An Approach to Model and Query Event-Based Temporal Data

Elisa Bertino, Elena Ferrari
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano - Italy
{bertino,ferrarie}@dsi.unimi.it

Giovanna Guerrini
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova - Italy
guerrini@disi.unige.it

Abstract

Temporal database systems support all functions related to the management of large amounts of constantly changing data. However, current temporal database systems support a flat view of the history of data changes, in that all the changes are considered equally relevant and are, therefore, all stored in the database. However, many applications, such as monitoring and planning applications, call for more flexibility. Monitoring applications, in particular, may require that the history of a data item is stored only whenever a certain event occurs. For other applications the history of data changes may be less important than the event causing the changes. In this paper we propose an event-based temporal object model which allows to keep track of selected values within the history of a data object attribute. The portions, within the history of a data object, which are actually stored into the database are identified by relating them to events. In the paper, besides defining the data model, we investigate the problem of querying a database with incomplete temporal information.

1 Introduction

A temporal database [13] typically stores the entire history of data over time, rather than storing only current data as conventional databases. Some models support, in addition, static attributes [3], that is, attributes for which the changes in time are not meaningful with respect to the given application domain and, thus, need not to be stored. Such approach avoids storing irrelevant old values of attributes [12]. There are, however, many applications, such as monitoring and planning applications, for which it would be useful to selectively record values of data objects, that is, to store the past values of data objects only whenever certain situations occur or certain conditions are met. Monitoring applications may require that whenever a certain event occurs on a certain data object, the history of the data object (or of some of the object's attributes) is recorded in the database from that moment on. Consider as an example the price of a stock; a monitoring application could need to record all the price variations of the stock only when the price of the stock increases over a given amount. Similar examples can be found in the medical application domain, where for example the temperature variations for a patient need only to be recorded when the temperature becomes higher than 36C degrees. Planning applications may require to keep the history of a data object after certain modifications on the object have been performed, in order to subsequently analyze the consequences of the modifications. Object-oriented database systems supporting object evolution may also require a selective recording of object histories [5]. For example, it could be meaningful to store the value of the salary of an employee when he is promoted to a manager, or when he retires, or when he works in a certain division or on a certain project. As pointed out by Dean and McDermott in [9], time can be seen as a map, where not all instants or time intervals are equally relevant. Some instants or time intervals may be more important than others, as they indicate relevant application events.

Current temporal DBMSs do not provide support for event-based temporal histories. One possibility would be supporting such histories through application programs on top of a temporal DBMS. Such an approach, however, has many drawbacks. The applications would have to be in charge of detecting events and determining when an old value of a data item must be kept or must be purged. As with integrity constraints, relying on application programs makes it difficult, if at all possible, the specification of events and

the management of event-based temporal histories. Versioning mechanisms suffer from the same drawbacks because it is up to the users or applications to explicitly require the creation of a new version of a given data object. Therefore, no information is kept into the database concerning events that caused the new version to be created (unless some specific code for doing so is added to all application software). A relevant issue in a data model supporting selective temporal histories is related to queries. Only queries accessing attributes at time instants in which a value for that attribute is stored can be *exactly* solved. The other queries can be rejected, or alternately, can be *approximated* by using an appropriate approximation method. Such issue, however, has not been so far addressed.

In this paper we develop an event-based temporal object data model by extending the T_Chimera temporal object model [2] with the possibility of selectively storing the past values of object attributes by associating with attributes *snapshot conditions*, expressed in a rich event language. When the snapshot condition associated with an attribute occurs, the history of the attribute is updated by inserting the current value of the attribute. The notion of snapshot condition and its use in event-based temporal histories has been first proposed in [3]. However, in [3] a very simple language for expressing snapshot conditions was considered. In this paper, we consider a powerful event language, allowing to combine database operations, conditions on the database, temporal and periodic events by means of several operators. Our event specification language has almost the same expressive power of the event specification language Snoop [8]. The main difference is that we provide a more powerful formalism to express periodic time, since in our language expressions with level 2 periodicity can be specified, like *Each working day between 9 a.m. and 12 a.m.*

Besides defining the data model, we address the issue of queries, which was not dealt with in [3]. We consider temporal attribute accesses requiring the value of a certain attribute for an object at a specified time, and we investigate conditions ensuring that the access can be exactly solved. For accesses that cannot be exactly solved, we introduce the notion of approximate query, to supply a value obtained through an *approximation method*. Moreover, we show how our event language support *meta-queries*, that is, queries to retrieve all the objects or attributes whose changes have been recorded because of an arbitrary event occurrence. This is a relevant issue since often knowing the event which has changed the state of an object is more important than the change itself. Work on this direction has been carried on in the context of relational databases [10]. However, to our knowledge, we are the first addressing these issues in the context of the object-oriented model.

The remainder of this paper is organized as follows. Section 2 introduces the event specification language. Section 3 deals with event containment. Section 4 presents the data model, whereas Section 5 deals with queries. Finally, Section 6 concludes the paper and outlines future work. The event language syntax and semantics are reported in Appendix A.

2 Event Language

In this section we give an informal description of the language we provide to specify events. The formal syntax and semantics are reported in Appendix A.

The language supports the specification of two main categories of events: *basic events* and *composite events*, built by applying a set of predefined *event constructors* to basic or composite events. Basic events are of two main types: *database events* and *temporal events*. Database events denote database operations such as the migration of an object from a class to another or the increase of an attribute value up to a given threshold. Database events can be either *instantaneous* (i.e., they last only one instant), or *persistent* (i.e., they last over a time period). For instance, `migrate` is an instantaneous event, whereas `(salary > 60K)` is a persistent event, since it lasts from the first instant on which the salary exceeds `60K` till the first instant on which the salary becomes again lesser than `60K`. We use the term *starting time* to denote the instant on which an event begins to occur. Similarly, the *ending time* of an event is the time on which the event finishes. We use the notation `mm/dd/yy:hh:mm:ss` to represent time instants. For example, the notation `1/1/94:08` represents `8 a.m.` on `1/1/94`. When `1/1/94` is used as a minimum, it denotes the first instant of the first day of January 1994, while, as a maximum, it denotes the last instant of `1/1/94`.

Note moreover that an event can occur more than one time during the database history.

Example 2.1 Consider the event `(salary > 60K)` and suppose that `salary = 30K` during the period `[1/1/94,1/1/95]`. Suppose that on `1/2/95` the salary increases up to `70K` and remains unchanged till `1/1/96` when it assumes the value `20K`. Suppose moreover that the salary remains unchanged till `4/1/96` when it is set equal to `80K`. Therefore the event `(salary > 60K)` occurs from the first instant of `1/2/95` to the last instant of `12/31/95` and from the first instant of `4/1/96` up to now. The first instants of `1/2/95` and `4/1/96` represent the starting times of the event `(salary > 60K)`, whereas the last instant of `12/31/95` represents the ending time of the event.

Temporal events can be either *absolute*, as the first hour of a specific day, *periodic*, as each working-day between 9 a.m. and 12 a.m., or *relative*, as five hours after the salary

of a given employee has been modified. To represent periodic time we adopt the symbolic formalism proposed by Niezette and Stevenne in [11], based on the notions of *calendar* and *periodic expression*. We postulate the existence of a set of predefined calendars containing at least the calendars *Hours*, *Days*, *Weeks*, *Months*, and *Years*, and we consider *Hours* as our finest granularity calendar. Calendars are combined to represent more general sets of periodic intervals, not necessarily contiguous, as, for instance, the set of *Mondays* or the set of *The third hours of the first day of each month*. These complex sets of periodic intervals are represented by means of *periodic expressions*. Table 1 illustrates a set of periodic expressions and their meaning. More details can be found in [11].

Moreover, the language provides a set of event constructors from which complex events can be defined.

A first set of constructors, called *event modifiers*, consists of a number of unary operators that transform an arbitrary event into one or more related events. We support four different types of event modifiers. Let E be an event, `begin`(E) and `end`(E) occur on the starting and ending times of E, respectively; `begin_on`(E) occurs from the first occurrence of E up to now, whereas `end_on`(E) occurs from the first time E ends up to now.

Example 2.2 With reference to Example 2.1, `begin(salary > 60K)` occurs on `1/2/95` and `4/1/96`, `end(salary > 80K)` occurs on `12/31/95`, `begin_on(salary > 60K)` occurs from `1/2/95` up to now, whereas `end_on(salary > 60K)` occurs from `12/31/95` up to now.

Additionally, the language provides the following constructors (in the following, E_1 and E_2 denotes two events):

- *Disjunction*. E_1 OR E_2 occurs when E_1 or E_2 occurs;
- *Conjunction*. E_1 AND E_2 occurs when both E_1 and E_2 occur;
- *Sequence*. This constructor conditions the occurrence of an event to the temporal relations occurring between the starting times of two arbitrary events. We provide four different forms of the sequence constructor: `sequence`(E_1, E_2,$\mathtt{t}_{min}$,$\mathtt{t}_{max}$), which occurs on the starting time of E_2, provided that the ending time of E_1 occurs $\mathtt{t}_{max}$ - $\mathtt{t}_{min}$ instants before the starting time of E_2;[1] `sequence`(E_1, E_2,$\mathtt{t}_{min}$,`null`) which occurs at the starting time of E_2, provided that the ending time of E_1 occurs at least $\mathtt{t}_{min}$ instants before the starting time of E_2; `sequence`(E_1, E_2,`null`,$\mathtt{t}_{max}$) which occurs at the starting time of E_2, provided that the ending time of E_1 occurs no more than $\mathtt{t}_{max}$ instants before the starting time of E_2; `sequence`(E_1, E_2,`*`) which occurs on the starting time of E_2, provided that E_1 has already occurred.
- *Occurrence*. This constructor relates the occurrence of an event to the multiple occurrences of another event in a set of time intervals bound by the starting/ending times of two arbitrary events. There are two variants of the occurrence constructor: `happen`(E_1, n, E_2, E_3) occurs on the starting time of the n-th occurrence of event E_1 in the closed interval determined by the starting time of E_2 and the ending time of E_3; `happen`($E_1, *, E_2, E_3$) occurs on the starting time of any occurrence of event E_1 in the closed interval determined by the starting time of E_2 and the ending time of E_3.
- *Non-occurrence*. This constructor relates the occurrence of an event to the non occurrence of another event in a set of time intervals bound by the starting/ending times of two arbitrary events. There are two variants of the non-occurrence constructor: `not_happen`(E_1, n, E_2, E_3) occurs on the ending time of E_3 provided that the number of occurrences of E_1 in the closed interval determined by the starting time of E_2 and the ending time of E_3 is lesser than n; `not_happen`($E_1, *, E_2, E_3$) occurs on the ending time of E_3 provided that E_1 does not occur in the closed interval determined by the starting time of E_2 and the ending time of E_3.

In the following, $\mathcal{E}$ denotes the set of events that can be expressed in our language.

Example 2.3 The following are events in $\mathcal{E}$:
`generalize`: it occurs each time an object in the database migrates to a superclass;
`salary > 60K` AND `status =''part time''`: it occurs each time the salary of a part time employee is greater than `60K`;
`([1/1/94,`∞`],Mondays)`: it occurs each Monday from `1/1/94`;
`happen(increase(salary), 5, status = ''part time'', status = ''full time'')`: it occurs if the salary of an employee has increased 5 times during the period he was a part time employee.

3 Containment between Events

In this section we investigate the property of event containment, which will be useful for characterizing a number of conditions in our temporal data model. Intuitively, given

[1] Note that, as a particular case, `sequence`(E_1, E_2,`0`,`0`) occurs on the starting time of E_2, provided that the starting time of E_2 is immediately after the ending time of E_1.

periodic expression	meaning
Weeks + {2,6}.*Days*	Mondays and Fridays
Months + 20.*Days*	The twentieth day of every month (Pay-days)
Years + 7.*Months* ▷ 3.*Months*	Summer-time
Weeks + {2,. . .,6}.*Days* + 10.*Hours* ▷ 3.*Hours*	Each Working day between 9 a.m. and 12 a.m.

Table 1. Example of periodic expressions

two events E and E', E is contained in E' if each time E occurs, E' occurs too.

To formally define the notion of containment, we need first to introduce some preliminary notations.

We model the database as an history

$$db\_history = \langle [t_0, t_1), db_0 \rangle \longrightarrow^{E_1} \langle [t_1, t_2), db_1 \rangle \longrightarrow^{E_2} \ldots \ldots \longrightarrow^{E_n} \langle [t_n, now], db_n \rangle$$

where, t_i, $i = 1, \ldots, n$, is a time instant, E_i, $i = 1, \ldots, n$, is an update event which arose at time t_i and *now* denotes the current time. The *db_history* models that the database has evolved from its initial state db_0 through a sequence of states db_i and each transition arises because of an update event E_i. Moreover, we make use of a function $f : \mathcal{E} \times \mathcal{TIME}^2 \rightarrow \{true, false\}$ that, given an event $E \in \mathcal{E}$ and a time instant *t*, returns *true* if E occurs on *t*; it returns *false* otherwise. This function, modeling event semantics, is specified in Appendix A.

Definition 3.1 Let E and E' be two events in $\mathcal{E}$. E is *contained* in E', denoted as $E \subseteq E'$, if and only if $\forall db\_history$: $\{t \mid f(E, t) = true\} \subseteq \{t' \mid f(E', t') = true\}$.

Example 3.1 The following are examples of containment relationships between events:

- {[1/1/94,1/1/95],[1/1/96,1/2/97]} ⊆ {[1/1/93,1/2/97]};
- `salary > 60K` AND `status = ''part time''` ⊆ `salary > 60K` ⊆ `salary > 20K`.

An algorithm to test containment between events in $\mathcal{E}$ has been developed.

4 Data Model

The temporal data model we propose allows to selectively keep track of data modifications. The idea is to associate a *snapshot condition*, expressed in the language introduced in Section 2, with an attribute, so that the history of the attribute is updated whenever the snapshot condition is true. The data model we propose here extends the object-oriented temporal data model T_Chimera [2] with the possibility of associating snapshot conditions with attributes. However, the idea of selectively keeping track of modifications to data is highly independent from T_Chimera and can be applied to any object-oriented or relational temporal data model.

In the following, we denote with $\mathcal{OI}$ a set of object identifiers, with $\mathcal{CI}$ a set of class identifiers (i.e., class names), with $\mathcal{AN}$ a set of attribute names, and with $\mathcal{V}$ the set of T_Chimera legal values.

T_Chimera supports the notion of *type*. A finite set of basic predefined types is provided by the language, containing in addition to the usual (nontemporal) types, a `time` type whose domain is the set $\mathcal{TIME}$ = {0,1,. . .,*now*} and which represents transaction time. T_Chimera also supports *structured types* such as sets, lists and records, and allows the use of class names in the definition of structured types. Finally, *temporal types* are supported: for each type T, a corresponding temporal type, $temporal(T)$, is defined. Instances of type $temporal(T)$ are partial functions from instances of type `time` to instances of type T. Temporal types can be used in the definition of structured types.

In our data model, attributes have temporal types as domains and their values are thus partial functions from $\mathcal{TIME}$ to the set of legal values for the attribute.[3] Throughout the paper we represent the value of a temporal attribute of type $temporal(T)$ as a set of pairs $\{\langle \tau_1, v_1 \rangle, \ldots, \langle \tau_n, v_n \rangle\}$, where $v_1, \ldots, v_n \in \mathcal{V}$ are legal values for type T, and $\tau_1, \ldots, \tau_n \in \mathcal{TIME} \times \mathcal{TIME}$ are time intervals such that the attribute assumes the value v_i for each time instant in τ_i, $i = 1, \ldots, n$. Moreover, given the value v of a temporal attribute and a time instant *t*, $v(t)$ denotes the value taken by function v on input *t*, according to the usual notation for function application. A snapshot condition in $\mathcal{E}$ can be associated with each attribute, specifying the conditions upon which the attribute value is stored. If no snapshot condition is associated with an attribute, the entire history of attribute changes is recorded in the database. Given a class $c \in \mathcal{CI}$, $A(c)$ denotes the set of attributes of instances of that class, whereas $dom(a, c)$ and $\varepsilon(a, c)$ denote the domain and the snapshot condition

[2] The set $\mathcal{TIME} = \{0, 1, \ldots, now\}$ is our temporal domain.

[3] Actually, T_Chimera supports temporal, static and immutable attributes. In this paper we only consider temporal attributes.

of attribute a in class c, respectively.

Example 4.1 Class `employee` and its subclass `manager` are examples of T_Chimera classes:
A(employee) = {name, salary, status, division, manager, w_hours},
dom(name,employee) = temporal(string),
dom(salary,employee) = temporal(integer),
dom(status,employee) = temporal(string),
dom(division,employee) = temporal(string),
dom(manager,employee) = temporal(manager),
dom(w_hours,employee) = temporal(integer),
ε(salary,employee) = (create,∞, Pay-Days),
ε(manager,employee)= sequence(decrease(salary), increase(w_hours),*)

A(manager) = A(employee)$\cup${dependents, official_car},
dom(dependents,manager) = temporal(set-of(employee)),
dom(official_car,manager) = temporal(string)
ε(dependents,manager) = count(dependents) $\geq$ 5,
ε(official_car,manager) = update(salary)

The value of attribute `salary` is recorded the twentieth day of every month (i.e., the "pay-day") starting from the time of the object creation; the history of attribute `manager` is updated only upon a salary reduction followed by an increase of the working hours. Manager dependents are recorded only when the dependents are more than five, whereas manager official car is recorded only whenever the manager salary is updated. For all the other attributes, the entire history of changes is maintained.

For a proper redefinition of attributes in subclasses, we impose that: (*i*) the domain of an attribute in a subclass is refined in a subtype of the domain in the superclass; (*ii*) the snapshot condition associated with an attribute in a subclass must contain the snapshot condition associated with the attribute in the superclass. These conditions, formally stated by the following rule, ensure the substitutability of the subclass instances with respect to the superclass.

Rule 4.1 Let c_1 and $c_2 \in \mathcal{CI}$ such that c_2 is a subclass of c_1. Then, $\forall a \in A(c_1)$, the following conditions must be satisfied: *i)* $dom(a, c_2) \leq_T dom(a, c_1)$; *ii)* $\varepsilon(a, c_1) \subseteq \varepsilon(a, c_2)$, where $\leq_T$ denotes the subtype relationship [2].

An object o is characterized by an object identifier i, a lifespan and a record value which represents its state, as formally stated by the following definition.

Definition 4.1 An *object* o is a 3-tuple $(i, lifespan, v)$, where $i \in \mathcal{OI}$ is the oid of o; $lifespan \in (\mathcal{TIME} \times \mathcal{TIME})$ is the lifespan of o; $v \in \mathcal{V}$ is a record value $(a_1 : v_1, \ldots, a_n : v_n)$, where $a_1, \ldots, a_n \in \mathcal{AN}$ are the names of the attributes of o, and $v_1, \ldots, v_n \in \mathcal{V}$ are their corresponding values.

Example 4.2 Consider the classes of Example 4.1, and suppose that $\mathtt{i}_1, \ldots, \mathtt{i}_7 \in \mathcal{OI}$. Objects $\mathtt{o}_1$ and $\mathtt{o}_2$ specified as follows are examples of T_Chimera objects :[4]

i = $\mathtt{i}_1$
$lifespan$ = [1/1/97,now]
v = {(name: {⟨[1/1/97,now],'Alan Smith'⟩}),
(salary: {⟨1/20/97,20K⟩,⟨2/20/97,15000⟩}),
(status: {⟨[1/1/97,now],'full-time'⟩}),
(division: {⟨[1/1/97,1/31/97],'Disks'⟩,
⟨[2/1/97,now],'Printers'⟩}),
(manager: {⟨3/1/97,$\mathtt{i}_4$⟩}),
(w_hours: {⟨[1/1/97,2/28/97],38⟩,
⟨[3/1/97,now],40⟩})}

i = $\mathtt{i}_4$
$lifespan$ = [2/1/97,now]
v = {(name: {⟨[2/1/97,now],'Mary Dole'⟩}),
(salary: {⟨2/20/97,30K⟩}),
(status: {⟨[2/1/97,now],'full-time'⟩}),
(division: {⟨[2/1/97,now],'Printers'⟩}),
(w_hours: {⟨[2/1/97,now],35⟩}),
(dependents: {⟨[2/15/97,now],
{$\mathtt{i}_1, \mathtt{i}_3, \ldots, \mathtt{i}_7$}⟩})}

Note that, no value for attributes `manager` and `official_car` is recorded in object $\mathtt{o}_2$, since there does not exist an instant in which the corresponding snapshot conditions are verified.

In order to ensure object consistency, we require that for each attribute of an object, a value is stored for each instant satisfying the associated snapshot condition. This requirement is formalized by the following rule.

Rule 4.2 Let o be an object, and let c be the class to which o belongs at time t. Then, $\forall a \in A(c)$ the following condition must be satisfied: $o.v.a(t)$ is defined $\Leftrightarrow f(\varepsilon(a, c), t) = true$.

Moreover, the value must be of the appropriate type, as usually in data models.

5 Queries

Because in our model, only selected portions in an attribute's history can be recorded, a query could be issued requiring the value of the attribute at a time belonging to a non-recorded portion. An important question concerns which accesses can be exactly answered and how to provide a value also for the time instants for which no value exists in the database. In this section we address such issues.

For the sake of simplicity we do not introduce a complete query language. Rather, we focus on temporal attribute accesses, which are the basis of any object-oriented query language. We consider temporal attribute accesses of the form $o.a \downarrow E$, where o is an object reference (e.g.

[4] In the example, by abuse of notation, we use t to denote the interval $[t, t]$.

a variable) denoting an object of a given class c (e.g. the type of the variable), a is an attribute of class c, and E is a *temporal specification*, that is, an expression specified in the event language introduced in Section 2. Note that other languages could have been considered as well, but we refer to the event language both for the sake of uniformity and because it is convenient for expressing *meta-queries*.[5] However, we restrict ourselves to event expressions denoting single time instants, though temporal attribute accesses involving time intervals could be easily handled [4]. We do not consider them here since they complicate the discussion without bringing in any relevant issue. Thus, we consider the subset of events introduced in Section 2 which occur on specific time instants (that is, instantaneous events). This means that we only consider event expressions E such that $\{t \mid f(E,t) = true\}$ is a singleton set $\{\bar{t}\}$. Given an event E, let t_E denote this unique time instant. Finally, note that, as a particular case, E can be a time instant t (i.e., the time of the attribute access can be explicitly denoted).

Example 5.1 Given a variable `X` of type `employee`, `X.salary`$\downarrow$`1/1/94:08` is an example of temporal attribute access requiring the value of attribute `salary` of the employee object denoted by variable `X` at `8 a.m.` on `1/1/94`. A further example of temporal attribute access is `X.salary` $\downarrow$ `end(salary > 80K)`. Referring to the $db\_history$ of Example 2.1, the above temporal attribute access is equivalent to the access `X.salary` $\downarrow$ `12/31/95`.

5.1 Exact Queries: Static Conditions

In this subsection we consider a temporal attribute access $o.a \downarrow E$ and we deal with the problem of (statically) deciding whether the value of attribute a of the object denoted by o is available at the time denoted by E. Intuitively, an attribute access $o.a \downarrow E$ can be exactly solved, that is, a value for attribute a of object o at time t_E is available, if the snapshot condition of attribute a in the class of o, is true at time t_E. The value denoted by $o.a \downarrow E$, if available, is the non-temporal value $o.v.a(t_E)$, that is, the value of the (partial) function $o.v.a$ on t_E. Thus, given an object reference o of type c and an attribute a of class c, the attribute access $o.a \downarrow E$ denotes an available value, and, thus, can be exactly answered, if and only if $f(\varepsilon(a,c), t_E) = true$.

Example 5.2 Consider the objects of Example 4.2. Let `X` be a variable of type `employee` and `Y` be a variable of type `manager`, denoting object $\mathtt{o}_1$ and $\mathtt{o}_2$, respectively. The attribute accesses `X.name` $\downarrow$ `2/1/97`, `X.salary` $\downarrow$ `1/20/97`, `X.manager` $\downarrow$ `3/1/97`, `Y.salary` $\downarrow$ `2/20/97` and `Y.dependents` $\downarrow$ `2/16/97` can be exactly answered and they denote the values `Alan Smith`, `20K`, $\mathtt{i}_4$, `30K`, and $\{\mathtt{i}_1, \mathtt{i}_3, \ldots, \mathtt{i}_7\}$, respectively. By contrast, the attribute access `X.salary` $\downarrow$ `1/21/97` cannot be exactly answered since the event `(create,`∞`,PayDays)` did not occur on `1/21/97`.

If the value of attribute a for object o at the time denoted by E is not available, the query could be rejected, or could be approximated. Query approximation will be dealt with in the following subsection. However, a user can explicitly require that the query must be exactly answered. Determining whether a query cannot be exactly answered, without executing the query, would avoid many unnecessary database accesses. It is however not always possible to statically detect whether an attribute access can be exactly solved. For instance, if $\varepsilon(a,c)$ is a database event (either an update event or a value event) and E is a time instant t we cannot decide at query compile-time whether $f(\varepsilon(a,c), t)$ is true, since this decision depends on the specific $db\_history$.

Example 5.3 Given the database event `salary > 60K` and the time instant `1/1/94:08` we cannot decide, independently from the $db\_history$, whether $f($`salary > 60K,1/1/94:08`$) = true$.

More precisely, we can ensure that an attribute access $o.a \downarrow E$ can be exactly answered, if E is contained[6] in the snapshot condition associated with a, that is, if $E \subseteq \varepsilon(a,c)$. In this case, indeed, by definition of event expression containment, for each $db\_history$: $t_E \in \{t \mid f(\varepsilon(a,c)) = true\}$.

Example 5.4 The temporal attribute access `X.salary` $\downarrow$ `begin(salary > 60K)` can be exactly answered in a database where the snapshot condition `salary > 60K` is associated with attribute `salary` in class `employee` since `begin(salary > 60K)` $\subseteq$ `salary > 60K`.

5.2 Approximate Queries: Approximation Methods

In this subsection we deal with attribute accesses that cannot be exactly solved, that is, accesses of the form $o.a \downarrow E$ when the value of attribute a of the object denoted by o is not available at time t_E. This can be a common situation in our model, since we allow a partial recording of attribute temporal histories. When the value at a given instant is not available, different options can be taken with respect to which value should be returned. The most intuitive and easy solution is to return a null value. However, several situations can be devised in which it could be more appropriate to return an *approximate* value. For instance, suppose that a user requests the salary of a given employee

[5] We will elaborate on this in Subsection 5.3.

[6] Containment between events has been discussed in Subsection 3.

$$wavg() = v \leftarrow v = [(o.a \downarrow t, last()) * (t - time(o.a \downarrow t, last())) + (o.a \downarrow t, next()) * (time(o.a \downarrow t, next()) - t)] / (time(o.a \downarrow t, next()) - time(o.a \downarrow t, last())).$$

Figure 1. Example of approximation method

at time t and suppose that the salary is recorded in the database once a month. If no salary amount is stored at time t, it is reasonable to return the latest stored value for the salary attribute. This is equivalent to assume that the value of the salary attribute persists in time until a new value is explicitly stored in the database [6]. Other alternatives could be returning the next explicitly given value, or the average of the last and the next explicitly given values. More sophisticated approximation methods can be adopted, such as returning the average of the last n values.

We therefore allow an approximate value to be returned for queries that cannot be exactly solved. This is achieved by assuming the existence of a pre-defined set $\mathcal{AM}$ of *approximation methods* which can be used in a query. Such methods define how to derive implicit information from that explicitly stored. Approximation methods are functions that given an instant t and an attribute a of type T return a value of type T representing the value to be returned as the value of a at time t, when the value of a at time t is not available. This value is computed by appropriately combining the available values for attribute a. We assume that $\mathcal{AM}$ contains at least the method *last*() which given an instant t and an attribute a returns the more recent value among those stored before time t, the method *next*() which returns the first value for a, if any, stored after time t, and the method *avg*() which returns the average of the values returned by *last*() and *next*().

Attribute accesses for which an approximate value should be returned are called *approximate attribute accesses* and are formally defined as follows.

Definition 5.1 An *approximate attribute access* is a pair ($o.a \downarrow E$,*app*), where o is an object reference of type c, a is an attribute of class c, E is a temporal specification, and *app* is an element of $\mathcal{AM}$.

Example 5.5 Suppose that the value of attribute salary is `60K` on `12/30/93` and `80K` on `2/1/94`. Suppose moreover that no other value for attribute salary is recorded in the period `[12/30/93,2/1/94]`. Then the answer to (`X.salary`$\downarrow$`1/1/94:08`,$last()$) is `60K`, the answer to (`X.salary`$\downarrow$`1/1/94:08`,$next()$) is `80K`, whereas the answer to (`X.salary`$\downarrow$`1/1/94:08`,$avg()$) is `70K`.

Note moreover that user-defined approximation methods can be specified as well, in addition to methods provided by the system. User-defined approximation methods are used to specify ad-hoc approximation methods for certain attributes and accesses. Though several languages could be used to express them, we consider approximation methods expressed in a deductive style, that is, through rules which may contain in their bodies temporal attribute accesses and the predefined approximation methods $last()$, $next()$ and $avg()$.

Example 5.6 The approximation method $wavg()$ illustrated in Figure 1 computes a weighted average of the values returned by approximation methods $last()$ and $next()$. It makes use of a function $time$ that, applied to an approximate access, returns the time instant at which the access is approximated.

Approximation methods which can appear in an approximate attribute access depend on the type of the attribute. Approximation methods denoting aggregate functions, such as $avg()$, make sense only when they apply to numerical values, such as an `integer` or a `real`; in the case of non-numerical values, such as object identifiers, or for structured values containing non-numerical components, approximation methods that can be applied are those returning one of the stored values for the attribute, like for instance $last()$ and $next()$. Moreover, approximation methods also depend on the semantics of the attribute. For instance, it is reasonable to associate with attribute `status` of Example 4.1 the approximation method $last()$, since the status of an employee could be reasonably assumed to be the last status recorded, whereas the approximation method $wavg()$ of Example 5.6 could reasonably be associated with attribute `salary` of Example 4.1, if we assume that the salary of an employee can be approximated by a linear function.

5.3 Meta-queries

In the above subsections we have dealt with queries requiring the value of an attribute at a given time instant. However, our language also support *meta-queries*. A meta-query contains conditions on event occurrences. In particular, we support two different types of meta-queries:

1. *Attribute meta-queries* that, given an event and an object, return all the attributes of the object whose temporal histories have been updated because of the event occurrence.

2. *Object meta-queries* that, given an event, retrieve all objects whose state has been modified because of the event occurrence.

Note that two different interpretations are possible for the above meta-queries. Consider an attribute meta-query on an event E. Under a *strong* interpretation, the query retrieves only the attributes of the specified object with E as snapshot condition. Under a *weak* interpretation the query retrieves all the object attributes whose associated snapshot condition contains E and such that there exists at least an instant, among the ones for which a value for the attribute is stored, in which E occurs. A similar distinction applies to object meta-queries.

Example 5.7 Consider object $\mathtt{o}_2$ of Example 4.2, and the attribute meta-query requiring all the attributes of $\mathtt{o}_2$ whose temporal histories have been recorded because of the occurrence of the event `count(dependents)` $\geq$ `6`. Under a strong interpretation, the query does not return any value, since there is no attribute of object $\mathtt{o}_2$ with `count(dependents)` $\geq$ `6` as snapshot condition. By contrast, under a weak interpretation the query returns the attribute `dependents`, since the snapshot condition `count(dependents)` $\geq$ `5`, associated with attribute `dependents`, contains the event specified in the query, and $\forall t \in$ `[2/15/97,now]` f(`count(dependents)`$\geq$`6`,t) $= true$. Finally, if the event associated with the meta-query is `count(dependents)` $\geq$ `8`, no attribute is returned under both the weak and the strong interpretation.

In what follows the weak interpretation is always assumed. However, the treatment can be easily extended to the strong interpretation.

The notion of meta-query is formalized by the following definitions.

Definition 5.2 Let o be an object of type c, and let E be an event. The *attribute meta-query* $o \mid\mid E \mid\mid$ returns all the attributes $a \in A(c)$ which satisfy the following conditions: *i)* $E \subseteq \varepsilon(a, c)$; *ii)* $\exists t$ such that $f(E, t) = true$.

Definition 5.3 Let o be an object, and let E be an event. The *object meta-query* $\mid\mid E \mid\mid$ returns all the objects o such that $o \mid\mid E \mid\mid \neq \emptyset$.

Example 5.8 Consider the objects of Example 4.2 and suppose that they are the only objects in the database. $\mathtt{o}_1$ ||`sequence(decrease(salary),increase(w_hours),*)`|| = `manager`, $\mathtt{o}_2$ || `count(dependents)` $\geq$ `6` || = `dependents`, || `count(dependents)` $\geq$ `5` || = $\mathtt{o}_2$.

6 Conclusions and Future Work

In this paper we have presented an event-based temporal object data model which allows to record selected portions within the history of an object attribute. The portions which are actually stored are identified by relating them to events. We have also investigated the problem of querying a database with incomplete temporal information.

We plan to extend this work along several directions. First, we are currently developing a complete query language based on our data model. The language, defined as a temporal extension of the OQL language [7], will also support meta-queries. We also plan to extend the current model to support multiple temporal granularities and integrity constraints. Finally, implementation issues are being investigated; in particular, we are implementing the proposed model on top of the Ode active OODBMS, by extending with event-based selective attribute recording the existing prototype implementation of T_Chimera [1].

References

[1] Bertino, E., Bevilacqua, M., Ferrari, E. and Guerrini, G. Approaches to Handling Temporal Data in Object-Oriented Databases. TR 192-97, Department of Computer Science, University of Milano, 1997.

[2] Bertino, E., Ferrari, E. and Guerrini, G. A Formal Temporal Object-Oriented Data Model. In *Proc. 5th Int'l Conf. on Extending Database Technology*, pages 342–356, 1996.

[3] Bertino, E., Ferrari, E. and Guerrini, G. T_Chimera: A Temporal Object-Oriented Data Model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.

[4] Bertino, E., Ferrari, E. and Guerrini, G. Navigational Accesses in a Temporal Object Model. *IEEE Trans. on Knowledge and Data Engineering*, to appear.

[5] Bertino, E., Guerrini, G. and Rusca, L. Object Evolution in Object Databases. In *Dynamic Worlds: From the Frame Problem to Knowledge Management.* Kluwer, 1998, to appear.

[6] Bettini, C., Wang, X.S., Bertino, E. and Jajodia, S. Semantic Assumptions and Query Evaluation in Temporal Databases. In *Proc. of the ACM SIGMOD Conference*, pages 257–268, 1995.

[7] Cattel, R. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, 1996.

[8] Chakravarthy, S., Krishnaprasad, V., Anwar, E. and Kim, S.K. Snoop: An Expressive Event Specification

⟨event⟩ ::= ⟨event⟩ AND ⟨event⟩ | ⟨event⟩ OR ⟨event⟩ |
sequence(⟨event⟩, ⟨event⟩,⟨time_spec⟩, ⟨time_spec⟩) |
sequence(⟨event⟩, ⟨event⟩,⟨*⟩) | happen(⟨event⟩, ⟨freq_spec⟩, ⟨event⟩, ⟨event⟩) |
not_happen(⟨event⟩, ⟨freq_spec⟩, ⟨event⟩, ⟨event⟩) | begin_on⟨event⟩ |
end_on⟨event⟩ | begin⟨event⟩ | end⟨event⟩ | ⟨db_event⟩ | ⟨temporal_event⟩

⟨db_event⟩ ::= ⟨update_event⟩ | ⟨value_event⟩

⟨update_event⟩ ::= generalize| specialize| create| delete | migrate | ⟨update_attr_event⟩ | ⟨meth_name⟩

⟨update_attr_event⟩ ::= update(⟨path_expr⟩) | increase(⟨path_expr⟩) | decrease(⟨path_expr⟩)

⟨path_expr⟩ ::= ⟨attr_name⟩ | ⟨class_name⟩.⟨path_expr⟩

⟨value_event⟩ ::= ⟨simple_value_event⟩ ⟨comp_op⟩ ⟨simple_value_event⟩

⟨simple_value_event⟩ ::= ⟨value⟩ | ⟨path_expr⟩ | count(⟨simple_value_event⟩) |
⟨simple_value_event⟩ ⟨op⟩ ⟨simple_value_event⟩

⟨comp_op⟩ ::= $>$ | $<$ | $\geq$ | $\leq$ | $=$ | $\neq$ | $\in$ | $\notin$

⟨op⟩ ::= + | - | * | / | $\cup$ | $\cap$ | $\setminus$

⟨temporal_event⟩ ::= ⟨set_of_time_instants⟩ | ⟨time_instant⟩ | ⟨set_of_time_intervals⟩ |
(⟨time_interval⟩,⟨periodic_expr⟩) | (⟨event⟩,⟨event⟩, ⟨periodic_expr⟩) |
(⟨event⟩,⟨time_instant⟩)

⟨time_spec⟩ ::= ⟨time_instant⟩ | null

⟨freq_spec⟩ ::= ⟨nat_number⟩ | *

Figure 2. Event language syntax

Language for Active Databases. *Data & Knowledge Engineering*, 14:1–26, 1994.

[9] Dean, T. L. and McDermott, D. V. Temporal Data Base Management. *Artificial Intelligence*, 32(1):1-55, April 1987.

[10] Jensen, C.S. and Mark, L. Queries on Change in an Extended Relational Model. *IEEE Trans. on Knowledge and Data Engineering*, 4(2):192–200, April 1992.

[11] Niezette, M. and Stevenne, J. An Efficient Symbolic Representation of Periodic Time. In *1st International Conference on Information and Knowledge Management*, 1992.

[12] Ozsoyoglu, G. and Snodgrass, R.T. Temporal and Real-Time Databases: a Survey. *IEEE Trans. on Knowledge and Data Engineering*, 7(4):513-532, August 1995.

[13] Tsotras, V.J. and Kumar, A. Temporal Database Bibliography Update. *SIGMOD-RECORD*, 25, 1996.

A Event Language Syntax and Semantics

In the following we illustrate the syntax and semantics of our event language.

A.1 Syntax

The syntax in BNF form of our event language is reported in Figure 2. Non terminal symbols ⟨value⟩, ⟨attr_name⟩, and ⟨class_name⟩ represent elements of the domains $\mathcal{V}$, $\mathcal{AN}$, and $\mathcal{CI}$, respectively. ⟨meth_name⟩ denotes a method name. Finally, non terminal symbols ⟨time_instant⟩, ⟨set_of_time_instants⟩, ⟨set_of_time_intervals⟩, and ⟨nat_number⟩ represent elements of the domains $\mathbb{N} \cup \infty$, $2^{\mathbb{N} \cup \infty}$, $2^{\mathbb{N} \cup \infty \times \mathbb{N} \cup \infty}$, and $\mathbb{N}$, respectively.

A.2 Semantics

The semantics of events in $\mathcal{E}$ is given by their interpretation $\mathcal{I}()$ in first order logic and is reported in Table 2. In defining $\mathcal{I}()$ we make use of two functions *start*() and *end*(), that receive as input an event and return the starting and ending times of E, respectively. Formally, let $E \in \mathcal{E}$:

start(E) = $\{t \mid f(E,t) \wedge \neg f(E,t-1)\}$

end(E) = $\{t \mid f(E,t) \wedge \neg f(E,t+1)\}$[7]

Moreover, we make use of function *card* which computes the cardinality of a given set. P denotes a generic periodic expression. $\Pi(\mathtt{P})$ denotes the set of time instants represented by P.[8] In addition, in the semantics of value events, $db \models E$ denotes that the value event E evaluates

[7] Function f has been defined in Subsection 3.

[8] We refer to [11] for the formal definition of $\Pi()$.

Event	Interpretation $\mathcal{I}()$
E = update_event,[a] $E \neq$ increase, decrease	$\forall t(\exists t', \exists t''(\langle [t', t), db\rangle \longrightarrow^{E} \langle [t, t''), db'\rangle \in db\_history) \rightarrow f(E, t))$
E = value_event	$\forall t(\exists t', \exists t''(\langle [t', t'']\, db\rangle \in db\_history \wedge db \models E \wedge t \in [t', t'']) \rightarrow f(E, t))$
E = increase(p)/decrease(p)	$\forall t(\exists t', \exists t'', \exists E'(\langle [t', t), db\rangle \longrightarrow^{E'} \langle [t, t''), db'\rangle \in db_history) \wedge E' =$update(p) $\wedge \langle db, db'\rangle \models E) \rightarrow f(E, t))$
$E = \mathtt{t}_1$	$\forall t(t = \mathtt{t}_1 \rightarrow f(E, t))$
$E = \{\mathtt{t}_1, \ldots, \mathtt{t}_n\}$	$\forall t(t = \mathtt{t}_1 \vee \ldots \vee t = \mathtt{t}_n \rightarrow f(E, t))$
$E = \{[\mathtt{t}_1,\mathtt{t}_2], \ldots, [\mathtt{t}_n,\mathtt{t}_{n+1}]\}$	$\forall t(\mathtt{t}_1 \leq t \leq \mathtt{t}_2 \vee \ldots \vee \mathtt{t}_n \leq t \leq \mathtt{t}_{n+1} \rightarrow f(E, t))$
$E = ([\mathtt{t}_1,\mathtt{t}_2],\mathtt{P})$	$\forall t(\mathtt{t}_1 \leq t \leq \mathtt{t}_2 \wedge t \in \Pi(\mathtt{P}) \rightarrow f(E, t))$
$E = (E_1, E_2,\mathtt{P})$	$\forall t(\exists t_1, \forall t'(t_1 \in start(E_1) \wedge t_1 \leq t \wedge (t_1 \leq t' < t \rightarrow t' \notin end(E_2)) \wedge$ $\wedge(t_1 < t' \leq t \rightarrow t' \notin start(E_1)) \wedge t \in \Pi(\mathtt{P})) \rightarrow f(E, t))$
$E = (E_1, t')$	$\forall t(\exists t_1, \forall t''(t_1 \in start(E_1) \wedge t = t_1 + t' \wedge (t_1 < t'' \leq t \rightarrow t'' \notin start(E_1)) \rightarrow f(E, t))$
$E =$begin(E_1)	$\forall t(\exists t_1(t_1 \in start(E_1) \wedge t_1 = t) \rightarrow f(E, t))$
$E =$end(E_1)	$\forall t(\exists t_1(t_1 \in end(E_1) \wedge t_1 = t) \rightarrow f(E, t))$
$E =$begin_on(E_1)	$\forall t(\exists t_1(t_1 \in start(E_1) \wedge t_1 \leq t) \rightarrow f(E, t))$
$E =$end_on(E_1)	$\forall t(\exists t_1(t_1 \in end(E_1) \wedge t_1 \leq t) \rightarrow f(E, t))$
$E = E_1$ OR E_2	$\forall t(f(E_1, t) \vee f(E_2, t) \rightarrow f(E, t))$
$E = E_1$ AND E_2	$\forall t(f(E_1, t) \wedge f(E_2, t) \rightarrow f(E, t))$
$E =$sequence$(E_1, E_2,\mathtt{t}_{min},\mathtt{t}_{max})$	$\forall t(\exists t_1(\forall t'(t_1 \in end(E_1) \wedge t \in start(E_2) \wedge t = t_1 + \mathtt{t}_{max} - \mathtt{t}_{min} \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin start(E_2)) \wedge (t_1 < t' \leq t \rightarrow t' \notin end(E_1)))) \rightarrow f(t, E))$
$E =$sequence$(E_1, E_2,\mathtt{t}_{min},$null$)$	$\forall t(\exists t_1(\forall t'(t_1 \in end(E_1) \wedge t \in start(E_2) \wedge t_1 + \mathtt{t}_{min} \leq t \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin start(E_2)) \wedge (t_1 < t' \leq t \rightarrow t' \notin end(E_1)))) \rightarrow f(t, E))$
$E =$sequence$(E_1, E_2,$null$,\mathtt{t}_{max})$	$\forall t(\exists t_1(\forall t'(t_1 \in end(E_1) \wedge t \in start(E_2) \wedge t \leq t_1 + \mathtt{t}_{max} \leq t \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin start(E_2)) \wedge (t_1 < t' \leq t \rightarrow t' \notin end(E_1)))) \rightarrow f(t, E))$
$E =$sequence$(E_1, E_2,$*$)$	$\forall t(\exists t_1 \forall t'(t_1 \in end(E_1) \wedge t \in start(E_2) \wedge t_1 < t) \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin start(E_2)) \wedge (t_1 < t' \leq t \rightarrow t' \notin end(E_1)))) \rightarrow f(t, E))$
$E =$happen(E_1, n, E_2, E_3)	$\forall t(\exists t_1(\forall t'(t_1 \in start(E_2) \wedge t \in start(E_1) \wedge t_1 \leq t \wedge$ $card(\{\overline{t} \mid \overline{t} \in start(E_1) \wedge t_1 \leq \overline{t} < t\}) = n - 1 \wedge (t_1 \leq t' < t \rightarrow t' \notin end(E_3)) \wedge$ $\wedge(t_1 < t' \leq t \rightarrow t' \notin start(E_2)))) \rightarrow f(t, E))$
$E =$happen$(E_1, *, E_2, E_3)$	$\forall t(\exists t_1(\forall t'(t_1 \in start(E_2) \wedge t \in start(E_1) \wedge t_1 \leq t \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin end(E_3)) \wedge (t_1 < t' \leq t \rightarrow t' \notin start(E_2)))) \rightarrow f(t, E))$
$E =$not_happen(E_1, n, E_2, E_3)	$\forall t(\exists t_1(\forall t'(t_1 \in start(E_2) \wedge t \in end(E_3) \wedge t_1 \leq t \wedge$ $\wedge card(\{\overline{t} \mid \overline{t} \in start(E_1) \wedge t_1 \leq \overline{t} \leq t\}) < n \wedge (t_1 \leq t' < t \rightarrow t' \notin end(E_3)) \wedge$ $\wedge(t_1 < t' \leq t \rightarrow t' \notin start(E_2)))) \rightarrow f(t, E))$
$E =$not_happen$(E_1, *, E_2, E_3)$	$\forall t(\exists t_1(\forall t'(t_1 \in start(E_2) \wedge t \in end(E_3) \wedge t_1 \leq t \wedge (t_1 \leq t' \leq t \rightarrow t' \notin start(E_1)) \wedge$ $\wedge(t_1 \leq t' < t \rightarrow t' \notin end(E_3)) \wedge (t_1 < t' \leq t \rightarrow t' \notin start(E_2)))) \rightarrow f(t, E))$

[a] If E is a method invocation then $f(E, t)$ is true whenever the method is being executed, that is, $start(E)$ denotes the time of method invocation, whereas $end(E)$ denotes the time of return from method invocation.

Table 2. Semantics of events

to true on the database state db, whereas $\langle db, db'\rangle \models E$ denotes that E evaluates to true with respect to the state transition $\langle [t_1, t_2), db\rangle \longrightarrow^{E} \langle [t_2, t_3), db'\rangle$.