0/0 and 90/6/11/23/59.

We shall treat the proper ty of lengt h as an attribute of a time object.

In Smalltalk (Goldberg and Robson 1989) and Eiffel (Meyer 1992), a method or function with no argument may not be distinguished from an attribute by its clients. This favorable ambiguity gives the interface designer a freedom to change the design of an attribute into a method, and *vice versa*. The length of a time object could be implemented as an attribute if we can save repeated computation of the value. In this paper, we propose implementing the property length as a function for two reasons.

1. The length of a time object may depend on other attributes. For example, the length of a time interval depends on the granularity as well as its endpoints. If the length is implemented as an attribute, it must be modified each time when we modify the positions of its ends or when we change the granularity.

2. The interface of an object should consist of only functions, which are accessors and implementors (Coad and Nicola 1993). The function implementation of length does not prevent us from having a private data member, say Length, encapsulated in the object.

Therefore, the inclusion of a length function in the interface of time objects introduces flexibility for the programmers of a time class.

The length function needs an actual argument of type Granularity. Thus, a user can request the length of a time object with respect to the user's assumed granularity rather than the granularity encapsulated in the time object. In terms of C++, the member function *length* in the interface of class TIME has a prototype

float *length*(Granularity = NULL);

The default value NULL can be used to indicate that the granularity stored in the target object should be used to calculate the length.

Related to the length of a time object, we have a group of arithmetic functions for class TIME. For the completeness, the length function is also included in the following list of arithmetic function prototypes:

- float *length*(Granularity = NULL);
- float *add*(float, Granularity = NULL);
- float *add*(TIME&);
- float *subtract*(float, Granularity = NULL);
- float *subtract*(TIME&);
- float *multiply*(float);
- float *divide*(float);

In the above list, we have two *add* functions and two *subtract* functions. The first *add* requires a scalar value along with an optional granularity as arguments. It accommodates the information connoted in sentence "I finished the job in ten more minutes than another job." If the mentioned another job encapsulates a time object, which is an instance t_j of class TIME, the time required by the current job will be

$$t_j.add(10, \text{minute}).$$

The second *add* has an instance of TIME as argument. It is designed to calculate the time described in sentence: "To finish this job, I have to do job 1 and job 2." If the time objects encapsulated in jobs 1 and 2 are t_1 and t_2, the time required by the whole job will be

$$t_1.add(t_2).$$

The two *subtract* functions are used similarly.

The function *multiply* is used to compute the length of a TIME object multiplied by a scalar value. The function *divide* returns the length of a TIME object divided by the actual argument. As a matter of fact, only one of the two functions is necessary since the former function with actual argument f can be simulated by the second with argument $1/f$, and *vice versa*. For the sake of conceptual completeness, we include both member functions in class TIME.

Note that each of the above arithmetic functions, except *length*, can be realized by an expression that involves function *length* and arithmetic operators $+, -, *$, and $/$. But, the functions do more than just simple arithmetic operations. For example, the first *add* function understands granularity, which is not understood by the ordinary addition operator $+$.

In temporal data processing, we often need to compare the lengths of two time intervals. We define *relational operators* that compare the lengths of two TIME objects. Particularly, we overload the C++ binary operators >=, <=, >, <, ==, and != as follows:

- Boolean **operator** >= (TIME&);

- Boolean **operator** <= (TIME&);
- Boolean **operator** > (TIME&);
- Boolean **operator** < (TIME&);
- Boolean **operator** == (TIME&);
- Boolean **operator** != (TIME&);

The relational operators can be implemented with the *length* function and the ordinary relational operators. In the implementation, a problem is that how we can compare two lengths that may be measured in different granularities. A solution is when we invoke the function *length* for two TIME objects, we can pass the same value for the optional Granularity argument. Thus, we do not have to explicitly deal with the granularities for the relational functions. The above operators make it easy to encode queries against a temporal database.

The spatial relation between time intervals is important for AI (Allen 1984) and for temporal information deduction (Maiocchi et al. 1992). In addition to the above arithmetic relational operators, we include the following *spatial relational functions* in the interface of class TIME:

- Boolean *before*(TIME&);
- Boolean *equal*(TIME&);
- Boolean *meets*(TIME&);
- Boolean *overlap*(TIME&);
- Boolean *during*(TIME&);
- Boolean *starts*(TIME&);
- Boolean *finishes*(TIME&);

To simplify the discussion, we now assume that each time point is a genuine time point, and each time interval is a genuine time interval. That is the length of each time point is zero and the length of each time interval is greater then zero.

For two time intervals with both ends closed, the meaning of the above functions follows the usual semantics of the corresponding relations, which has been well established (Allen 1983, Allen 1984, Maiocchi et al. 1992). Since we no longer insist that both ends of a time interval be closed, we need to extend the above relational functions for time points and intervals with open ends. The principle is that a time point that happens to be the open end of a time interval does not meet, overlap, start, or finish the time interval. It is *before* the time interval if the time point is the lower open end of the interval, and the interval is *before* the point if the point is the higher open end of the interval. Based on the principle, we can uniquely determine the relative positions between any two TIME objects. For example, if two time points represent the same spatial position, the only relationship between the two points is *equal*, they do not satisfy any of the relations *meets, overlap, during, starts*, and *finishes*. For a time point t and a time interval s, if the interval includes the point, there are three possible relationships between them, which are

- t *meets* s, if s has its lower end point closed and equal to t;
- t is *during* s, if t is between the two end points (each of which may be open) of s;
- s *meets* t, if s has its higher end point closed and equal to t.

We now assume that each TIME object is either a time point, which is a genuine point or a disguised interval, or a time interval, which has a length equal to zero or a non-zero value. We can generalize Allen's completeness conclusion with the following proposition.

Proposition 3.1. *For any two* TIME *objects* t_1 *and* t_2, *exactly one of the relational functions before, equal, meets, overlap, during, starts, and finishes can be used to replace the function name Func so that the disjunctive Boolean expression*

$$t_1.Func(t_2) \vee t_2.Func(t_1)$$

has truth value true.

PROOF. By simple enumeration of all the possible functions and the relative spatial positions of the two time objects t_1 and t_2. □

The spatial relational functions described above for TIME objects can help avoid the truth inconsistency. A proposition that is true at an open end of a time interval may be false during the whole time interval, and *vice versa*.

The TIME class does not prevent the interpretation of a time interval as an infinite set of time points or as a discrete set of chronons. If we regard a time interval as a set of time points, the ordinary set operations can provide convenient ways to combine two TIME objects or subtract

a TIME object from another. Here, a time point is regarded as a singleton set. We include the following set operations in the interface of class TIME:

- TIME& *intersect*(TIME&);
- TIME& *union*(TIME&);
- TIME& *subtract*(TIME&);

Note that applying a set operation to two TIME objects may not always make sense because the operation may not be able to produce a valid TIME object. For example, if two time objects do not intersect with each other, the function *intersect* applied to the two objects cannot find any meaningful TIME object as the result. Therefore, the responsibility of correctly applying the above functions will be delegated to the clients of the TIME class. Testing the intersection of two TIME objects should be conducted before the function *intersect* is invoked for them.

In the above list of interface functions for class TIME, we have not declared a function for creating TIME objects. In terms of the C++ language, we have not declared a constructor for the class yet. Since a constructor of a class is responsible for initializing the data members in each instance, the design of a constructor for the class needs the design of an internal data representation for the class. We shall address the question of constructor design after we decide the internal structure for TIME objects.

3.3. Basic Time Objects and Time Class Hierarchies

Traditionally, time point and interval are two different notions. As mentioned before, for information processing, we may need to interpret a time point as a time interval, and *vice versa* (Ariav 1986, Maiocchi et al. 1992). We shall present two object-oriented designs for time object. The first design follows the traditional viewpoint and provides two different classes for time point and time interval. The second design integrates time point with time interval.

For the first design, the class TIME described in the last subsection will be designated as an *abstract* class, which cannot have any instance and which provides common conventional functions for temporal information. To honor the distinctions between a time point and a time interval, we derive subclasses TPoint and TInterval from the class TIME. Since the two subclasses inherit class TIME, we only need to declare new attributes (data members) and define methods (member functions).

In class TPoint, a variable called position of float type records the spatial position of a time point. The granularity is recorded in a variable granularity of type Granularity. In class TInterval of time intervals, variables s_position and e_position of float type records the staring and ending positions of a time interval. A variable granularity of type Granularity is also declared in class TInterval. The classes TPoint and TInterval inherit arithmetic functions from the class TIME. The definitions of the functions are based on the above interpretations of function *length*. The constructor for the class TPoint or TInterval should assign default values for the data member position or data members s_position and e_position. It should also assign a default value of Granularity to the data member granularity.

With the above design, it is not easy to physically convert an object in class TPoint to an object in class TInterval, nor *vice versa*. If we need to abstract a time interval into a time point or interpret a time point as a time interval, we have to explicitly create a new instance of a time class, say TInterval, initialize the object with the information stored in an existing instance of another class, TPoint, and finally delete the existing object. The "conversion" is expensive in terms of computation time.

In the second class design for time objects, a notion of general programming technique called *variant record* (Headington and Riley 1994, p. 223) is used to accommodate different types of data with the definition of one class. The class TIME will be expanded with data members to accommodate a time point or a time interval. Particularly, we declare two variables, called s_position and e_position, in the class TIME. To represent a time point with an instance of the class TIME, we can simply assign both s_position and e_position with the same float value. A variable named granularity is in the class TIME. It is used to determine the distances of the positions s_position and e_position from their common reference point and the *length* of the TIME object.

For both designs of time class, there are four possible types of closeness for a time interval:

- a time interval may be *open* at both ends;
- a time interval may be *closed* at both ends;
- a time interval may be *semi-closed* with only the lower end closed;
- a time interval may be *semi-open* with only the lower end open.

The four closeness types will be used to define an enumeration type, named Closeness, which consists of four constant enumerators open, closed, semi_closed, and semi_open. It is possible to use all the four types of time intervals in a single application. For the first design of time classes, we declare variable closeness only in the class TInterval. The variable is in class TIME in the second design.

In addition to the property closeness, another property for a time interval is *discreteness* or *denseness*. In some applications, a finite time interval must be interpreted as a dense interval, i.e., it consists of an unaccountable number of points. For other applications, the same time interval may be interpreted as consisting of chronons and, thus, it is discrete. The denseness of a time interval should be decided by the applications. We leave the specification of denseness of time intervals to the users.

3.4. Time Structures

In the study of temporal data model, a history is a linearly ordered list indexed on pairwise disjoint time points and/or intervals. A linear data structure such as an array or a linked list is appropriate for a history. As a matter of fact, a set of time objects recorded for a property, an event, or an activity may not be pairwise disjoint. A forest may be applied to store the time intervals that satisfy the IN-property. In each tree of the forest, the parent node represents a time interval which includes each of its child time objects. A forest is also appropriate for time branching property, studied in temporal logic (Van Benthem 1982).

4. An Object-Oriented Temporal Data Model

Ariav (Ariav 1986) provided an operational temporal data model, named TODM, based on the three-dimensional metaphor of time. The notion of a *data cube* consists of the three concepts, *entity*, *attribute*, and *time*. There is psychological evidence (Aaronson 1972) which suggests the cubic form is the favorite metaphor for presenting temporal information, and a vertical axis for the time dimension is the least ambiguous to identify temporal relationships (Ariav 1986, p. 501). The temporal path through which a database has progressed is indexed with the dense and continuous time axis. A database based on TODM contains three-dimensional cubes of data as its data spaces.

We can apply Ariav's approach to the object-oriented data model. For an entity object and a time object, say an aircraft a and a time point t, we can uniquely determine a readiness level by a function named readinessHistory. Let us assume two class names: the class READINESS describes the domain of the readiness level, and the instances of class AIRCRAFT represent aircraft. A C++ prototype of the function would be

```
READINESS AIRCRAFT
    ::readinessHistory(TIME&);
```

Note that the member function readinessHistory included in an instance of AIRCRAFT may not be interesting or important for a user. But, the function allows a user of the object-oriented system to move upwhen and downwhen by invoking the function with different time objects. To present the relationship directly for an end-user of database, a simple interface between the database and the user should be designed to hide the function readinessHistory. The interface can be used to provide a three-dimensional cubic representation for the stored data.

In terms of the above example, we describe a conceptual temporal data model, named object-oriented temporal data model (OOTDM). Like other object-oriented data models, it has the ordinary object-oriented concepts. In a class in OOTDM, we provide a member function such as readinessHistory to describe a dynamic property for the instances of the class. The member

function is called a *temporal function*, which requires a time object as an argument and which returns an attribute value as result. By varying the time object argument, we may observe the states of an object at different moments.

Note that the object dimension in Ariav's TODM is realized with the notion of a relational tuple. The data model OOTDM is more flexible than TODM. It provides more expressive power than Ariav's TODM. Since the function used in Ariav's approach and that used in the OOTDM for integrating time with attribute have similar syntax and interpretation, we can ascribe the OOTDM as an object-oriented data model that is based on the three-dimensional metaphor of time.

We use an object-oriented modeling and design (OOMD) approach to analyze the model OOTDM. In OOTDM, we use a member function to describe a dynamic aspect of real world entities. The function associates TIME objects with the domain of an attribute. In OOMD, we often need to describe the relationship between two classes. If we make the involved object explicit, the relationship between the class TIME and the attribute can be described as a ternary *association*, which is an important notion of the OOMD approach proposed by Rumbaugh *et al* (Rumbaugh et al. 1991). The three dimensions of OOTDM are modeled by a ternary association.

4.3. An Internal Representation of Temporal Data

For data management, a temporal attribute is essentially a finite collection of binary tuples. Each tuple has a basic value from the domain of the attribute as one component and a TIME object as another component. The second component may be a structure of TIME objects so that the entity may have the attribute value at different times.

We can use the data structure *dictionary* to manage TIME objects for temporal attributes. A dictionary can be used to answer whether a key k is in the structure with query MEMBER(k). It can be modified by the insertion and deletion of keys with commands INSERT(k) and DELETE(k) (Aho et al. 1974). In this paper, we assume that along with each key k, the dictionary also keeps an "explanation" of the key. The explanation is a structure v_k of TIME objects t such that the entity has the attribute v. We assume the structure v_k to be a vector. To use the information stored in the vector v_k, we need to extend the operations on the dictionary as follows. The answer to a query MEMBER(k) is a vector v_k of TIME objects. The query returns an empty vector v_k to indicate that the entity has not been in the state which has the attribute value k. The operation INSERT(k, t) has a TIME object t as its second argument. It stores the object t in to the vector v_k. The operations DELETE(k, t), DELETE(t), and DELETE(k) can be similarly implemented for deleting a value-time tuple, a time object, or an attribute value. Note that when we use the command DELETE(k) to delete an attribute value k, the vector v_k has to be deleted as well.

The method of encapsulation applies to record temporal information in two ways in the model OOTDM. First, encapsulation is applied to define the classes for entities by encapsulating temporal attributes into the entities. Secondly, the temporal attributes encapsulate both the ordinary domain values and the TIME objects.

5. Temporal Queries

Let us assume an object-oriented database that is based on the model OOTDM. For an entity object described in the database and a temporal attribute of the entity, we assume that a dictionary is used to keep the history of the changes of the attribute value. Particularly, we assume that the dictionary associates a vector of TIME objects for each basic value in the domain of the attribute.

To simplify the following discussion, we assume that the TIME objects in the vector for an attribute value do not intersect; otherwise, we can replace the intersecting TIME objects with their union, which is a single TIME object. We also assume that if a TIME object is an interval with positive length, the lower end of the interval is closed and the upper end is open. A reader can modify the following algorithms for the general situations, in which a vector may contain time point and all the four types of time intervals.

For an entity class E and an attribute A of the entities described in class E, we use the predicate

$\text{HOLDS}_{E,A}$, or simply HOLDS when the class E and attribute A are clear from the context, to describe the contents of a temporal database. The predicate $\text{HOLDS}(o, k, t)$ is used to indicate that an entity object o in class E has value k for attribute A at time t. For a temporal database based on OOTDM, the predicate has a unique truth value for each combination of the argument values o, k, t. The meaning of the predicate HOLDS with respect to model OOTDM is that in an object o, if the vector v_k from the data member A contains the TIME object t, the predicate $\text{HOLDS}(o, k, t)$ is true. By replacing the arguments o, k, and/or t with variables of appropriate types, the predicate can be used to query the database for an entity, for an attribute value, and/or for a TIME object such that the predicate holds. We can extend the exiting object-oriented database languages with the predicate HOLDS so that a user can propose temporal queries to access the temporal information stored in an OOTDM-based database.

In this section, we study three types of temporal query. Other types of temporal query can be handled similarly. First, we study how to answer the query for the attribute value of a given entity at a given time point. The second type of query asks for the sequence of attribute values for a given object in a given time interval. The third type of query requests for the time objects when we already know the attribute value of a given entity. The three types of temporal query are discussed in the following three subsections. In the first two subsections, we assume that there is no implication or subsumption relationships between the values of the attribute. This assumption will be relaxed in the last subsection. In the following discussion, we assume a given object o.

5.1. Query for Attribute Value at Time Point

We use this type of query to illustrate how to apply the temporal density principle to deduce the attribute value. The question to be answered is for a given TIME object t that is a time point, we need to find a value k of attribute A such that the predicate $\text{HOLDS}(o, k, t)$ is true with respect to a temporal database. It is possible that no TIME object that is equal to or contains t has recorded in the database for the entity o and attribute A. In this case, we have to apply the *temporal density principle* (Ariav 1986) to determine an attribute value. That is if prior to the given TIME object t, the latest time point that has been recorded for the entity is t' and at t' the entity o has attribute value k, the answer to the query should be k.

We now describe an algorithm for answering the query. In the following algorithm, for TIME object t and value k of the attribute A, we try to find in the vector v_k a TIME object t' that is equal to or includes the given TIME object. If we can find such a recorded TIME object t' in some vector v_k, the value k is returned as the answer. If the search fails for every vector v_k, we apply the temporal density principle. This requires us to determine a TIME object $s(k, t)$ from the TIME vector v_k such that $s(k, t)$ is the latest among all the object in v_k that are earlier than t. We can determine the latest TIME object from the set of TIME objects $s(k, t)$ for all the attribute values k. If the latest TIME object comes from vector v_k, the value k is returned as the answer to the query. Note that the TIME vector v_k can be returned by a query $\text{MEMBER}(k)$ for the data member A.

In the presentation of algorithms for processing temporal queries, we often use the term *earlier* to indicate the relationship *before* between two time objects. The term *later* will be used as the inversion of *before*. We use the term the *latest time object* with respect to a given time object t to refer to a time object s in a set S of time objects such that s is earlier than t but later than all other time objects in S that are earlier than t.

Algorithm 5.1: Query for Attribute Value at Time Point

Input: Entity object o, time point t, and temporal attribute A.

Output: A value k in the domain of attribute A such that the predicate $\text{HOLDS}(o, k, t)$ is true for the given database.

Step 0 (Initialization) Let S represent a set of TIME objects. Initialize S with an empty set.

Step 1 For each value k of attribute A, select a TIME object, denoted by $s(k, t)$, from the vector v_k such that

- either $s(k, t)$ is a point and is equal to t or $s(k, t)$ is an interval and contains

t if such a TIME object exists in v_k; otherwise,

- determine a TIME object $s(k,t)$ from the vector v_k such that $s(k,t)$ is earlier than t and it is later than all other objects in v_k that are earlier than t. Add the object $s(k,t)$ to the set S.

If a TIME object $s(k,t)$ can be found in the first case for any value k, return the value k as the answer and terminate the algorithm; otherwise, if the set S is empty, return and report that "no value of attribute A can be determined for the time point t."

Step 2 Note that the TIME objects in set S do not intersect with each other. Determine the latest object $s(k,t)$ from S. Return the value k as the answer. □

The above algorithm determines the attribute value k for a given entity o of class E and a given TIME object t, which is a time point, so that the predicate HOLDS(o,k,t) holds for the database. We explain Algorithm 5.1 as follows. The vectors v_k of all the values k can be combined into a linear list T of TIME objects such that each pair of adjacent objects t_i and t_{i+1} in the list satisfies the relationship $before(t_i, t_{i+1})$. If the given time point t is equal to or is included in some element t_i in the list T, the value k that is associated with the TIME object t_i is returned as the answer in the first case of Step 1. Otherwise, we have to find the latest TIME object s from T that is earlier than t. If such a TIME object s exists in T, it must be in the set S, which is built in Step 1. The readiness level k associated with the TIME object s determines the answer according to the temporal density principle. If the set S is empty, there is no TIME object recorded for entity o and attribute A that is earlier than or equal to t and, therefore, we cannot find an answer for the query. We use the following example to illustrate Algorithm 5.1.

Example 5.1: (Query for attribute value at time point) We use a real number to describe a time point and a pair of real numbers enclosed by [and) to represent a time interval. Suppose an aircraft o in a database, which has recorded only the readiness levels 1 and 3 for the entity. The vectors

$$v_1 = \{[0.5, 0.9), 1.2, [1.5, 1.9)\},$$
$$v_3 = \{0.4, [1.0, 1.1), 2\},$$
$$v_k = \emptyset \quad \text{for } k \neq 1, 3.$$

Note that the list T, which includes all the TIME objects recorded for object o, is

$$T=\{0.4, [0.5, 0.9), [1.0, 1.1), 1.2, [1.5, 1.9), 2\}.$$

Assume we are given time point $t = 0.3$. Since no TIME object earlier than t has ever been recorded in the database for the attribute readiness level, we cannot decide an attribute value for the time t. Accordingly, the second case in Step 1 of Algorithm 5.1 returns an empty set S.

For a given time point $t = 1.7$, the interval $[1.5, 1.9)$ contains t. We can decide the attribute value 1 for the time t. Accordingly, the first case in Step 1 of Algorithm 5.1 returns the readiness level 1 as the answer.

For the time point $t = 1.3$, there is no TIME object in T that is equal to or includes t. Therefore, we have to apply the temporal density principle. The second case of Step 1 will collect the TIME objects 1.2 and $[1.0, 1.1)$ from the vectors v_1 and v_3 into the set S. In Step 2, the latest TIME object in S is $s(1,t) = 1.2$, and the readiness level $k = 1$ is returned as the answer. □

Notice that we assume both time points and intervals may appear in a vector v_k. An algorithm similar to Algorithm 5.1 is still necessary when only time points or only time intervals are allowed to appear in vectors v_k. As a matter of fact, we still need the temporal density principle to answer some queries for the temporal database.

5.2. Query for Attribute Values during Time Interval

We now study how to answer a user's query for the attribute values during a time interval i. A difference of the query from the query processed by Algorithm 5.1 for a time point is that the time interval i may be long enough so that the attribute has changed its value during the time interval. Multiple values of the attribute may be returned as the answer to the query.

Let us assume again the linear list T, which includes the TIME objects in the vectors v_k for all the attribute values k. We now use the list T to describe an algorithm for answering the query. We denote the interval with $i = [p, q)$ with real numbers p and q. The algorithm decides whether there is a TIME object in T that includes q or whether there is a TIME object in T that is earlier than q. If there is no such object in T, there is no recorded attribute value for any time point that is included in or earlier than i, we cannot answer the query and the algorithm terminates. Otherwise, for the ending time point q of interval i, we determine a TIME object s_0 from the linear list T such that one of the following two conditions is true:

- s_0 is an interval that *includes* q if there is such an interval in T; otherwise
- s_0 is the TIME object earlier than q and it is the latest among all the TIME objects in T that are earlier than q.

The attribute value associated with the TIME object s_0 is included in the answer set. Note that the reasons for selecting the attribute value as an answer to the query are different for the two cases. In the first case, an attribute value has been recorded for the time point that is immediately before q; in the second case, we have to apply the temporal density principle to imply an attribute value for the time point that is immediately before q. For the first case, the algorithm will use the starting point of the interval s_0 to replace q. For the second case, we need to distinguish two situations: the TIME object s_0 is a point or it is an interval. If s_0 is a time point, the algorithm will use s_0 as the new value of q; if s_0 is an interval, the algorithm will use the starting point of s_0 as the new value of q. After we reset q to its new value, we have to check whether q is later than p. If q is later than p, we repeat the above process; otherwise, all the possible attribute values during the interval i have been found and the algorithm terminates. Thus, the algorithm accumulates a list of attribute values, which is the answer to the given query.

Since the algorithm is more complex than Algorithm 5.1, we use an example to illustrate the basic idea encoded in the algorithm before we present the algorithm.

Example 5.2: (Query for attribute values during time interval) Like in Example 5.1, we suppose an aircraft o in a database, which has recorded only the readiness levels 1 and 3 for the entity, and the vectors

$$v_1 = \{[0.5, 0.9), 1.2, [1.5, 1.9)\},$$
$$v_3 = \{0.4, [1.0, 1.1), 2\},$$
$$v_k = \emptyset \quad \text{for } k \neq 1, 3.$$

The list T, which includes all the TIME objects, is

$$T=\{0.4, [0.5, 0.9), [1.0, 1.1), 1.2, [1.5, 1.9), 2\}.$$

Assume we are given a time interval $i = [1.05, 2.3)$.

First, we have $q = 2.3$. From each of the vectors v_1 and v_3, we need to select a TIME object that includes q or is earlier than q. Thus, we establish a set $S = \{[1.5, 1.9), 2\}$. None of the elements in S includes q. The latest object in S is 2, which is from the vector v_3. Therefore, we insert the readiness level 3 in the answer set and reset q with 2.

For $q = 2$, the latest TIME objects that either include q or are earlier than q in the vectors v_1 and v_3 are $[1.5, 1.9)$ and $[1.0, 1.1)$. Since the time interval $[1.5, 1.9)$ from v_1 is later than $[1.0, 1.1)$, we have a readiness level 1 for the aircraft during the given time interval i. In this case, we also need to reset the time point variable q with value 1.5.

For $q = 1.5$, the latest TIME objects that either include q or are earlier than q in the vectors v_1 and v_3 are 1.2 and $[1.0, 1.1)$. Thus, we have a readiness level 1 again for the aircraft during the given time interval i and we reset the time point variable q with value 1.2.

For $q = 1.2$, the latest TIME objects that either include q or are earlier than q in the vectors v_1 and v_3 are $[0.5, 0.9)$ and $[1.0, 1.1)$. Thus, we have a readiness level 3 for the aircraft during the given time interval i and we reset the time point variable q with value 1.0.

Since the new value of q is less than $p = 1.05$, the algorithm outputs the ordered list $\{2, 1, 1, 2\}$ as the desired answer and terminates. □

Algorithm 5.2: Query for Attribute Values during Time Interval

Input: Entity object o, time interval i, and temporal attribute A.

Output: A list of values k in the domain of attribute A such that the predicate HOLDS(o, k, i) holds for the given database.

Step 0 (Initialization) Assume two variables p and q, which are initialized with the values s_positon and e_position encapsulated in i respectively; i.e., we have $i = [p, q)$. Let S represent a set of TIME objects and R be a stack with elements from the domain of A. Initialize S and R with empty set and empty stack, respectively.

Step 1 For each value k of attribute A, select a TIME object, denoted by $s(k, q)$, from the vector v_k such that

- $s(k, q)$ is an interval and contains q if such a TIME object exists in v_k; otherwise,
- determine a TIME object $s(k, q)$ from the vector v_k such that $s(k, q)$ is earlier than q and it is later than all other objects in v_k that are earlier than q. Add the object $s(k, q)$ to the set S.

If a TIME interval $s(k, q)$ can be found in the first case for a value k, push the value k into stack R, set q with the starting point of $s(k, q)$, and go to Step 2. If the set S is empty, go to Step 3. Otherwise, determine the latest object $s(k, q)$ from S, insert value k into stack R. If $s(k, q)$ is a time point, set $q = s(k, q)$; otherwise, set q with the s_position value of $s(k, q)$.

Step 2 If p is earlier than q, reset S into an empty set and go to Step 1; otherwise, go to Step 3.

Step 3 Pop the values from stack R as output and terminate the algorithm. □

If we denote the number of vectors v_k that are recorded in the database with m and the maximum size of the vectors with n, the above algorithm has a time complexity $O(m^2 \times n^2)$. In fact, the factor $O(m \times n)$ represents the time required by Step 1. It also represents the number of times that Step 2 is repeated. If binary search is applied to each vector v_k to find the latest TIME object that is earlier than q in Step 1, the time required by Step 1 will be $O(m \times \log n)$. If we denote the total number of TIME objects in all the vectors as N, the time required by Step 2 will be in the order $O(N)$. Therefore, another expression for the time complexity of Algorithm 5.2 is $O(N \times m \times \log n)$.

5.3. Query for TIME Objects

If there is no logical relationship between the domain values of attribute A, the query HOLDS(o, k, t), where o is an instance of entity class E, k is a value in the domain of attribute A, and t is a variable of type TIME, can be easily handled. The answer to the query is the set of TIME objects contained in the vector v_k. The vector is returned by query MEMBER(k) for the attribute A. (Note that the attribute value is a dictionary.)

Sometimes, relationships exist among the attribute values. For example, we can assume that if an aircraft o is at readiness level i, it is also at any readiness level j with $j \geq i$. This type of implication can be recorded in the intentional component of a deductive database. It may be directly reflected in a query so that for a given readiness level j, it requires all the time objects associated with readiness levels i with $i \leq j$. Under the assumed implication relationship between the readiness levels, to process the query HOLDS(o, j, t) for an aircraft o and the readiness level j, we have to return the union of the vectors v_i for all $i \leq j$.

6. Concluding Remarks

Here, we present an object-oriented temporal data model, named OOTDM, for information and database systems to store and process temporal information. The model features with elaborated classes for modeling time and a conceptual framework for integrating time objects with entities and attributes. In a time object, we encapsulate not only the values for identifying the spatial position of the time but also powerful and flexible functions for calculating the length, relative positions, and set operations of time objects. Both the conceptual model and the internal view of OOTDM follow the pervasive metaphor of time. The class of time objects can be easily integrated into any object-oriented software system. The model OOTDM

can be used to design object-oriented database systems.

Object-orientation helps the model OOTDM define data and declare operations. We demonstrate how to apply the object-oriented concepts encapsulation, inheritance, polymorphism, and other fundamental notions of object-oriented paradigm to define time objects and to integrate the time objects with entity objects and attributes. The temporal data model is also suitable to object-oriented query processing. The temporal model OOTDM makes it possible for an object-oriented database system to be extended with temporal attributes. The object-oriented approach uses inheritance to generalize ordinary attributes into temporal ones. Thus, an existing object-oriented database can be easily adapted to include temporal attributes. The data model makes it possible to take all the benefits of object-oriented software engineering, which include reducing development time and enhancing reliability (Meyer 1992).

We conclude that the object-oriented paradigm makes it possible for us to develop the data model OOTDM to support powerful and flexible database systems that store temporal information. The data model is extensible. It is not based on other data models. In fact, it can coexist or be integrated with other object-oriented data models in object-oriented database systems.

References

(Aaronson 1972) B. AARONSON. Time, time stance, and existence. In *The Study of Time*, Fraser *et al.*, editors. Springer–Verlag, New York, 1972, pp. 293–311.

(Aho et al. 1974) A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN. *The Design and Analysis of Computer Algorithms*. Addison–Wesley Publishing Co., Reading, Mass, 1974.

(Allen 1983) J. F. ALLEN. Maintaining knowledge about temporal intervals. *Communications of the ACM*, Vol. 26, 1983, pp. 832–843.

(Allen 1984) J.F. ALLEN. Towards a general theory of action and time. *Artificial Intelligence*, Vol. 23, 1984, pp. 123–154.

(Andrews and Harris 1987) T. ANDREWS AND C. HARRIS. Combining language and database advances in an object-oriented development environment. In *Proceedings of 2nd International Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Orlandoo, FL. Oct. 1987).

(Ariav and Clifford 1984) G. ARIAV AND J. CLIFFORD. A system architecture for temporally oriented data management. In *Proceedings of International Conference on Management on Information Systems* (Tucson, Ariz. Nov. 1984), pp. 177–186.

(Ariav 1986) G. ARIAV. A Temporally oriented data model. *ACM Transactions on Database Systems*, Vol. 11, 1986, pp. 499–527.

(Arlein et al. 1994) R. ARLEIN *et al.* *Ode 3.X User Manual*. AT&T Lab, Murray Hill, New Jersey, 1994.

(Coad and Nicola 1993) P. COAD AND J. NICOLA. *Object-Oriented Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

(Dillon 1994) L. K. DILLON, *et al.* A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodologies*, Vol. 3, No. 2, April 1994, pp. 131–165.

(Ellis and Stroustrup 1990) M. A. ELLIS AND B. STROUSTRUP. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, Massachusetts, 1990.

(Goldberg and Robson 1989) A. GOLDBERG AND D. ROBSON. *Smalltalk–80, the Language*. Eddison Wesley, Reading, Massachusetts, 1989.

(Headington and Riley 1994) M. R. HEADINGTON AND D. D. RILEY. *Data Abstraction and Structures Using C++*. D. C. Heath and Company, Lexington, Massachusetts, 1994.

(Kim 1990) W. KIM. *Introduction to Object-Oriented Databases*. The MIT Press, Cambridge, Massachusetts, 1990.

(Kafer and Schoning 1992) W. KAFER AND H. SCHONING. Realizing a temporal complex-object data model. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management on Data*. (San Diego, California, June 1992), pp. 266–275.

(Meyer 1992) B. MEYER. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, Hertforshire, UK, 1992.

(Mitschang 1988) B. MITSCHANG. Towards a unified view of design data and knowledge representation. In *Proceedings of the Second International Conference on Expert Database Systems*. (Tysons Corner, 1988), pp. 33–50.

(Maiocchi et al. 1992) R. MAIOCCHI, B. PERNICI, AND F. BARBIC. Automatic deduction of temporal information. *ACM Transactions on Database Systems*, Vol. 17, No. 4, 1992, pp. 647–688.

(Object 1990) OBJECT DESIGN. *ObjectStore Reference Manual*. Versant Object Technology Inc, Burlington, MA, 1990.

(Rumbaugh et al. 1991) J. RUMBAUGH *et al.* *Object-Oriented Modeling and Design.* Prentice Hall, Englewood, New Jersey, 1991.

(Sciore 1991) E. SCIORE. Using annotations to support multiple kinds of versioning in an object-oriented database system. *ACM Transactions on Database Systems*, Vol. 16, No. 2, 1991, pp. 417–4381.

(Snodgrass 1995) R. SNODGRASS. Temporal object-oriented databases: a critical comparison. In *Modern Database Systems–The Object-Model, Interoperability, and Beyond*, W. Kim, editor, ACM Press, 1995, pp. 386–408.

(Van Benthem 1982) J. F. K. VAN BENTHEM. *The Logic of Time.* D. Reidel Publishing Co., Hingham, Mass, 1982.

(Versant 1990) VERSANT OBJECT TECHNOLOGY. *Versant Reference Manual.* Versant Object Technology Inc, Menlo Park, CA, 1990.

Received: August, 1995
Accepted: November, 1995

Contact address:
Liwu Li
School of Computer Science
University of Windsor
Windsor, Ontario
Canada N9B 3P4
e-mail: liwu@cs.uwinsdor.ca
tel: (519) 253–4232 ext 2994
Fax: (519) 973–7093

LIWU LI holds a Ph.D. of Computing Science from University of Alberta. He is with School of Computer Science, University of Windsor. His research interests include formal specification and logical foundation for object-oriented software engineering, object oriented database systems, and nonmonotonic reasonong.