Computers and Artificial Intelligence, Vol. 19, 2000, Pageproofs

FUNCTION ORIENTED HISTORY REPRESENTATION IN DATABASES

Lázsló Kovács

University of Miskolc
Dept. of Information Technology
H-3515 Miskolc Egyetemváros, Hungary
e-mail: `kovacs@iit.uni-miskolc.hu`

Costas Vassilakis

University of Athens
Dept. of Informatics
Panepistimiopolis, TYPA
15771 Athens, Greece
e-mail: `costas@di.uoa.gr`

Abstract. In the past years the management of temporal data has attracted numerous researchers resulting to a large number of temporal data extensions to the relational and object oriented data models. In this paper, the proposed temporal data model focuses on the functional characteristics of the histories. The paper introduces a set oriented description of the calendars together with a function oriented history concept with a history-algebra. The completeness of the proposed model with respect to the reduced temporal algebra TA is also proven. The expressive power of the proposed model is demonstrated in the end of the paper by a hospital example.

Keywords: Temporal databases, calendars, history, history algebra.

1 INTRODUCTION

Temporal database management systems (TDBMS) extend the traditional database management systems by incorporating the temporal aspects of the data. Temporal databases store the history of the data updates including the different values and the dates when changes occurred. Several approaches were developed during the last years to support the temporal aspects in databases [14], [16]. These temporal databases allow the users to query not only the current state, but the history of the database as well, and sometimes also the investigation of anticipated future events is possible. The proposals for the temporal extensions are mainly related to the three basic data models:

1. ER Semantic Data Model [8]
2. Relational Data Model [3]
3. Object Oriented Data Model [1]

Beside the time-invariant objects, i.e. the objects whose value does not change, the different approaches propose several types of time-variant objects. Most of the proposed temporal database management systems given in [6] include temporal objects with version-based characteristics. These objects may change their values with arbitrary frequency. Every version constitutes of a date and a complex value. If the changes occur regularly, according to a particular pattern of time, we get the time-series objects [15]. This pattern of time is called a *calendar*. The administration of the time-series objects differs from the management of the version-based objects as the regularity involves new aspects, different implementation techniques and new integrity constraints. In [12], an integrated temporal data model is given that incorporates both version-based and time-series objects.

In this paper, we propose another approach for interpreting both the version-based and time-series temporal objects in an integrated way. We treat the list of historical values as function over the time axis. Based on this assumption, we can define a set of operators to manage the temporal data. Besides the operators, we deal with the integrity constraints related to the derived new data type. To compare our proposal with other well-known query interfaces, we prove the completeness with respect to the reduced temporal algebra TA for the non-grouped, flat data model. The functionality of the proposed approach is demonstrated through an application-oriented example. The example refers to a clinical research information system, where patients are examined several times, according to a number of criteria.

The paper is organized as follows. In Section 2, the different aspects of the temporal data and the existing representation alternatives are presented. Section 3 gives the structural description of our approach. The history list is defined in detail in Section 4. Section 5 presents the proposed operators and integrity constraints. A completeness proof is contained in Section 6. The hospital example is demonstrated in Section 7.

2 BASIC CONCEPTS

In this paper, unless otherwise stated, we will use the term "time" for the valid time [7]. We will use the term *object* to refer to the information element which time is related to.

The time related to an object can be described by different types of quantities. In [17], the following types of time are defined:

1. an *instant* is a time point,
2. a *period* is a quantity of time between two time instants, called boundaries,
3. an *interval* is a duration of time with known length but without specified boundaries.

The value of a time entity can be expressed in different formalisms. Usually a discrete time model is employed. The smallest segment of the time axis is called granule. The concept of *calendar* [11] is used to describe the granularity dependant characteristics of time assignment. A calendar is usually given by a set of allowed granules, and the mapping between the different granularities. The smallest granule is called chronon.

During the lifetime of an object, several events may be recorded in the database with different time values. Thus a list of time values is usually assigned to every temporal object in the database. The *history* of an object can be described by a list of the following form:

$$\{(t_1, \langle objectstate \rangle), (t_2, \langle objectstate \rangle), \dots\}$$

where t_i denotes a temporal value.

According to [12], we can distinguish two types of lists. The first type is called *version based list*, while the second one is called *time-series list*. The main difference between these types is that in the case of time-series the $\mathbf{t}_i$ time values strictly follow a time pattern, i.e. the $t_{i+1} - t_i$ value is the same for every i. In time-series lists, the time values are usually based on a calendar. On the other hand, in version based lists, the t_i values may be arbitrary.

Regarding the granularity of the temporal attributes, we distinguish:

1. *object versioning*: the history list contains object states and each time value is assigned to the whole objects [13],
2. *attribute versioning*: every attribute of an object may have its own history list, i.e. the t_i values are assigned to the attribute values [5]. Each temporal real-word object is represented by a complex, nested database object.

3 SET-ORIENTED DESCRIPTION OF CALENDARS

Let B be a finite interval of natural numbers, i.e.

$$B = [a, b], \text{ where } a, b \in N \wedge a < b.$$

Let c_i denote a chronon unit of the time axis. The

$$C_0 = \{c_i\}$$

where $i \in B$ is called the chronon-calendar. The set C_0 is totally ordered with respect to the following property: $c_i < c_j$ if and only if $i < j$. In other words, C_0 is equivalent to B regarding the '$<$' relation.

We define the *set of calendars* X as follows:

(a) $C_0 \in X$

(b) If $C_1 \in X \wedge C_2 \subset 2^{C_1}$:

   (1) $\forall e_1, e_2 \in C_2 \Rightarrow e_1 \cap e_2 = \emptyset$
   (2) $\forall e_1, e_2 \in C_2, \forall x_1, x_2 \in e_1, \forall y_1, y_2 \in e_2 : (x_1 < y_1 \Leftrightarrow x_2 < y_2) \wedge (x_1 > y_1 \Leftrightarrow x_2 > y_2)$

   then $C_2 \in X$ i.e. C_2 is a calendar. In this case, calendar C_1 is called the *base calendar* of C_2 and will be denoted as $B(C_2)$.

(The notation 2^{C_1} denotes the *powerset* of C_1, i.e. the set of subsets of C_1.)

In this definition, calendars other than the chronon calendar are defined as a set of subsets; this enables transformations between the different granularity levels with minimal information loss. When moving to finer granularities, no information loss occurs.

An ordering relation may be defined among the elements of calendar C_2 as follows: Let e_1, e_2 denote two arbitrary elements of C_2 and let x and y be arbitrary elements of e_1 and e_2, respectively. Then

(a) $e_1 < e_2$, if $x < y$,

(b) $e_1 > e_2$, if $x > y$.

Since $E = \{x \mid x \in e_i \wedge e_i \in C_2\}$ is a subset of C_1, E is totally ordered. Thus calendar C_2 is totally ordered as well.

Based on this result, we can define an ordered list of the values $e \in C_2$ and we can assign the position number to an e element as an index. Thus if $e = \{x_i, x_j, \ldots, x_n\} \in C_2$, the index of element e is

$$\min(y) : x_y \in e.$$

By virtue of this definition, the index set of any calendar C is drawn from B, and $e_i < e_j$ if and only if $i < j$.

We call two calendars, C_1 and C_2 non-overlapping, if taking the corresponding representation at the chronon granularity level, the two sets are disjoint.

Since every calendar except the C_0 calendar has a base calendar, the calendars can be structured into an hierarchy. The parent calendar of a calendar C is $B(C)$. The root element of this hierarchy is C_0. If C_1 and C_2 are two arbitrary calendars and there is a path from C_1 to C_2 towards the leafs, then C_1 is an *ancestor* of C_2.

Given two calendars C_1 and C_2, C_1 is *convertible* into C_2 if C_2 is an ancestor of calendar C_1. We define the conversion function $M(C_1, C_2)$ as follows:

$$M(C_1, C_2) = \begin{cases} \{e \mid \exists x \in C_1 : e \in x\}, \text{ if } C_2 = B(C_1) \\ M(M(C_1, B(C_1)), C_2), \text{otherwise.} \end{cases}$$

In other words, converting a calendar C_1 to an ancestor calendar C_2 is equivalent to selecting the elements of the ancestor calendar which are included (at any depth) in C_2. We call the result of the $M(C_1, C_2)$ function *the representation of* C_1 *at the* C_2 *level.*

We introduce the term of filtering for the calendars. This operation can be used to extract a subset of the calendar. If C_1 and C_2 are two arbitrary calendars and C_1 is an ancestor of C_2, then

$$\mathit{Filter}(C_1, C_2) = \{e \mid e \in C_1 \wedge \forall x \in e : x \in M(C_2, C_1)\}$$

where $M(C_2, C_1)$ denotes the representation of calendar C_2 at the $B(C_1)$ level. The result subset contains only those elements from C_1 which are members of the calendar C_2.

For every calendar C except C_0, we can define a membership function over its base calendar. This function has values zero or one, defined as follows:

$$f(x) = \begin{cases} 1, \text{ if } \exists\, e \in C : x \in e \\ 0, \text{ otherwise.} \end{cases}$$

We can see that this type of formalism is equivalent to the original calendar formalism, but it provides additionally a more homogenous and flexible tool to manage the temporal data. We can use the following algorithm to convert a normal calendar specification into a set-oriented formalism:

> Let $C\{gr, pt, pr, st, et\}$ be a calendar according to [12], i.e. *gr* is a granularity specification, *pt* a pattern set, *pr* a period specification, *st* a starting time value and *et* an ending time value. As the *gr* defines the units of C, i.e. the elements of the calendar, it is straightforward to relate the granularity to the role of the base calendar. If the *pr* is not null, then the membership function of the C calendar is periodical. If *pr* is null then the membership function has no period. The *pt* set together with the *pr* value determine the concrete form of the membership function. The *pt* is the description of the membership function within a period. This membership function fragment should be repeated in every period. The *st* and *et* values give the absolute boundaries of the membership function.

The problems of the normal calendar definition and interpretation can be illustrated by the following example from [12]. It defines the calendar *Months*:

Calendar Months $\langle$
granularity: Day,
pattern: $\{[1, 28] \mid [1, 29] \mid [1, 30] \mid [1, 31]\}$
period: $28 \mid 29 \mid 30 \mid 31$
start time: 1
end time: $\infty\rangle$

The confusing side of this definition is that it does not express unambiguously the relationship between periods and patterns, neither it provides information when the different periods or patterns should be used. This shows that we cannot oversimplify the formalism without the losing of some crucial information parts.

4 STRUCTURAL DESCRIPTION OF HISTORY LISTS

According to [12], the history of an object can be described by a list of the following form:

$$\{(t_1, \langle objectstate\rangle), (t_2\langle objectstate\rangle), \ldots, (t_m, \langle objectstate\rangle)\}$$

where t_i denotes a temporal value. We assume the following:

(a) the t_i values are disjoint (i.e. $\forall i, j \leq m : t_i \cap t_j = \emptyset$),

(b) $\forall i : t_i < t_{i+1}$ is met,

(c) every t_i has the same granularity.

These conditions are usually met in real applications. $\langle objectstate\rangle$ is an arbitrary structure.

We can assign a calendar C to every history list. We will call this calendar a *history calendar*. The history calendar C has the following properties:

(a) every element e_i of C corresponds to the t_i from the history,

(b) $x \in e_i \Leftrightarrow x \in t_i$,

(c) $|C| = m$.

Based upon this formalism, we will consider the

$$\{(t_1, \langle objectstate\rangle), (t_2, \langle objectstate\rangle), \ldots, (t_m, \langle objectstate\rangle)\}$$

history list as a function

$$H : C \to \{\langle objectstate\rangle\}$$

from the history calendar into the set of subsets of possible object state values. This function returns an object state for every element of C, i.e. for every time position stored in C. We call this function as a *history function.* $Dom(H)$ denotes the domain of H, i.e. the calendar C. We use the symbol $ValDom(H)$ to denote the database domain of the corresponding object states.

Since an hierarchical relationship exists among the calendars, function H may be transferred along this hierarchy. Mapping the H function into the base calendar, the result function is denoted by

$$M(H) : B(C) \rightarrow \{\langle objectstate \rangle\}.$$

Function $M(H)$ is defined as follows:

$$M(H)(x) = \begin{cases} H(e), \text{ if } x \in B(C), e \in C, x \in e \\ \text{undefined, otherwise.} \end{cases}$$

Applying this transformation recursively, function H may be mapped to the C_0 level. Thus every history list has a descriptor function at the chronon granularity.

5 OPERATORS

The functionality of a data model is mainly expressed by the corresponding set of operators. All proposals for temporal data models contain an operational part too. In [12], for example, the following operators are introduced as the basic operators of the model:

(a) calendar operators: select, intersect, union, exclude,

(b) history operators: insert_entity, delete_entity, modify_attribute.

The formalisms used in the proposals are diverse, although their functionality is quite similar. As the data manipulation part contains generally very clear and unambiguous activities, we are focusing here only on the query part. First we take the requirements and the operational structure into consideration. Regarding the required functionality, we use the requirements given in [9] as guiding principles.

It is expected that the temporal model will be implemented by extending an existing data model (like relational or object oriented). The extension may be implemented in a *layered* fashion [19], according to which the existing data model becomes the bottom layer, supporting the non-temporal characteristics, whereas temporal features are implemented in a *temporal layer*, operating on top of the existing data model. We assume that the existing model contains the following structural elements:

(a) entity set level (a relation in terms of the relational model),

(b) attribute level (an attribute in terms of the relational model).

We will use the term *host layer* to refer to the existing layer. We assume that the host layer contains a powerful operational part that meets the requirements regarding the attributes and entity set elements. The temporal layer will connect to this host layer. In our proposal, the temporal elements, the history lists are contained in the attributes.

Before going on, we should mention that our assumption (i.e. keeping history lists in attributes rather than tuples) seems to be a restriction regarding the application of tuple versioning. In our model only the attributes may have temporal descriptor, indeed. But it can be shown that every tuple-versioned formalism can be converted into attribute-versioned description without any loss of information provided that it exists an unambiguous grouping expression [19].

We have to involve new operators manipulating the history lists. The coupling of the two query systems should be solved without hurting the rules of the host system. We denote the host query system by $Q1$, and the set of new temporal part as $Q2$.

Analysing the requirements given in [9] we can identify the necessity of the following new operator classes:

(a) $Q2$ internal operators:
Operators of this type are used to process the history list instances, i.e. the temporal attribute values. The results of these operators are history lists. We call the set of these operators *a history list algebra*. An example is the selection of a sub-history based on the calendar or on the state value.

(b) $Q2 - Q1$ operators:
These operators accept one or more history lists (temporal attributes) as parameters and return a value defined in the host system. We can divide these operators into the following groups:

   (1) *Intra-history operators*: the calculation uses only the data values stored within a single history list. The maximal duration of the history intervals can be calculated by an operator of this group.

   (2) *Inter-history operators*: the calculation uses the data values stored within several history lists. Within this group we define two sub-groups:

      i. *intra-instance*: the different history lists belong to the same object entity,

      ii. *inter-instance*: the different history lists belong to the different object entities.

The elements of $Q2$ will be appear as an extension to the $Q1$ query system. The extension covers the introduction of

(a) new data types and literal types,

(b) new operators and functions.

5.1 $Q2$ Internal Operators, History Algebra

In order to provide a flexible query and manipulation language, we introduce an algebra for the history functions. Let H_1, H_2 denote two arbitrary history functions. We define the following basic operators for the history functions:

(a) $\mathit{Select}(H_1, \mathit{cond}|C)$ or $\sigma_t^H(H_1, \mathit{cond}|C)$, to select a subpart of H_1.

(b) $\mathit{Project}(H_1, \mathit{ident}-\mathit{list})$ or $\pi^H(H_1, \mathit{ident}-\mathit{list})$, the projection on a list of object elements.

(c) $\mathit{Join}(H_1, H_2)$ or $\times^H(H_1, H_2)$, which joins those object states of H_1 and H_2 which pertain to the same time element, into a single tuple.

(d) $\mathit{Union}(H_1, H_2)$, to merge the two H functions.

(e) $\mathit{Extend}(H_1, C)$, to extend the H_1 function's domain with C.

(f) $\mathit{Expand}(H_1)$, to set the H_1 function's domain to its basic calendar.

(g) $\mathit{Group}(H_1, C, \mathit{aggr})$, to set the H_1 function's domain to a higher level calendar C using the given aggregation function *aggr*.

Using these basic operators, we can define a set of derived operators, which can be used for complex tasks and which can be expressed in terms of the basic operators. We will mention here only two such complex operators:

(a) $\mathit{Normalize}(H_1)$, to set the H_1 function's domain to C_0.

(b) $\mathit{Reformat}(H_1, C, \mathit{aggr})$, to set the H_1 function's domain to C.

All of these operators result in a new history function, i.e. the set of the history functions is closed related to the defined operators.

The $\mathit{Select}(H_1, \mathit{cond}|C)$ operator can be used to filter the H_1 function. The second parameter contains the filtering condition. The filter may refer both to the calendar time value and to the object state. The *cond* part contains a filter for the object state which is evaluated for every object state instant. C denotes a calendar that is used as a filtering condition. If C is used, only those domain elements which are contained in C remain in the domain. Calendar C should be convertible into $\mathit{Dom}(H_1)$. The domain of the result history function is a subset of the domain belonging to the H_1 function.

The $\mathit{Project}(H_1, \textit{indent-list})$ operator can be used to reduce the object state structure elements to a subset of this set. The second parameter is a list of object state structure component identifiers. The result history list has the same domain as H_1 has, but the value domain will be reduced.

The $\mathit{Join}(H_1, H_2)$ operator results in couples of the elements of H_1 and H_2. The precondition for the join operation is that both H_1 and H_2 are based on the same base calendar. The result calendar consists of the

$\{(h_1(t), h_2(t)), t\}$

history elements for every t in the common domain.

The $Union(H_1, H_2)$ operator is used to merge the H_1 and H_2 functions. To successfully perform the merging, the following conditions should be met:

(a) $ValDom(H_1) = ValDom(H_2)$,

(b) $Dom(H_1)$ and $Dom(H_2)$ are disjoint calendars.

The domain of the result history function is based on the greatest common divisor calendar. The elements come either from H_1 or H_2. The term of *greatest common divisor calendar* corresponds to the nearest common ancestor node in the base-calendar tree. The calendar C_1 is the parent of the calendar C_2 in the base calendar tree, if $B(C_2)$ is equal to C_1.

The $Extend(H_1, C)$ operator is used to extend the H_1 function's domain with the C calendar. To successfully perform the extension, the following conditions should be met:

(a) $Dom(H_1)$ and C are disjoint calendars,

(b) H_1 should be an extendable history.

Not every history is extendable as it has no sense to assign a state for such point of time where the object has no state. This may occur if the object stores the history of events. The events are related to fixed time-points. The domain of the resulting history function H is based on the greatest common divisor calendar. Let C_r denote the domain of the result function. For every $e \in C_r$ the corresponding $H(e)$ is calculated as follows:

(a) $H(e) = H_1(e')$, if e was derived from $e' \in Dom(H_1)$,

(b) otherwise the value $H(e)$ is interpolated (or extrapolated) from the H_1 history function.

We distinguish different types of interpolation algorithm, depending on the characteristic of the $ValDom(H_1)$ database domain.

The $Expand(H_1)$ operator is used to create a new history function which is derived from H_1 and it is based on the basic calendar of H_1. The result of the $Expand(H_1)$ operation is equivalent to the $M(H_1)$ function.

The $Group(H_1, C, aggr)$ operation is in some way the opposite of the *Expand* operator. It is used to set the H_1 function's domain to a higher level using the given *aggr* aggregation function. To successfully perform the aggregation, the following condition should be met:

$Dom(H_1)$ is an ancestor of the C calendar.

When performing aggregation, several $e \in \mathit{Dom}(H_1)$ are mapped to the same $a \in C$. The resulting history set H is based on C, i.e. $\mathit{Dom}(H)$ is equal to C. The $H(e)$ values for every $a \in C$ are calculated by means of the *aggr* function.

By means of these basic operators, we can introduce new, complex operators too. The $\mathit{Normalise}(H_1)$ is used to set the H_1 function's domain to C_0. This can be achieved by applying the $\mathit{Expand}(H_1)$ operation iteratively, until the domain of the result function is equal to C_0.

The $\mathit{Reformat}(H_1, C, \mathit{aggr})$ operator can be used to set the H_1 function's domain to a new calendar C. The *Reformat* operation, by means of a series of *Extend* and *Group* operators transforms the calendar into an equivalent calendar with another base domain.

5.2 $Q2 - Q1$ Intra-History Operators

In order to perform queries and to define constraints, functions should be incorporated into the proposed model. Only some basic functions are mentioned here, however more functions are necessary to enhance expressiveness. In general, the usual operators, descriptor elements for the mathematical functions as well the usual relations can be adapted here. The main purpose of this section is only to demonstrate the embedding of the functions into the model.

We first introduce logical functions, which yield Boolean values. In the description, symbols H, H_1, ... denote history functions, while symbols C, C_1, ... denote calendars.

(a) $\mathit{CMatch}(H_1, C_2)$: it returns *true* if and only if for every element e_2 of C_2 there exists an element e_1 in $C_1 = \mathit{Dom}(H_1)$ such that e_1 is a subset of e_2. The *CMatch* logical function can be used to check whether C_1 contains a history element for every time interval given by C_2.

(b) $\mathit{MonInc}(H, \mathit{expr})$: it returns *true*, if for every e_1, $e_2 \in \mathit{Dom}(H)$ the following condition is met:

$$H(e_1).\mathit{expr} < H(e_2).\mathit{expr} \Leftrightarrow e_1 < e_2$$

where *expr* is an expression based on the attributes of H. The $H(e).\mathit{expr}$ denotes the value of the expression at the e time element.

(c) $\mathit{CrosValUp}(H, \mathit{expr}, \mathit{val})$: it returns *true* if and only if there exists an $e \in \mathit{Dom}(H)$ such that

$$H(e).\mathit{expr} > \mathit{val}.$$

This function can be used to check whether the object values crossed a threshold value or not.

We can additionally define functions to access the different components of the history list. These functions can return a list of values or a single value, returning a component, a subpart of a history.

(a) *GetList*(H): it returns the whole history list in a readable form. The concrete form depends on the host database environment.

(b) *GetVal*$(H, ident)$: it returns the values of the *ident* attribute from the history list.

(c) *GetDom*(H): it returns the domain calendar of the H history list.

(d) *GetAggVal*$(H, ident, func)$: it returns only one aggregated value for the set of *ident* attribute values from the history list.

(e) *GetCount*(H): it returns the number of elements in H.

(f) *GetMinTime*(H): it returns the time position of the first element in H.

(g) *GetMaxTime*(H): it returns the time position of the last element in H.

5.3 $Q2 - Q1$ Intra-History Operators

Inter-history operators facilitate comparisons between different histories and calculations on sets of histories, which includes aggregate functions.

(a) *VDomin*$(H_1, iden_1, H_2, iden_2)$: it returns *true* if H_1 is dominated by H_2, i.e. the $H_2(e).iden_1$ value is greater than the $H_1(e).iden_2$ value for every time position. We assume that the conditions

(1) $Dom(H_1) = Dom(H_2)$
(2) $ValDom(H_1) = ValDom(H_2)$

are met.

(b) *HSum*$(A, expr, [group])$: this aggregation function computes the sum of *expr* evaluated on the A attribute of history type for every tuple instance in the object instance set. *expr* is a list of expressions. The result is a composite value, similar to an object state. Elements of *expr* may refer to the time component too. If the group tag is given, then the list elements are grouped according to the group tag. In this case, the *expr* may contain the group expression element too. The expression is evaluated for every group. If group tag is not present, only one list element is contained in the result value structure.

(c) *HMin*$(A, expr, [group])$: this aggregation function computes the minimum of the *expr* evaluated on the A attribute of history type for every tuple instance in the object instance set for every group.

(d) *HMax*$(A, expr, [group])$: this aggregation function computes the maximum of *expr* evaluated on the A attribute of history type for every tuple instance in the object instance set for every group.

(e) $HCount(A, expr, [group])$: this aggregation function computes the count of the tuples for which the simple expression *expr* evaluates to a non-null value. Again, *expr* is evaluated on the A attribute of history type for every tuple instance in the object instance set for every group.

(f) $HAvg(A, expr, [group])$: this aggregation function computes the average of the *expr* evaluated on the A attribute of history type for every tuple instance in the object instance set for every group.

5.4 Integrity Constraints

The introduced operators and functions can be used to define integrity constraints over the history lists. We include two types of integrity constraints:

(a) *static*, which are bounded to database history list data elements and behave similarly to the *CHECK* integrity constraints in the relational model,

(b) *dynamic*, which are bounded to database history list data elements and behave similarly to the *ALERT* active integrity constraints.

The static integrity constraint is given by an expression yielding a logical value. If X is a database object of history list type, then we can assign a logical expression to X. The expression should be evaluated on the instances of X. It contains references to X which are replaced during evaluation by the actual instances of X. Only the instances for which the expression evaluates to *true* are valid. This condition is checked when

(a) X is modified or

(b) new instances of X are inserted in the database.

For example, if temperature values are not allowed to exceed 1000, the constraint expression may be the following:

NOT $CrosValUp(X$, temperature, 1000).

The dynamic integrity constraints may also be given by means of *logical expressions*. But, in this case, the expression value is continuously monitored and when the value becomes true, then an action procedure is activated. As the condition may refer to the current clock time, the need to determine the time points on which it must be checked arises. This can be handled by specifying checking intervals.

For example, we require that every day should be added a new element into the history. We create an alert that calls a 'message' stored procedure if there exists a day that has no corresponding value in the history. We assume the domain of the X is based on an Hour calendar. The corresponding expression to be checked can be given as follows:

NOT $\mathit{CMatch(GetDom}(X), (D)\{\{\mathit{GetMinTime}(X), \mathit{NOW}\}\})$;

the function to be triggered is:

$\mathit{message}()$

and the controlling period is

(D).

6 ON COMPLETENESS OF THE OPERATIONAL PART

To describe the expressive power of the different query languages we can use the metric of completeness [4]. We state that the proposed operational part is complete with respect to the reduced temporal algebra TA over the temporally ungrouped relations defined in [4]. The schema of a temporally ungrouped relation is a 3-tuple:

$$R_u = \langle A, K, D\rangle$$

where $A \cup \langle \mathit{TIME}\rangle$ is the set of attributes in R_u, K is the key,

$$D : A \cup \langle \mathit{TIME}\rangle \rightarrow \mathit{Dom}(A) \cup \mathit{Dom}(\mathit{TIME})$$

is the value domain assignment. The tuple in R_u contains beside the normal attributes $a \in A$, a t time attribute also, which expresses the valid time of the actual tuple. In R_u every attribute has scalar value. This type of relations corresponds to the tuple-versioned temporally relations. We assume the following conditions:

(a) the value of K is non-temporal,

(b) there is an integrity constraint to denote that $\forall a \in A$: a is non-temporal.

The schema of the temporally grouped relation R_g differs from R_u in the definition of D. The schema of a temporally grouped relation is a 3-tuple:

$$R_g = \langle A, K, D\rangle$$

where A is the set of attributes in R_g, K is the key and

$$D : A \rightarrow \mathit{Dom}(A) \cup \mathit{Dom}(\mathit{TIME})$$

is the value domain assignment. The attributes in R_g may contain a history.

The term of completeness can be summarised as follows [4]: given a data model M and two query languages L_1 and L_2 for M, language L_2 is complete with respect to L_1 if and only if

$$\forall db \in M, \forall q \in L_1, \exists q' \in L_2 : q'(db) = q(db)$$

where db is a database in M.

To compare two languages based on different data models, we should first find a correspondence between the database objects of the two different models. The mapping is defined between the structural components of the data models. We denote this correspondence mapping function by

$$\Omega : DS_1 \rightarrow DS_2$$

where DS_i is the corresponding structural component of the data models. Given data models $M_1 = (DS_1, QL_1)$ and $M_2 = (DS_2, QL_2)$ we say that M_2 is *weakly complete* with respect to M_1 if and only if there exist two mappings

$\Omega : DS_1 \rightarrow DS_2$ (which is a correspondence mapping)

and

$$\Gamma : QL_1 \rightarrow QL_2$$

such that for all $\phi \in QL_1$ and for all instances D of structures DS_1:

$$\Omega(\phi(D)) = \Gamma(\phi)(\Omega(D)).$$

If the Ω correspondence mapping is a one-to-one function then we say that M_2 is *strongly complete* with respect to M_1. The two data models to be compared are the proposed temporally grouped model and the temporally ungrouped model defined in [4]. Please note that, as stated in [4], *a temporally grouped model and a temporally ungrouped model supplemented with group-ids are strongly equivalent.* The temporally ungrouped model and algebra to which we compare our approach, satisfies the requirement for the existence of group-ids (we assume the existence of a non-temporal key), so it is strongly equivalent to a temporally grouped model and algebra. Thus, showing that our model is strongly complete with respect to the specific temporally ungrouped model effectively proves strong completeness with respect to temporally grouped models, which are, in general, semantically and operationally richer than temporally ungrouped ones.

We assume that the host model is based on the relational model with the relational algebra. Let RTG denote the set of grouped relations relevant to the problem area. We will define Ω_2 as a correspondence mapping function that converts any $r \in RTG$ into a temporally ungrouped relation. The set of ungrouped relations RTU corresponding to RTG is defined as follows:

$$RTU = \{\Omega_2(r) \mid r \in RTG\}.$$

We assume that every history list in RTG is based on the chronon calendar, C_0. This is not a restriction as every history can be converted into this base calendar, without any information loss. Another assumption is that we require the existence of a non-temporal key for every $r \in RTG$.

Function $\Omega_2 : RTG \rightarrow RTU$ is defined as follows:

(1) $\Omega_2(r) = \cup\Omega_2(q)$, where $q \in r$, i.e. q is a tuple in r.

(2) let $q = \{k, a_1, \ldots, a_n, a_{n+1}, \ldots, a_m\}$ be a tuple in r, where k is the key and $a_1, \ldots, a_m$ the non-key attributes. Without loss of generality, we assume that attributes $a_1, \ldots, a_n$ are non-temporal, whereas attributes $a_{n+1}, \ldots, a_m$ are temporal. Let *Lifespan*(q) denote the set of time positions in which at least one of the temporal attributes has a not null value, and $a_i(t)$ denote the value of the temporal attribute a_i (n + 1 ≤ i ≤ m) at time position t. If attribute a_i is not defined for time position t, $a_i(t) = NULL$. Then,

$$\Omega_2(q) = \{(k, a_1, \ldots, a_n, a_{n+1}(t), \ldots, a_m(t), t) \mid t \in \mathit{Lifespan}(q)\}.$$

The Ω_2 has an inverse mapping Ω_1 that is defined as follows:

(1) Group the tuples by the key k and the non-temporal attributes.

(2) Create a history for every temporal attribute $a \in REL(r)$ from the tuples contained in the group.

(3) Create a new tuple in r for every group.

The $\Omega_1 : RTU \to RTG$ function is a correspondence mapping between RTU and RTG. We can see that Ω_1 function describes a one-to-one mapping between the two sets.

Let AU and AG denote the two algebras to be compared where AU is based on the RTU and AG is based on the RTG, and let $M_u = (RTU, AU)$, $M_g = (RTG, AG)$. We prove the completeness by showing that for every operation $\phi \in AU$, there exists a counterpart operation $\Gamma(\phi) \in AG$ such that for every $D \in RTU$ the result in M_u is equivalent with the result in M_g, i.e.:

$$\Omega_1(\phi(D)) = \Gamma(\phi)(\Omega_1(D)).$$

From the set of operators in AU defined in [4], we omit the recursive operator, thus the considered AU contains the following operators:

(1) *Selection*: $S = \sigma_F(R)$ iff $S_t = \sigma_F(R_t), \forall t$ and F is a first order non-temporal formula.

(2) *Projection*: $S = \pi_{A1,\ldots,An}(R)$ iff $S_t = \pi_{A1,\ldots,An}(R_t), \forall t$.

(3) *Cartesian product*: $S = R \times Q$ iff $S_t = R_t \times Q_t, \forall t$.

(4) *Set difference*: $S = R - Q$ iff $S_t = R_t - Q_t, \forall t$ and R and Q are union-compatible.

(5) *Union*: $S = R \cup Q$ iff $S_t = R_t \cup Q_t, \forall t$ and R and Q are union-compatible.

The following conditions are met in AU:

(1) $\sigma(\cup t_i) = \cup\sigma(t_i), t_i \in R_u$,

(2) $\pi(\cup t_i) = \cup\pi(t_i), t_i \in R_u$.

6.1 Selection

We first consider the *Select* operation. Let r_u denote an arbitrary relation in RTU. r_g denotes the corresponding structure in RTG, i.e.

$$r_g = \Omega_1(r_u)$$
$$r_u = \Omega_2(r_g)$$

It follows from the definition of Ω that

$$r_u = \cup_q \Omega_2(q)$$

where q is a tuple in r_g. Let the considered selection be

$$s_u = \sigma_F(r_u).$$

It can be expressed in the following way:

$$s_u = \sigma_F(r_u) = \sigma_F(\cup_q \Omega_2(q)) = \cup_q \sigma_F(\Omega_2(q)).$$

Thus, if we can find a $\Gamma(\phi)$ for which

$$\sigma_F(\Omega_2(q)) = \Omega_2(\Gamma(\sigma_F)(q)) \tag{c1}$$
$$\Gamma(\sigma_F)(\cup q) = \cup\Gamma(\sigma_F)(q) \tag{c2}$$

are met for any arbitrary q then the $\Gamma(\sigma_F)$ is the required counterpart of the selection. The proof can be given in the following way:

$$\begin{aligned}\Omega_2(\Gamma(\sigma_F)(r_g)) &= \Omega_2(\Gamma(\sigma_F)(\cup q)) = \Omega_2(\cup\Gamma(\sigma_F)(q)) = \cup\Omega_2(\Gamma(\sigma_F)(q)) \\ &= \cup\sigma_F(\Omega_2(q)) = \sigma_F(\cup(\Omega_2(q))) = \sigma_F(\Omega_2(\cup q)) = \sigma_F(\Omega_2(r_g)).\end{aligned}$$

We can assume that the F formula does not contain $\vee$ and $\forall$ operators as they can be replaced by other operators $\neg, \wedge, \exists$. We will show that the counterpart operation of $\sigma_F(r_u)$ can be given by

$$\Gamma(\sigma_F)(r_g) = \pi'_{P'}(\sigma'_{F'}(r_g))$$

where $\pi_{P'}$ and $\sigma'_{F'}$ denote the projection and selection operators for the grouped relations. In this case (c2) is satisfied as for both π' and $\sigma'_{F'}$, the condition $\cup\theta(q) = \theta(\cup q)$ is met. If F is atomic then F may be one of the following cases:

(1) $F = [\alpha]\theta\beta$, where α, β are non-temporal attributes or literal values in RTU, θ is a binary or unary operator. In this case

   (a) $F' = Dom(\sigma^H(a_p, [\alpha]\ \theta\ \beta)) \neq \emptyset$ where a_p is a temporal attribute.

   (b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(a_p, [\alpha]\ \theta\ \beta)))$ if a_i temporal$]$.

(2) $F = [\alpha]\theta\beta$, where α denotes non-temporal attributes or literal values, β denotes a temporal attribute in r_u and θ is a binary or unary operator. In this case

   (a) $F' = Dom(\sigma^H(a_p, [\alpha]\ \theta\ \beta)) \neq \emptyset$
   (b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(a_p, [\alpha]\ \theta\ \beta)))$ if a_i temporal$]$

   where a_p is the temporal attribute of r_g whose sub-attribute is the β attribute of r_u.

(3) $F = [\alpha]\theta\beta$, where α, β are temporal attributes in r_u and θ is a binary operator. In this case

   (a) $F' = Dom(\sigma^H(\times^H(a_p, a_q), \alpha\ \theta\ \beta)) \neq \emptyset$
   (b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(\times^H(a_p, a_q), \alpha\ \theta\ \beta)))$ if a_i temporal$]$

   where a_p is a temporal attribute of r_g whose sub-attribute is the β attribute of r_u and a_q is the temporal attribute of r_g whose sub-attribute is the α attribute of r_u.

(4) $F = \alpha\theta t$, where α is a temporal literal value, t denotes the time position and θ is a binary operator. In this case

   (a) $F' = Dom(\sigma^H(a_p, \alpha\ \theta\ t)) \neq \emptyset$
   (b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(a_p, \alpha\ \theta\ t)))$ if a_i temporal$]$

   where a_p is a temporal attribute of r_g.

In the case of the non-atomic formulas, we should consider the $\neg$, $\wedge$ and $\exists$ operators.

(1) $F = \neg f$, where f is a well-formed first-order formula. We assume that the f formula is described by

   (a) $F' = Dom(\sigma^H(h, I)) \neq \emptyset$
   (b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h, I)))$ if a_i temporal$]$

   where h is the corresponding history list expression and I the corresponding condition. As we mentioned before, the $Dom(h)$ is equal to the $Lifespan(q)$ for every history-based attribute in tuple q. Assuming a two-valued logic, the true-value domain for $\neg f$ is the complement of $Dom(\sigma^H(h, I))$. As $\forall e \in h$: $I(e) \wedge \neg I(e) = 0$ and $I(e) \vee I(e) = 1$, so

$$Dom(\sigma^H(h, \neg I))$$

   yields the required complement. Thus, the parameters of the counterpart operation for $\sigma_{\neg f}$ are:

(a) $F' = Dom(\sigma^H(h, \neg I)) \neq \emptyset$
(b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h, \neg I)))$ if a_i temporal].

Similar considerations are taken for the conjunction too.

(2) $F = f_1 \wedge f_2$ and the corresponding parameters for the tag formulas are:

(a) $F' = Dom(\sigma^H(h_1, I_1)) \neq \emptyset$
(b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h_1, I_1)))$ if a_i temporal]

and

(a) $F' = Dom(\sigma^H(h_2, I_2)) \neq \emptyset$
(b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h_2, I_2)))$ if a_i temporal]

then the parameters for $f_1 \wedge f_2$ are:

(a) $F' = Dom(\sigma^H(h, I_1 \wedge I_2)) \neq \emptyset$
(b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h, I_1 \wedge I_2)))$ if a_i temporal]

with $h = \times^H(h_1, h_2)$.

(3) The last operator is the $\exists$. The usage of this operator is somewhat different in the algebra and in the calculus. The usage in the calculus is

$$\exists x(f(x)).$$

In a safe algebra we consider only those cases when x is based on a finite set. We assume that this finite set is a result of a query. Thus,

$$f(x) = f'(x) \wedge R(x)$$

where R denotes the result relation. This formula is usually implemented by the

$$\exists(\sigma_{f'}(R))$$

expression. As the existence operator is present in *AG* too, it can be proved that the counterpart expression of a $\exists(\sigma_{f'}(R))$ formula has the following parameter:

(a) $F' = Dom(\sigma^H(h, \exists(\Gamma(\sigma_{f'}(R))))) \neq \emptyset$
(b) $P' = k, \ldots, [a_i$ if a_i non-temporal $\mid \sigma^H(a_i, Dom(\sigma^H(h, \exists(\Gamma(\sigma_{f'}(R))))))$ if a_i temporal]

where h is history list over the *Lifespan* of the tuple and containing all the attributes referenced by $\Gamma(\sigma_{f'}(R))$.

As every selection with atomic formula has an equivalent expression in *AG* and we showed that after every step in building a non-atomic formula the result has a counterpart in *AG*, every selection *AU* has an equivalent operation in *AG*.

6.2 Project

The operation

$$\pi_{a1,\dots,an}(r_u)$$

eliminates some attributes from r_u where each a_i may be temporal or non-temporal attribute. As the Ω_2 mapping transfers every non-temporal attribute and every temporal sub-attribute into attributes of r_u, we can perform the projection in AG similarly to the projection in AU. Thus the counterpart of $\pi_{a1,\dots,an}(r_u)$ is:

$$\pi'_{P'}(r_g)$$

where

$$P' = k, \dots, [a_i \text{ if } a_i \text{ non-temporal } | \pi^H(a_p, a_i) \text{ if } a_i \text{ temporal}]$$

where a_p is the attribute for the a_i sub-attribute in r_g.

6.3 Cartesian Product

The Cartesian product of two RTU relations R, Q is

$$S = R \times Q = \cup_t R(t) \times Q(t) = \cup_t \cup_{i,j} (r_i(t), q_j(t), t)$$

where $R(t)$, $Q(t)$ denote the subsets belonging to the time position t and r, q are tuples from R and Q. We can show that the counterpart of this operation in AG is the normal Cartesian product, i.e.

$$\Omega_1(S) = \Omega_1(R) \times' \Omega_1(Q).$$

The product relation contains all possible pairs of tuples:

$$\Omega_2(\Omega_1(R) \times' \Omega_1(Q)) = \Omega_2(\cup_{i,j}(r_{gi}, q_{gj})) = \cup_t(\cup_{i,j}(r_i(t), q_j(t), t)) = S.$$

This shows the equivalence of the two expressions.

6.4 Union and Difference

As $\Omega(R) = \cup\Omega(q)$, we can see that

$$\Omega(R \cup Q) = \Omega(R) \cup \Omega(Q),$$
$$\Omega(R - Q) = \Omega(R) - \Omega(Q).$$

The set-based operators have the same meaning in both algebras:

$$\Gamma(\cup) = \cup',$$
$$\Gamma(-) = -'.$$

We can see that every query expression in AU has an equivalent expression in AG. Thus, the AG is complete with respect to AU.

We will demonstrate the operator equivalence rules through an example. The TG table is as follows:

K	A	B
1	(<2>, 1) (<->, 2) (<3>, 3)	(<3>, 1) (<4>, 2) (<6>, 3)
2	(<->, 1) (<3>, 2) (<->, 3) (<2>, 4)	(<5>, 1) (<4>, 2) (<3>, 3) (<4>, 4)

The corresponding $TU = \Omega_2(TG)$ is as follows:

k	a	b	t
1	2	3	1
1	-	4	2
1	3	6	3
2	-	5	1
2	3	4	2
2	-	3	3
2	2	4	4

We perform the $\sigma_{a \text{ IS NOT NULL } \wedge b>3}(TU)$ operation. The result is:

k	a	b	t
1	3	6	3
2	3	4	2
2	2	4	4

The corresponding operation on TG is formulated in the history algebra as follows:

$$\pi^H_P(\sigma^H_S(TG))$$

where

$$S = Dom(\times^H(A,B), a \text{ IS NOT NULL } \wedge b > 3) \neq \emptyset)$$
$$P = k, \sigma^H(A, Dom(\times^H(A,B), a \text{ IS NOT NULL } \wedge b > 3)),$$
$$\sigma^H(B, Dom(\times^H(A,B), a \text{ IS NOT NULL } \wedge b > 3))$$

The result of this query is

K	A	B
1	(<3>, 3)	(<6>, 3)
2	(<3>, 2) (<2>, 4)	(<4>, 2) (<4>, 4)

The mapping of this TG' table is the same as the required table, so the operation expressions are equivalent for the tables given in the example.

k	a	b	t
1	3	6	3
2	3	4	2
2	2	4	4

7 THE HOSPITAL EXAMPLE

To demonstrate the functionality of the proposed model we will use it for modelling the temporal requirements of a hospital. During the hospitalisation period, each patient receives a number of tests implying some temporal requirements. For this example, we identify the following temporal needs:

(1) A *constraint* is defined on the patients' temperature stating that at least one value per day must be recorded for all days within the hospitalisation period. An additional constraint is that the temperature should always be between two values (*lowerTempLimit*, *upperTempLimit*). Each temperature measurement is associated with an instant-type timestamp, denoting the time point that the patient's temperature was measured.

(2) The blood tests for a patient are associated with instant-type timestamps, indicating the date on which the patient took the test; a constraint is defined for blood tests stating that five days after the patient's hospitalisation at least one blood test must have been recorded. If this condition is not met, an alert should be issued for the director of the hospital. Within each blood test, the patient's LDL and HDL cholesterol are measured, together with the number of blood cells. For the values of LDL and HDL cholesterol, it must hold that $HDL > LDL$.

(3) The treatment cost history for each patient is instant-timestamped, with each (*timestamp*, *cost*) pair indicating that the cost for the patient's treatment for the period starting at the patient's admission and ending at the designated timestamp is *cost*. Since updates to the treatment cost incorporate additional costs (e.g. costs for examination tests) to the overall cost, a constraint is defined stating that the overall cost should increase with time. The initial cost for the

patient's treatment is set to a specified value, indicating the fees for the initial medical examinations and the admission committee.

(4) Immediately after an operation, a patient's temperature is expected to be high, but it should progressively drop to normal levels. A history object gives the maximum allowed temperature for each hour after the operation, and this history object is associated with the patient immediately after the operation, to accommodate for severity and complication parameters. We would like to retrieve the patients whose temperature is abnormal, i.e. it exceeds the specified value for *any* measurement performed on the patient.

In the following paragraphs we show how the proposed modelling constructs can be used to fully satisfy the requirements of this example.

(1) Temperature measurements TM have the constraint $cons_{tm}$ that for each day within the hospitalisation period HP at least one measurement must exist. The hospitalisation period HP for a patient is an historical object, since it may be updated either within a single hospitalisation (e.g. the patient's stay is prolonged), or for multiple hospitalisations (the patient leaves the hospital and is then readmitted). The current (most "up-to-date") hospitalisation period for a patient p will be denoted as hp_p and can be defined as follows:

$$hp_p = \{x : (vt_x, x) \in HP \wedge (\forall (vt_i, x_i) \in HP : \max(vt_x) \geq \max(vt_i))\}.$$

The instants tm_p at which the temperature of patient p is measured are given by the expression:

$$tm_p = \{x : (vt_x, x) \in HP\}$$

whereas the days hd_p during which patient p has stayed in the hospital are given by the expression:

$$hd_p = \{x \in hp_p : x < NOW\}.$$

Using the definitions presented above, the constraint $cons_{tm}$ can be modelled using the $CMatch$ function:

$$CMatch(tm_p, hd_p) = true$$

This is a dynamic constraint since the definition of hd_p depends on the NOW quantity, which always evaluates to the current instant. This implies the need for an implementation technique that will handle such conditions.

The additional requirement stating that for each temperature t,

$$lowerTempLimit \leq upperTempLimit$$

can be modelled using the *CMatch* function:

$(\mathit{CrossValUp}(\mathit{TM},\ \mathit{temperature},\ \mathit{upperTempLimit}) = \mathit{false}) \wedge$

$(\mathit{CrossValUp}(\mathrm{TM},\ \text{-}\ \mathit{temperature},\ \text{-}\ \mathit{lowerTempLimit}) = \mathit{false})$.

(2) Blood tests BT have a cardinality constraint as well, but this takes effect only five days after the admission date. This constraint can be formally stated as:

$|BT| \geq 1 \vee (NOW - \text{‘5 days’} < \min(hp_p))$.

This is a dynamic constraint as well, since it contains the NOW quantity. The constraint between the values of LDL and HDL cholesterol can be modelled using the *Vdomin* function as follows:

$\mathit{Vdomin}(BT, HDL, BT, LDL) = \mathit{true}$.

(3) The treatment cost history TC has a cardinality constraint

$|TC| \geq 1$

and a constraint that the overall cost should be always increasing. The latter constraint can be modelled using the *MonInc* function:

$\mathit{MonInc}(\mathrm{TC}, \mathrm{cost}) = \mathit{true}$.

(4) The patient's measured temperature should be *dominated* by the maximum temperature specified after the operation. This can be tested using the *Vdomin* function; however this function requires that both history lists are expressed in the same calendar. This may be facilitated in various ways; in this example we illustrate two approaches:

(a) Reducing both calendars to the chronon calendar, using the *Normalise* function.

(b) Using the $M(H)(x)$ function, to convert the representation of the *MaxTemperature* history (which uses a granularity of hours) to the representation of the *Temperature* history (which uses a granularity of minutes). This assumes that the calendar of hours is based on the calendar of minutes, which is a "normal" choice.

Moreover, we must ensure that only the patient's measurements pertaining to the designated period after the operation are taken into account and that measurements not taken (either skipped or future ones) do not affect the result. Effectively this corresponds to computing the intersection of the two calendars, or, equivalently, filtering each calendar using the other one as a filter. Based on the above, the condition for selecting patients with abnormal temperature behaviour may be formulated as follows:

$$\begin{array}{l} not(Vdomin(\\ \quad filter(Normalise(Temperature),\ Dom(Normalise(MaxTemperature))), \\ \qquad temperature, \\ \quad filter(Normalise(MaxTemperature),\ Dom(Normalise(Temperature))), \\ \qquad MaxTemperature)) \end{array}$$

if the Normalise function is used, or

$$\begin{array}{l} not(Vdomin(\\ \quad filter(Temperature,\ Dom(M(H)(MaxTemperature)),\ temperature, \\ \quad filter(M(H)(MaxTemperature),\ Dom(Temperature)),\ MaxTemperature)) \end{array}$$

if the $M(H)(x)$ function is used.

According to the above, a patient may be represented using the following composite object:

```
Patient = Tuple<Id: String, Name: String,
    Decision: Timestamped<Text, Instant, Day>,
    HospitalisationPeriod: Historical<Period, Period, Day>,
    Temperature: Historical: Historical<Float, Instant, Minute>,
    MaxTemperature: Historical<Float, Instant, Hour>,
    BloodTest: Historical<
            Tuple<HDL: Float, LDL: Float, RedBloodCells: Integer>,
            Instant, Day>,
    Cost: Historical<Integer, Hour>>
```

We use the *Tuple* constructor to introduce a record-structured composite datum, whereas the constructors *Timestamped* and *Historical* introduce single-valued and version-based temporal data. Both constructors accept three parameters, being the type of each history element, the timestamp type and timestamp granularity, respectively. In this scheme, the *Id* field is the "primary key" of any patient collection, i.e. there may not exist two distinct patients p_1 and p_2 such that $p_1.Id = p_2.Id$. Based on this representation we will demonstrate the querying functionality of operators presented in Sections 5.1 and 5.2:

(1) *Project*(*BloodTest, <HDL, LDL>*) may be used to construct a history list containing only the cholesterol measurements.

(2) *Join*(*Temperature, BloodTest*) may be used to produce a *combined history list* of temperature measurements and blood test results that were obtained within the same day.

(3) *Group*(*Temperature, DayCalendar, average*) may be used to compute a history list containing exactly one temperature value per day of hospitalisation. This value will be the average of all temperature measurements that were performed on the particular day.

(4) *GetCount*(*BloodTest*) returns the number of blood tests performed on a specific patient.

8 CONCLUSIONS

In this paper we have presented a a set oriented description of the calendars together with a function oriented history concept with a history-algebra. The advantage of this approach is that it allows calendar transformations with no information loss and permits handling of calendars using standard set operators. The proposed model is proved to be complete with respect to the reduced temporal algebra TA and the expressive power of the proposed model has been demonstrated by a real-world example.

REFERENCES

[1] Bertino, E.—Ferrari, E.—Guerrini, G.: T_Chimera: A Temporal Object-Oriented Data Model. Theory and Practice of Object Systems, Vol. 3, 1997, No. 2, pp. 103–125.

[2] Cattell R. (ed.): The Object Database Standard ODMG-93 (Release 1.1). Morgan Kaufmann Publishers, San Francisco, California.

[3] Chiu, J. S.—Chen, A. L. P.: A note on incomplete relational database models based on intervals. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, 1996, No. 1, pp. 189–191.

[4] Clifford, J.—Croker, A.—Tuzhilin, A.: On completeness of historical relational query languages. ACM Transactions on Database Systems, Vol. 19, 1994, No. 1, pp. 64–116.

[5] Elmasri, R.—Wuu: A Temporal Model and Query Language for ER Databases. Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA, pp. 76–83.

[6] Gadia, S. K.—Yeung, C. S.: A generalized model for a temporal relational database. Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988, pp. 251–259.

[7] Goh, C. H.—Lu, H.—Ooi, B. C.—Tan, K. L.: Indexing Temporal Data Using Existing B+-Trees. Data & Knowledge Engineering, Vol. 18, 1996, pp. 147–165.

[8] Gregersen, H.—Jensen, C. S.: Temporal Entity-Relationship Models — A Survey. IEEE Transactions on Knowledge and Data Engineering, Vol. 11, May/June 1999, No. 3, pp. 464–497.

[9] Jensen, C. (ed.): A Consensus Test Suite of Temporal Database Queries: Technical Report R 93-2034, Department of Mathematics and Computer Science, Institute for Electronic Systems, Aalborg University, 1993.

[10] Kalua, P.—Robertson, E.: Benchmark Queries for Temporal Databases. Technical Report #379, Indiana University, Computer Science Department, 1993.

[11] Kurt, A.—Ozsoyoglu, Z. M.; Modeling and Quering Periodic Temporal Databases. 6th Int. Conf. and Workshop on Database and Expert Systems Applications (DEXA'95) – Workshop Proceedings, pp. 124–133.

[12] Lee, J.—Elmasri, R.—Won, J.: An Integrated Temporal Data Model Incorporating Time Series Concept. Data and Knowledge Engineering, Vol. 24, January 1998, Num. 3, pp. 257–276.

[13] Navathe, S.—Ahmed, R.: A temporal relational model and a query language. Information Sciences, Vol. 49, 1989, Num. 1-3, pp. 147–175.

[14] Rose, E.—Segev, A.: TOOSQL — A temporal object oriented query language. 12th International Conference on the Entity-Relationship Approach, Arlington, Texas, USA, December 15-17, 1993, pp. 122-136.

[15] Schmidt, D.—Dittrich, A. K.—Dreyer, W.—Marti, R.: Time series, a neglected issue in temporal database research? Recent Advances in Temporal Databases (Clifford, J. and Tuzhilin, A., eds.), Proceedings of the International Workshop on Temporal Databases, Zurich, Switzerland, 17-18 September 1995, pp. 214–232.

[16] Snodgrass, R. T.: The temporal query language Tquel. ACM Transactions on Database Systems, Vol. 12, June 1987, Num. 2, pp. 247–298.

[17] Sotiropoulou, A.—Souillard, M.—Vassilakis, C.: Temporal extension to ODMG. Proceedings of the 3rd Biennial World Conference on Integrated Design and Process Technology, vol. 2, Issues and Applications of Database Technology (IADT), Berlin, Germany, 1998, pp. 304–311.

[18] The TSQL2 Language Design Committee: The TSQL2 Temporal Query Language (Snodgrass, R., ed.), Kluwer Academic Publishers, 1995.

[19] Vassilakis, C.—Georgiadis, P.— Sotiropoulou, A.: A comparative study of temporal DBMS architectures. Proceedings of DEXA 96 workshop, Zurich, September, 1996, pp. 153–164.

[20] Vassilakis, C.—Lorentzos, N.—Georgiadis, P.: Implementation of transaction and concurrency control support in a temporal DBMS. Information Systems, Vool. 23, 1998, No. 5, pp. 335–350.

Manuscript received 7 April 1999; revised 13 June 2000

László Kovács is currently an associate professor at the University of Miskolc, Hungary. He received M.S. degree in natural sciences from University of Debrecen, Hungary in 1985 and the Ph.D. degree in mechanical sciences from University of Miskolc in 1998. His main research areas are temporal and semi-structured databases, nearest neighbor queries in metric spaces and the classification using the CPN neural network.

Costas Vassilakis was born in Arta, Greece in 1968. He received his BS. degree and his Ph.D. in Informatics from the University of Athens in 1990 and 1995, respectively. He has participated in national and European projects and he now is a Special Advisor for the General Secretariat for Information Systems of the Ministry of Finance, Greece, and an one-year contract lecturer in the University of Athens. His research interests include databases, distributed systems and object-oriented systems. in Informatics