TEMPOS: A Platform for Developing Temporal Applications on top of Object DBMS

Marlon Dumas and Marie-Christine Fauvet and Pierre-Claude Scholl

January 8, 2001

TR-53

A TIMECENTER Technical Report

Title	TEMPOS: A Platform for Developing Temporal Applications on top of Object DBMS
	Copyright © 2001 Marlon Dumas and Marie-Christine Fauvet and Pierre-Claude Scholl. All rights reserved.
Author(s)	Marlon Dumas and Marie-Christine Fauvet and Pierre-Claude Scholl
Publication History	January 2001. A TIMECENTER Technical Report

Abstract

This paper presents TEMPOS [1], a set of models and languages intended to seamlessly extend the ODMG object database standard with temporal functionalities. The proposed models exploit object-oriented technology to meet some important, yet traditionally neglected design criteria, related to legacy code migration and representation independence.

Two complementary ways for accessing temporal data are offered: a query language and a visual browser. The former one, namely TEMPOQL, is an extension of OQL supporting the manipulation of histories regardless of their representations, by composition of functional constructs. Thereby, the abstraction principle of object-orientation is fulfilled, and the functional nature of OQL is enforced. The visual browser on the other hand, offers operators which support several time-related interactive navigation tasks, such as studying a snapshot of a collection of objects at a given instant, or detecting and examining changes within temporal attributes and relationships.

TEMPOS models and languages have been fully formalized both at the syntactical and the semantical level and have been implemented on top of the O_2 DBMS. Their suitability with regard to applications' requirements has been validated through concrete case studies.

Index terms: temporal databases, temporal data models, temporal query languages, time representation, upward compatibility, object-oriented databases, ODMG.

1 Introduction

Temporal data handling is a pervasive aspect of many applications built on top of Database Management Systems (DBMS). Accordingly, most of these systems provide datatypes corresponding to the concepts of date and span. These datatypes are adequate for modeling simple temporal associations such as the date of birth or the age of a person. However, they are insufficient when it comes to model more complex ones, such as the history of an employee's salary, or the sequence of annotations attached to a video. Since no datatypes dedicated to these kinds of associations are currently provided by DBMS, type constructors such as "list" and "tuple" should be used instead to encode them. The semantics of this encoding must then be integrated into the application programs, thereby increasing their complexity. Temporal database systems aim at overcoming these deficiencies [48, 41, 25, 44].

Research in this area has been quite prolific regarding extension proposals to data models and query languages. Whereas in the relational framework these works have led to the consensus language TSQL2 [42], there is no equivalent result in the object-oriented framework. Early attempts to define temporal extensions of object-oriented data models [41] had a limited impact, essentially due to the absence of a standard underlying data model. As the ODMG [11] proposal was released and started to be adopted by the major object DBMS vendors, a few temporal extensions of it were defined, among which TAU [32], TOOBIS [46] and T_ODMG [5]. However, we argue that these proposals lack at least some of the following four important features:

- Migration support, as to ensure a seamless transition of applications running on top of a non-temporal system to a temporal extension of it.
- Encapsulation of temporal types, as to separate the semantics of temporal data from its internal encoding.
- Formal semantics, to avoid many ambiguities generated by the richness and complexity of temporal concepts, and to serve as a basis for efficient implementation.
- Visual interfaces supporting user tasks such as navigating through a collection of temporal objects.

The goal of the TEMPOS project (Temporal Extension Models for Persistent Object Servers) [7, 21, 27, 22, 19], has been to contribute towards a consensus view on how to handle temporality in object-oriented models, by defining a temporal object database framework integrating the above features. This

[1]contact: Marie-Christine.Fauvet@imag.fr, http://www-lsr.imag.fr/Les.Personnes/Marie-Christine.Fauvet

paper summarizes the results of this effort. The proposed framework is based on a stratified temporal data model, on top of which two interfaces for retrieving and exploring temporal objects are provided: the TEMPOQL query language, and the pointwise temporal object browser.

The paper is structured as follows. Section 2 focuses on defining the requirements related to application migration and representation independence, and shows why existing temporal extensions of ODMG fail to fulfill them. Section 3, describes the TEMPOS data model, and section 4 presents the query language and the visual browser. In section 5 we present the prototype that has been developed to validate the feasibility of our proposal, and enumerate some applications that have been modeled and implemented in TEMPOS. Finally, in section 6 we end with an overview of the proposal and point some future research directions.

2 Motivations and related works

In this section, we present some of the major requirements that guided the design of TEMPOS. These requirements may be divided into two categories: those which deal with the migration of data and application programs from a non-temporal to a temporal environment, and those which deal with the abstract modeling and querying of histories.

2.1 Migration requirements

Most of the temporal data models and languages that have been proposed in the literature, are actually extensions of “conventional” ones. A common rationale for this design choice is that the resulting models can be integrated into existing systems, so that applications built on top of these systems may rapidly benefit from the added technology.

However, the smooth migration of existing data and application programs to temporal database systems may only be achieved if these latter fulfill some elementary compatibility requirements. Surprisingly, such requirements have traditionally been neglected by the temporal database community: it is until relatively recently that notions such as *temporal upward compatibility* [3, 45], have been seriously considered.

Following [3, 45], we are interested in specifying migration requirements between pairs of data models defined as follows.

Definition 1 (*data model*). A data model M is as a quadruple $(D, Q, U, [\![\,]\!]_M)$ composed of a set of database instances D, a set of legal query statements Q, a set of legal update statements U, and an evaluation function $[\![\,]\!]_M$. Given an update statement $u \in U$, a query statement $q \in Q$ and a database instance $db \in D$, $[\![u(db)]\!]_M$ yields a database instance, and $[\![q(db)]\!]_M$ yields an instance of some data structure. □

Hence, a database instance is seen as an abstract entity, to which it is possible to apply updates (statements whose evaluation map a database instance into another one), and queries (statements whose evaluation over a database yield an instance of some data structure). We suppose that the equality is defined over instances of data structures.

We successively introduce two levels of migration requirements: *upward compatibility* and *temporal transitioning support*. The definitions that we provide may be seen as adaptations to the object-oriented framework, of the notions of upward compatibility and temporal upward compatibility introduced in [3, 45].

Definition 2 (*upward compatibility*). A data model $M' = (D', Q', U', [\![\,]\!]_{M'})$ is upward compatible with another data model $M = (D, Q, U, [\![\,]\!]_M)$, iff:

- $D \subseteq D'$, $Q \subseteq Q'$ and $U \subseteq U'$ (syntactical upward compatibility in [3])
- For any db in D, for any q in Q, for any $u_1, u_2, \ldots, u_n$ in U and for any instants $d_1, \ldots d_n, d_{n+1}$: $[\![q^{dn+1}(u_n^{dn}(\ldots u_2^{d2}(u_1^{d1}(db))))]\!]_M = [\![q^{dn+1}(u_n^{dn}(\ldots u_2^{d2}(u_1^{d1}(db))))]\!]_{M'}$

□

Notice that in this latter expression, updates and queries, are parameterized by the instant at which they are issued. This is because, in some temporal data models, operations may depend on the instant at which they are issued as discussed in [14]. Apart from that, the above definition may be applied to any other data model extension.

In the setting of ODMG, the set of queries Q not only includes those which are submitted to the OQL interpreter, but also, all accesses to class extents and object properties via application programs. Similarly, the update operations include object creations and deletions, as well as updates to object properties.

To illustrate upward compatibility, let's consider an ODMG compliant DBMS managing a database about documents and users of a library. Essentially, the database stores information about document loans. Upward compatibility states that if the ODMG DBMS is replaced by a temporal extension of it, or equivalently, if a temporal cartridge is installed, then the data and the application programs accessing these data may be left intact. This implies in particular that the set of database instances recognized by the extension is a super-set of those recognized by the original DBMS, and that the database access and update statements have identical semantics in the original DBMS and in the temporal extension. Notice that at this stage, no notion of temporal support has been considered.

Now, suppose that once the legacy applications run on the temporal extension, the database administrator decides that the history of the loans should be kept, but in such a way that legacy application programs may continue to be operational (at worst they should be recompiled). New applications, on the other hand, should perceive the property as being "historical".

One way of fulfilling this requirement is to offer two different views of the database: a "snapshot view" and a "temporal view", letting the applications choose among them. This leads to a notion of *bi-accessible data model*.

Definition 3 (*Bi-accessible data model*). A bi-accessible data model is a tuple $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^1_M, [\![\,]\!]^2_M)$ such that $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^1_M)$ and $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^2_M)$ are data models. $\square$

A bi-accessible data model $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^S_M, [\![\,]\!]^T_M)$ is said to be *temporal*, if $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^S_M)$ is a snapshot data model and $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]^T_M)$ is a temporal data model, upward compatible with the former.

The second migration requirement that we introduce below, is an answer to the problem raised by the above scenario. It aims at ensuring the continuity of legacy code after the migration from a snapshot to a temporal schema.

Definition 4 (*temporal transitioning support*). Given a function $\mathcal{T}$ which partially or totally transforms a snapshot database instance into a temporal one (i.e. $\mathcal{T}$ attaches temporal support to some classes, attributes and/or relationships in the database's schema), a bi-accessible temporal data model $\mathsf{M_T} = (\mathsf{D_T},\mathsf{Q_T},\mathsf{U_T}$ $[\![\,]\!]^S_{M_\mathsf{T}}, [\![\,]\!]^T_{M_\mathsf{T}})$ is said to offer *temporal transitioning support*, if for any $\mathsf{db} \in \mathsf{D_S}$, for any $\mathsf{q} \in \mathsf{Q_S}$, for any $\mathsf{u_1}, \mathsf{u_2}, \ldots, \mathsf{u_n} \in \mathsf{U_S}$ and for any instants $\mathsf{d_0}, \mathsf{d_1}, \ldots \mathsf{d_n}, \mathsf{d_{n+1}}$: $[\![\mathsf{q}^{\mathsf{dn+1}}(\mathsf{u}_\mathsf{n}^{\mathsf{dn}}(\ldots \mathsf{u}_2^{\mathsf{d2}}(\mathsf{u}_1^{\mathsf{d1}}(\mathsf{db}))))]\!]_{M_S} = [\![\mathsf{q}^{\mathsf{dn+1}}(\mathsf{u}_\mathsf{n}^{\mathsf{dn}}(\ldots \mathsf{u}_2^{\mathsf{d2}}(\mathsf{u}_1^{\mathsf{d1}}(\mathcal{T}^{\mathsf{d0}}(\mathsf{db})))))]\!]^S_{M_T}$ $\square$

While upward compatibility can be achieved by simply adding new concepts and constructs to a model without modifying the existing ones, temporal transitioning support is more difficult to achieve. Indeed, [3] shows that almost none of the existing temporal extensions to SQL, including TSQL2, satisfy this latter requirement (called *temporal upward compatibility* by the authors).

It can be shown that the same remark holds for existing object-oriented temporal extensions, and in particular for the T_ODMG [5] temporal object model. For example, consider a Document class with a property loaned_by defined on it. In the context of T_ODMG, if some temporal support is attached to this property, then any subsequent access to it will retrieve not only the current value of the document's loaned_by property (as in the snapshot version of the database), but also its whole history.

TOOBIS does not exhibit this latter problem. However, in achieving temporal transitioning support, TOOBIS introduces some burden to temporal applications. Indeed, in TOOBIS TOQL for instance, each reference to a temporal property in a query should be prefixed by either keyword valid, transaction or bitemporal. This leads to rather cumbersome query expressions. Similar remarks apply to TOOBIS C++ binding. This approach is actually equivalent to duplicating the symbols for accessing data when adding temporal support, in such a way that for each temporally enhanced property x, there are actually two properties representing it in the database schema, say x and temporal_x. In the example of the library database, this means that when adding temporal support to property loaned_by, this temporal property is actually not modified and instead, a new temporal property is added (say temporal_loaned_by).

We advocate a different approach: when temporal support is added to some component of a database schema S, yielding a new schema S', application programs are divided into two categories: those which view data as if its schema was S, and those which view it with schema S'. Therefore, the problem of temporal transitioning support is seen as a particular case of schema evolution, so that techniques developed in this context apply. The reason for adopting this approach instead of TOOBIS's one, may be stated simply: if a property is modified to add temporal support, temporal applications should perceive this property as being temporal.

2.2 Representation independence

2.2.1 Point-based vs. interval-based temporal associations

A temporal association is as a piece of data locating a fact in the time-line. Temporal associations may be classified into point-based or interval-based, depending on whether they associate facts to instants or to intervals [13]. An interval-based association states that a fact is true during some interval, e.g. "The stock price raised by 50% between 1995 and 1999", without entailing that the fact is true at each instant in the interval, e.g. the above statement does not mean that the stock raised by 50% at each year between 1995 and 1999! On the other hand, a point-based association states that a fact is true at some point in time, observed with some precision (e.g. "The salary of some employee is 5000 at January 1998").

Temporal data models may be classified into point-based or interval-based, depending on whether they manage interval or point-based temporal associations [6]. TSQL2 for instance is point-based: stating that a tuple belongs to a temporal relation during interval [i1,i2), means that this tuple belongs to the relation at each instant between i1 and i2 (i2 excluded). On the other hand, SQL2 enhanced with an ADT for modeling intervals may be considered as interval-based: nothing in the semantics of SQL2 indicates that if a tuple is timestamped with an interval [i1,i2], then the tuple belongs to the relation at each instant between i1 and i2. Some data models are hybrid either because they provide functions for transforming interval-based associations into point-based ones (e.g. IXSQL [33] or SQL/Temporal [45]) or because they distinguish point-based from interval-based associations (e.g. TOOBIS [50]). It is worth noting that no temporal data model is actually purely interval-based. This is because point-based associations are prevalent in most temporal database applications [13]. TEMPOS has been designed as a point-based data model.

2.2.2 Representation of point-based associations

In temporal data models, related temporal associations are grouped into temporal relations (in the relational framework) or into object or attribute histories (in the object-oriented and the object-relational frameworks).

Point-based temporal relations or histories may be represented in several ways. For instance, it is possible to associate an instant timestamp to every tuple in a temporal relation at the logical level [49]. An alternative is to group several value-equivalent tuples into a single one, timestamped either by a temporal element [29], or by an interval. This latter is the most common approach in existing point-based data models. Operators on temporal data are then defined over this interval-timestamped representation, which renders their formalization quite cumbersome. Indeed, in addition to defining the result of the operation itself, the semantics must also describe the way this result is to be encoded. The same remark applies to query expressions, which leads to some undesirable tensions between query expressions and their intended semantics [49].

To illustrate this point, let's consider the query *retrieve all departments where Ed has been since he first moved from the accounting department*, expressed in TOOBIS's TOQL:

```
select distinct D2
from Employees as E, valid E.department as D1, valid E.department as D2
where E.Name = “Ed” and D1.name = “Accounting” and valid(D1) before valid(D2)
```

In this example, there is a clear difference in the level of abstraction between the query expression in natural language and in TOQL: in the natural language formulation, there is no reference to any maximal intervals during which Ed was in the accounting department, whereas valid(D1) and valid(D2) in TOQL's formulation are typical examples of such references. In addition, the notion of "since the first time", does not appear in TOQL's expression. This mismatch reflects some lack of declarativeness in the language, and more precisely, the need for operators on histories allowing one to reason about succession in time. Instead of providing such operators, TOQL, as well as most other temporal query languages, relies on an interval-timestamped representation of histories, together with operators over intervals, for expressing most kinds of temporal queries.

We advocate that exclusively relying on a fixed representation of temporal data to define the semantics of temporal operators, or for query expression, is an undesirable feature in a temporal data model, and especially in an object-oriented one, since it tends to violate some basic principles of object-orientation such as encapsulation. Instead, specific representation independent operators on histories should be provided, covering the fundamental temporal reasoning paradigms: succession, simultaneity, granularity change, etc. In this respect, our approach is different to those adopted in other works such as [40], [42], [32] and [50], and similar to those adopted in [54] and [30].

Nevertheless, we do not mean that manipulating a representation of histories is never useful for query expression. Indeed, some queries naturally involve such representations, e.g. *which employees had a constant salary during an interval of at least three years* [42], so that temporal query languages should also provide means for directly manipulating a particular representation of histories based either on instants, intervals or sets of instants.

The above considerations also apply to the modeling of temporal values. For instance, in most temporal data models, collections of instants are modeled as sets of disjoint non-contiguous intervals, (usually termed *temporal elements*). While this representation is probably adequate in many cases, imposing it at the logical level is useless, and introduces a gap between the modeled concept and the corresponding data model construct.

To summarize this discussion, we state a "representation independence" requirement on data models as follows: a temporal data model is said to be representation independent if (i) it defines all operators on temporal values and associations independently of any representation; (ii) it provides operators accounting for different kinds of temporal reasoning, so that it does not exclusively relies on the representation of temporal data for query expression.

3 The TEMPOS data model

The TEMPOS data model is based on a set of datatypes whose behavior is encapsulated into ODMG type interfaces. ODMG's distinction between interfaces (abstract type descriptions) and classes (concrete implementations) is exploited to enforce the separation between the semantics of the operators over these datatypes, and their implementation under some fixed representation.

TEMPOS is structured into three increasingly sophisticated levels. This enables a particular implementation to choose a degree of compliance, according to the requirements of the targeted applications and the extensibility of the underlying DBMS.

- The first level is composed of a set of datatypes modeling time values (i.e. instants, durations, sets of instants and intervals), expressed at multiple granularities.
- The second level introduces the concept of *history*.
- The third level extends the concepts of class, attribute and relationship, as defined in the ODMG standard. This leads to the concepts of *temporal class*, *temporal attribute* and *temporal relationship*. This level is designed so as to ensure temporal transitioning support, as defined in section 2.1.

3.1 Modeling time values and histories

3.1.1 Time model

Time units We adopt a discrete, linear and bounded time model in which time is structured in a *multi-granular* way [15, 53] by means of *time units*. A time unit is a partition of the time line into a set of convex sets: each of the elements of this partition is then seen as an atomic *granule* and every point of the time-line is approximated by the granule which contains it. Thus a time unit defines the precision at which time is observed. The granules of a time unit are numbered by natural integers: the order among these integers defines the notion of succession in time and the distance between them defines the notion of duration.

If a mapping can established between each granule of a time unit u1 and a set of consecutive granules of another unit u2, u2 is said *finer than* u1 ($u1 \prec u2$), or conversely, u1 is said *coarser* than u2. For instance, the time unit *month* is finer than the unit *year* but coarser than the unit *day*, because each year contains an integer amount of months and each month contains an integer amount of days. The *finer than* relation is a partial order because some pairs of units are not comparable (e.g. units *month* and *week*). We assume that there is a unique finest unit which as usual is called the *chronon*.

For each pair of time units ⟨u1, u2⟩ such that u1 ≺ u2, two conversion functions are defined: one for *expanding* a granule of the coarser unit (u2) into an interval of granules of the finer one (u1) (noted $\epsilon_{u2,u1}$), and the other for *approximating* a granule of u1 by a granule of u2 (noted $\alpha_{u1,u2}$), as shown in figure 1. Units u1 and u2 (u1 ≺ u2) are said to be *regular* if the intervals of granules generated by $\epsilon_{u2,u1}$ have all the same cardinality.

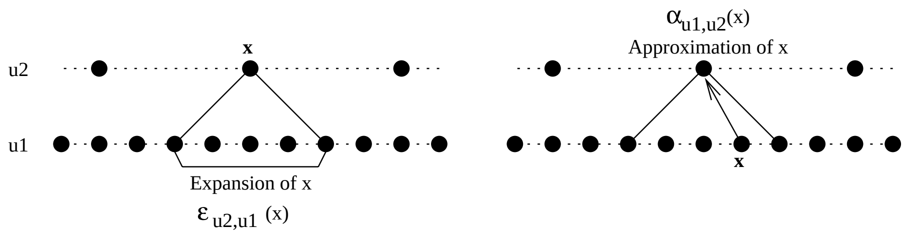


Figure 1: unit expansion and approximation

A *unit system* is a a sequence of comparable units in decreasing order according to the *finer than* relation, e.g. [Year, Month, Day]. As detailed in the next paragraphs, unit systems are used to express temporal values. The finest unit of a unit system defines the precision at which temporal values are expressed in this system.

Basic temporal types A *duration* is a number of granules of a time unit measuring an amount of time at the precision defined by this unit. Hence, values of type duration are pairs made of an integer and a time unit. Durations are signed so as to differentiate forward from backward motion of time. *Instant relative* *durations* (or *relative durations* in short) are expressed according to a unit system as in "1 month and 10 days". The qualifier *relative* points out the fact that the conversion in the finest unit of the unit system depends on the instant to which the duration is related, unless units are mutually regular as in "1 year and 3 months", in which case they can be mapped into absolute durations.

An *instant* is as an approximation of a connected region of the time line by a granule of an unit, called its *observation unit*. An instant is a point with respect to its observation unit. However, it may be seen as an interval with respect to a finer one. As durations, instants may be viewed as pairs composed of an integer and a unit, but unlike them, their semantics is defined with respect to some origin on the time line.

Several operators are defined on instants and durations: predicates related to the chronological order, addition and subtraction of durations, distance function between two instants, addition of instants and durations, and conversion from one unit to another. [8, 7] analyze the cases where these operations may deal with multi-granular arguments.

Input and output of instants and durations are managed through an extensible set of *formats*. Abstractly, a *format* is defined as a mapping from a regular language (i.e. a set of words recognized by a regular expression) to a set of temporal values at a given granularity. At the concrete level, a format is composed of a regular expression which describes the syntax of the recognized strings, a unit system which determines the interpretation of the numerical values or other granule references that appear in the string, and a permutation which allows to put these granule references in a canonical order. Indeed, the order of the granule references in the European date format is not the same as in the American one, so that "1/2/1998" means "February the 1st 1998" in Europe and "January the 2nd 1998" in the US. For a more detailed description of the notion of format, the reader may refer to [8, 7].

Temporal sequences At an abstract level, a temporal sequence (TSequence) is defined as a finite, chronologically ordered sequence of instants observed at some granularity. Since the instants in the sequence are canonically ordered, this notion models a *set of instants*. Among the various representations of temporal sequences, we distinguish two : sequences of instants (ISequences), which are in fact extensional representations of temporal sequences, and coalesced sequences of intervals (DSequences). Moreover, we distinguish two particular kinds of temporal sequences, whose characteristics allow to define specific operators over them: periodic sequences of instants (PSequences) and intervals. The latters are in fact particular cases of the formers, since an interval is a periodic sequence of instants with period one.

The specification of these types and their operators may be found in [8, 7]. These operators include classical set operators, constructors, and comparison operators. Comparison operators on intervals are defined on the basis of Allen's interval relations [2]. All these operators are defined independently of any particular representation so as to fulfill the representation independence requirement stated in section 2.

3.1.2 Historical model

At an abstract level, a history is defined as a function from a finite set of instants to a set of values of a given type. The domain and the range of a history are respectively called its *temporal* and *structural* domain.

In the sequel, we formally describe the types and operators related to histories. This description uses functional notations, since most of the operators on histories are higher-order operators (i.e. functions whose parameters may themselves be functions), and a simple ODMG-like description of them would not be accurate enough.

The following notations are used: $\mathsf{T1} \rightarrow \mathsf{T2}$ stands for the type of all functions with domain T1 and codomain T2. $\{\mathsf{T}\}$ and $[\mathsf{T}]$ respectively denote the type of sets of T and sequences of T. $\langle \mathsf{T1}, \mathsf{T2}, \ldots, \mathsf{Tn} \rangle$ designates the the type of tuples whose i th component is of type Ti ($1 \leq i \leq n$); tuple components may be labeled using the notation $\langle \mathsf{L1} : \mathsf{T1}, \mathsf{L2} : \mathsf{T2}, \ldots, \mathsf{Ln} : \mathsf{Tn} \rangle$. $\mathsf{T1}{<}\mathsf{T2}{>}$ denotes an instantiation of the parameterized type T1 with type T2: in particular, $\mathsf{History}{<}\mathsf{T}{>}$ denotes the type of histories with structural values of type T. $\langle$v1, v2, ... , vn$\rangle$ denotes a tuple value whose ith component is vi $(1 \leq i \leq n)$. Finally, if x is an instant or a temporal sequence, then Unit(x) denotes its observation unit.

The following specification introduces the History ADT and its elementary selectors.

type History$<$T$>$ = Instant $\rightarrow$ T
TDomain: History$<$T$>$ $\rightarrow$ TSequence */* retrieves the temporal domain */*
Unit: History$<$T$>$ $\rightarrow$ Unit */* Unit(h) = Unit(TDomain(h)) */*
SValue: History$<$T$>$, Instant $\rightarrow$ T */* SValue(H, I) is the structural value at instant I */*
/ precondition: I $\in$ TDomain(H) */*
SDomain: History$<$T$>$ $\rightarrow$ { T } */* retrieves the structural domain */*

Histories may be represented in several ways, mainly by means of collections of timestamped values, termed *chronicles*. Among these representations, some are useful for query expression, so that specific operators are defined, allowing one to convert a history into a chronicle. Concretely, a history may be represented by at least three kinds of chronicles:

- Instant-based representation: chronologically ordered list of instant-timestamped values, e.g. [$\langle$1, v1$\rangle$, $\langle$2, v1$\rangle$, $\langle$4, v1$\rangle$, $\langle$5, v2$\rangle$, $\langle$6, v2$\rangle$, $\langle$7, v2$\rangle$, $\langle$8, v3$\rangle$, $\langle$9, v1$\rangle$, $\langle$10, v1$\rangle$]. Such lists are termed IChronicles.
- Interval-based representation: chronologically ordered, coalesced list of interval-timestamped values, e.g. [$\langle$[1..2], v1$\rangle$, $\langle$[4..4], v1$\rangle$, $\langle$[5..7], v2$\rangle$, $\langle$[8..8], v3$\rangle$, $\langle$[9..10], v1$\rangle$]. This kind of list is called an XChronicle.
- TSequence-based representation: set of distinct values timestamped by disjoint temporal sequences, e.g. { $\langle${1, 2, 4, 9, 10}, v1$\rangle$, $\langle${5, 6, 7}, v2$\rangle$, $\langle${ 8 }, v3$\rangle$ }, which are termed DChronicles.

The following operators are provided to switch from histories to either of these representations and vice-versa.

IHistory : [$\langle$tvalue : Instant, svalue : T$\rangle$] $\rightarrow$ History$<$T$>$
/ Precondition: let [IS_1, ... ,IS_n] be the parameter of a call to operator IHistory: $\forall$k $\in$ [1..n$-$1] (Unit(IS_k.tvalue) = Unit(IS_{k+1}.tvalue) $\wedge$ IS_k.tvalue $<$ IS_{k+1}.tvalue) */*
XHistory : [$\langle$tvalue : Interval, svalue : T$\rangle$] $\rightarrow$ History$<$T$>$
/ Precondition: let [XS_1, ... ,XS_n] be the parameter of a call to operator XHistory: $\forall$k $\in$ [1..n-1] (Unit(XS_k.tvalue) = Unit(XS_{k+1}.tvalue) $\wedge$ XS_k.tvalue $<$ XS_{k+1}.tvalue $\wedge$ (XS_k.tvalue meets XS_{k+1}.tvalue $\Rightarrow$ XS_k.svalue $\neq$ XS_{k+1}.svalue)) */*
DHistory : { $\langle$tvalue : TSequence, svalue : T$\rangle$ } $\rightarrow$ History$<$T$>$
/ Precondition: let SDS be the parameter of a call to DHistory: $\forall$ DS, DS' $\in$ SDS (DS $\neq$ DS' $\Rightarrow$ Unit(DS.tvalue) = Unit(DS'.tvalue) $\wedge$ DS.tvalue $\cap$ DS'.tvalue $= \emptyset$ $\wedge$ DS.svalue $\neq$ DS'.svalue) */*
IChronicle: History$<$T$>$ $\rightarrow$ [$\langle$tvalue : Instant, svalue : T$\rangle$]
/ IHistory(IChronicle(h)) = h */*
XChronicle: History$<$T$>$ $\rightarrow$ [$\langle$tvalue : Interval, svalue : T$\rangle$]
/ XHistory(XChronicle(h)) = h */*
DChronicle: History$<$T$>$ $\rightarrow$ {$\langle$tvalue : TSequence, svalue : T$\rangle$}
/ DHistory(DChronicle(h)) = h */*

3.1.3 Algebraic operators on histories

Algebraic operators on histories are classified into two categories: *intra-point* and *inter-point*. An operator is said intra-point if the structural value of the resulting history at a given instant depends exclusively on the structural value of the argument histories at that instant, otherwise it is said to be inter-point (see figure 2). Notice that this classification is closed under composition: the composition of two intra-point operators yields an intra-point operator and the same is true of inter-point operators.

The semantics of each operator is formally described below by means of a first-order calculus-like expression defining the graph of the resulting history (a set of pairs $\langle$instant, value$\rangle$) in terms of that of the argument(s).

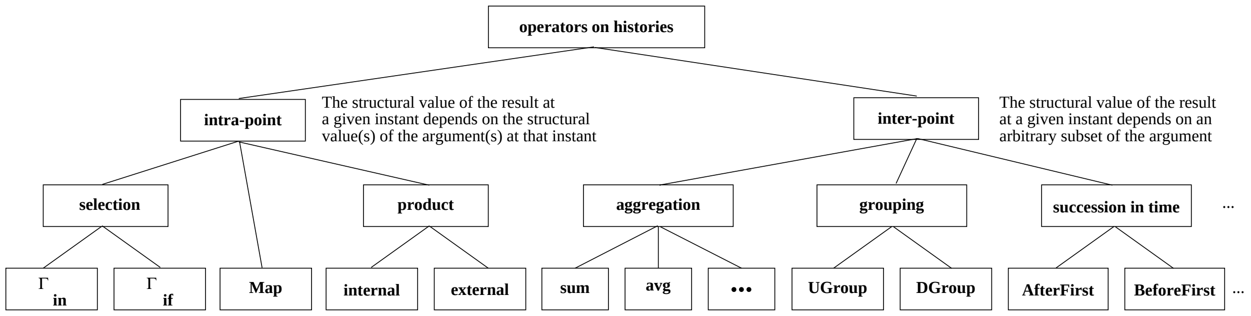

operators on histories
intra-point
The structural value of the result at a given instant depends on the structural value(s) of the argument(s) at that instant
inter-point
The structural value of the result at a given instant depends on an arbitrary subset of the argument
selection
product
aggregation
grouping
succession in time
...
Γ in
Γ if
Map
internal
external
sum
avg
•••
UGroup
DGroup
AfterFirst
BeforeFirst
...

Figure 2: Taxonomy of algebraic operators on histories

Intra-point operators Intra point operators Γ_{if} and Γ_{in} restrict the temporal domain of a history to those instants at which a given condition is true.

- $\_\,\Gamma_{\text{if}}\,\_$: History<T>, (T → boolean) → History<T> */* $h\ \Gamma_{if} P = \{\langle I,v\rangle \mid \langle I,v\rangle \in h \wedge P(v)\}$ */*
- $\_\,\Gamma_{\text{in}}\,\_$: History<T>, TSequence → History<T> */* $h\ \Gamma_{in} S = \{\langle I,v\rangle \mid \langle I,v\rangle \in h \wedge I \in S\}$ */*

The intra-point operator Map on the other hand, applies a given function to each structural value of a history.

Map: History<T>, (T → T') → History<T'> */* $Map(h, f) = \{\langle I,f(v)\rangle \mid \langle I,v\rangle \in h\}$ */*

A temporal join is a merging of two histories. Since two histories may have different temporal domains, we distinguish the *inner* temporal join ($*_\cap$) from the *outer* one ($*_\cup$), depending on whether the resulting history's temporal domain is the intersection or the union of the temporal domains of the arguments

The inner temporal join of two histories ($h1 *_\cap h2$) is a history whose structural values are pairs obtained by combining "synchronous" values of h1 and h2 (i.e. values attached to the same instant). The outer temporal join ($h1 *_\cup h2$), is similar to the corresponding inner temporal join, except that it attaches structural values of the form ⟨v, Nil⟩ or ⟨Nil, v⟩, to those instants where one of the argument histories is defined while the other is not. Here, Nil denotes the neutral element of the history's structural domain type (e.g. 0 for integers, nil for objects, etc.). More precisely:

- $\_ *_\cap \_$: History<T1>, History<T2> → History<⟨T1,T2⟩>
  / $h1 *_\cap h2 = \{\langle I, \langle v1, v2\rangle\rangle \mid \langle I, v1\rangle \in h1 \wedge \langle I, v2\rangle \in h2\}$ */*
  / precondition: Unit(h1) = Unit(h2) */*
- $\_ *_\cup \_$: History<T1>, History<T2> → History<⟨T1,T2⟩>
  / $h1 *_\cup h2 = h1 *_\cap h2 \cup \{\langle I,\langle v1, Nil\rangle\rangle \mid \langle I,v1\rangle \in h1 \wedge I \notin TDomain(h2)\} \cup \{\langle I,\langle Nil, v2\rangle\rangle \mid \langle I,v2\rangle \in h2 \wedge I \notin TDomain(h1)\}$ */*
  / precondition: Unit(h1) = Unit(h2) */*

Among the algebraic operators on sets, the intersection and the difference may be straightforwardly extended to histories by applying them to their graphs, since the resulting set also describes the graph of a history. The operators defined thereof are intra-point. The same process is not applicable to the union operator. Indeed, given two histories h1 and h2 having different values (v1 and v2) at some instant, what would be the value of $h1 \cup h2$ at that instant? Choosing v2 in this context yields the asymmetric union operator defined below.

- $\_ \cup_+ \_$: History<T>, History<T> → History<T>
  / $h1 \cup_+ h2 = \{\langle I, v\rangle \mid \langle I, v\rangle \in h2 \vee (\langle I, v\rangle \in h1 \wedge I \notin TDomain(h2))\}$ */*
  / precondition: Unit(h1) = Unit(h2) */*

Inter-point operators Inter-point operators on histories include aggregation, grouping, and operations dealing with succession in time.

Cumulative aggregate operators on histories compute, for each instant in the temporal domain of the history, an aggregate over all the structural values attached to instants preceding it. For instance, cumulative_sum applied to a history denoting the daily production of a product over a month yields, for each day in this month, the total amount of items of that product produced between the beginning of the month and that day. Aggregate operators such as sum, avg, max, min and duration may be straightforwardly expressed using the corresponding cumulative aggregate operators.

There are two grouping operators in TEMPOS: a unit-based grouping (UGroup) and a duration-based grouping (DGroup).

The unit-based temporal grouping operation UGroup(h, u2), h being at granularity u1 ($u1 \prec u2$), divides up h into groups according to unit u2. The result is a history (at granularity u2) of histories (at granularity u1), whose value at instant i is the temporal restriction of h to interval expand(i, u1) (see figure 3)[2]. Formally :

UGroup: History<T>, Unit $\rightarrow$ History<History<T>>
/ UGroup(h, u)* $= \{ \langle I, subh \rangle \mid \exists I' \in TDomain(h),\ approx(I',u) = I \wedge h\ \Gamma_{in}\ expand(I, Unit(h)) = subh \}$ **/*
/ precondition: Unit(h)* $\prec$ *u */*

On the other hand, the duration-based grouping DGroup(h, d), yields all convex sub-histories of h having duration d. The resulting history associates to instant i the restriction of h to interval $[i..i + d]$, if d is positive, or to $[i + d..i]$ if d is negative, provided that the corresponding interval is completely included in the temporal domain of h. In TSQL2's terminology one may say that if the duration is positive, then it works as a *leading value*, and otherwise it works as a *trailing value* [42].

DGroup : History<T>, Duration $\rightarrow$ History<History<T>>

$$/*\ DGroup(h, d) = \begin{cases} \{ \langle I, subh \rangle \mid [I..I + d] \subseteq h \wedge subh = h\ \Gamma_{in}\ [I..I + d] \} \text{ if d positive} \\ \{ \langle I, subh \rangle \mid [I..I + d] \subseteq h \wedge subh = h\ \Gamma_{in}\ [I + d..I] \} \text{ if d negative} \end{cases} */$$

/ preconditions: Unit(d) = Unit(h) or Unit(d) and Unit(h) are regular */*

This operation is useful to express moving window queries, as in *"compute the average sales for each seven-day period in the history of daily sales of a store"*.

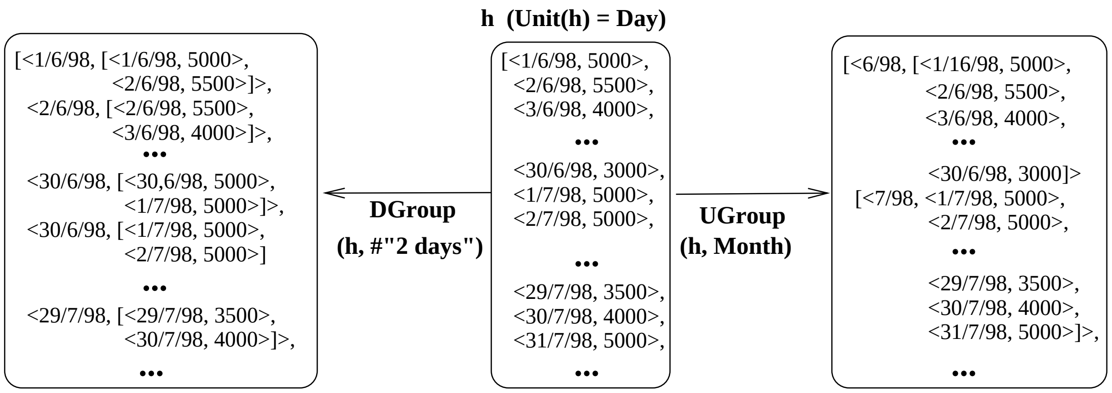


Figure 3: Unit-based and duration-based temporal grouping

To reason about successive values of histories and their correlations, TEMPOS provides four operators, which are in fact algebraic versions of the "sometime" operator of temporal logics [24], namely AfterFirst,

[2]Operation expand(i, u1) maps and instant i observed at some unit u2, to an interval at a finer unit u1. This corresponds to the expansion function $\epsilon_{u2,u1}$ defined over granules in section 3.1.1

BeforeFirst, AfterLast and BeforeLast. AfterFirst(h, P) yields the sub-history of h starting at the first instant at which the value of h satisfies predicate P, or the empty history if such instant does not exist. BeforeFirst(h, P), on the other hand, restricts h to those instants preceding the first instant at which the value of h satisfies P, or h if such instant does not exist. For any history h and any predicate P, h is equal to the union of BeforeFirst(h, P) and AfterFirst(h, P) (which are disjoint). Similar remarks apply to AfterLast and BeforeLast which are defined symmetrically.

AfterFirst: History$<$T$>$, (T $\rightarrow$ boolean) $\rightarrow$ History$<$T$>$
  / AfterFirst(h, P) =* $\{ \langle I, v\rangle \mid \langle I, v\rangle \in h \wedge \exists \langle I', v'\rangle \in h\ (P(v') \wedge I \geq I') \}$ **/*
BeforeFirst: History$<$T$>$, (T $\rightarrow$ boolean) $\rightarrow$ History$<$T$>$
  / BeforeFirst(h, P) =* $\{ \langle I, v\rangle \mid \langle I, v\rangle \in h \wedge \neg\exists \langle I', v'\rangle \in h\ (P(v') \wedge I \geq I') \}$ **/*

3.1.4 Boolean pattern-matching

Pattern matching queries involve retrieval of data by comparison to a standard of some kind. To some extent, all query languages, and temporal query languages in particular, allow to express queries based on simple sequencing patterns: *retrieve all employees who have worked in assembly lines L1 and L2*. The goal here is to express more complex sequencing patterns such as *retrieve all employees who have worked in assembly line L1, then in assembly line L2 during at least 3 months and then, in a 1 month delay, have worked in assembly line L1 again*. When applied to histories, such patterns provide a powerful tool for reasoning about succession in time, which is a fundamental issue in temporal databases.

TEMPOS offers a pattern-matching language [21] that takes into account this structure, therefore providing a powerful tool for expressing this kind of queries. The syntax and the semantics of this language are formally described in appendix B. It includes the following operators :

- Sequencing (operator ";"): for example, the pattern "production $<$ 100 ; production $>$ 200" characterizes *assembly lines whose production is less than 100 immediatly before being greater than 200*.
- Repetition (operator "$+$") allows to reason on repetition of a pattern: for example, the pattern "production $<$ 100; (production $>$ 200 or production = 0)$^+$; production $<$ 100" is used to retrieve assembly lines whose production was less than 100, then null or greater than 200 and subsequently less than 100 again.
- Duration constrained repetition: for example, the pattern "(production = 0)$^{>2\text{ days}}$" characterizes assembly lines whose production was null during more than 2 days.

The pattern-matching operator Matches on histories, checks if a given history contains at least one occurrence of a pattern (see appendix B for its formal definition):

Matches: History (T), Pattern $\rightarrow$ boolean
  / Matches (h, p) = true* $\Leftrightarrow \exists$ *i, j* $\in$ *TDomain (h),* $<$*TDomain (h),* $\mathcal{M} >, \nu, i, j >\models$ *p. */*

Intuitively, $<$TDomain (h), $\mathcal{M} >, \nu, i, j >\models$ p iff there is an occurrence, in h, of the pattern defined by p starting at i and ending at j (i and j being in TDomain(h)). More precisely, $\mathcal{M}$ is the sequence of interpretations in which relation symbols and global constant symbols are interpreted as usual, function symbols are interpreted by the corresponding functions, attributes and methods in the database schema, and the only local constant symbol appearing in pattern p is interpreted at each instant I $\in$ TDomain(T) by the structural value of h at instant I. Valuation ν is determined by the context (program or query).

A similar language to the above one has been proposed in [12]. The main originality of TEMPOS' pattern description language with respect to this latter, lies on the use of regular expression operators, i.e. sequencing, repetition with or without time constraints, and disjunction. The use of regular expression operators leads to a declarative semantics, and provides a framework for efficient implementation.

3.2 Temporal properties and classes

In this section, we extend the basic abstractions of ODMG's object model (class, property and their instances) to integrate temporal support. Throughout the presentation, we show how this extension fulfills the requirements formulated in section 2.

3.2.1 Temporality at the property level

Following the ODMG conventions, a property is defined as an attribute or traversal path of a relation attached to some class. For instance, possible properties of an Employee class include salary and department. Instances of properties are attached to objects, so that *property instances* are to objects what *properties* are to classes[3].

The following two paragraphs successively describe: (1) how temporal support is attached to properties? and (2) what is the effect of attaching such support on the values taken by property instances?

Temporal properties In TEMPOS, a property may be either *temporal*, in which case its successive values are meaningful and thus recorded, or *fleeting*, if only its most recent value is meaningful. When a property is temporal, the granularity at which its evolution is observed is determined by a specific characteristic of the property, namely its *observation unit*.

As in ODMG, a type is attached to a property. In the case of a fleeting property, this type defines the domain of possible values that an instance of this property may take, whereas for a temporal property, it models the *structural values* that an instance of this property may take at some instant. If the structural type of a temporal property is T, each of its instances has a history whose type is History<T>.

The *temporal dimension* of a temporal property determines the semantics of the temporal associations that it models. It may be *valid-time* or *transaction-time* depending on whether the facts are timestamped with respect to the modeled reality or with respect to the database evolution [43]. TEMPOS therefore distinguishes *valid-time* and *transaction-time* properties. By merging the concepts and operators defined on valid-time and transaction-time properties, it is possible to model bitemporal properties, although we do not address this issue in this paper.

Table 1 enumerates the characteristics of temporal properties. The notions of semantic assumption and padding value referenced on it will be explained in the next paragraph.

characteristics	**possible values**
Observation unit	minute / day / month / etc.
Structural type	real / instant / Person / etc.
Temporal dimension	valid time / transaction time
Semantic assumption	discrete / stepwise / linear / etc.
Padding value	0 / nil / etc.

Table 1: characteristics of temporal properties

Instances of temporal properties The *temporal domain* of a temporal property instance models the set of instants (i.e. the TSequence) during which the property is observed for a given object. Temporal domains may evolve dynamically according to the system clock.

The *history* of a temporal property instance is a history reflecting the values taken by the property instance at all instants when it is observed. The temporal domain of this history is equal to the temporal

[3]In ODMG's release 1.2 and subsequent releases, the term *property instance* has been replaced by *object property value*. We choose however to use the old term since it highlights the fact that a property instance is actually a variable, whose state is described by a value.

domain of the property instance itself. Its structural values, on the other hand, are either defined by some update, or derived from the inputted values using a *semantic assumption*.

More precisely, each temporal property instance has an *effective history*, corresponding to the inputted timestamped values attached to it. The effective history is contained in (but not necessarily equal to) the property instance's history, and the difference between them is called the *potential history* (i.e. the part of the history calculated using the semantic assumption).

In the sequel, we will refer to the temporal domain of a property instance's effective history as its *effective temporal domain*, or *effective domain* in short.

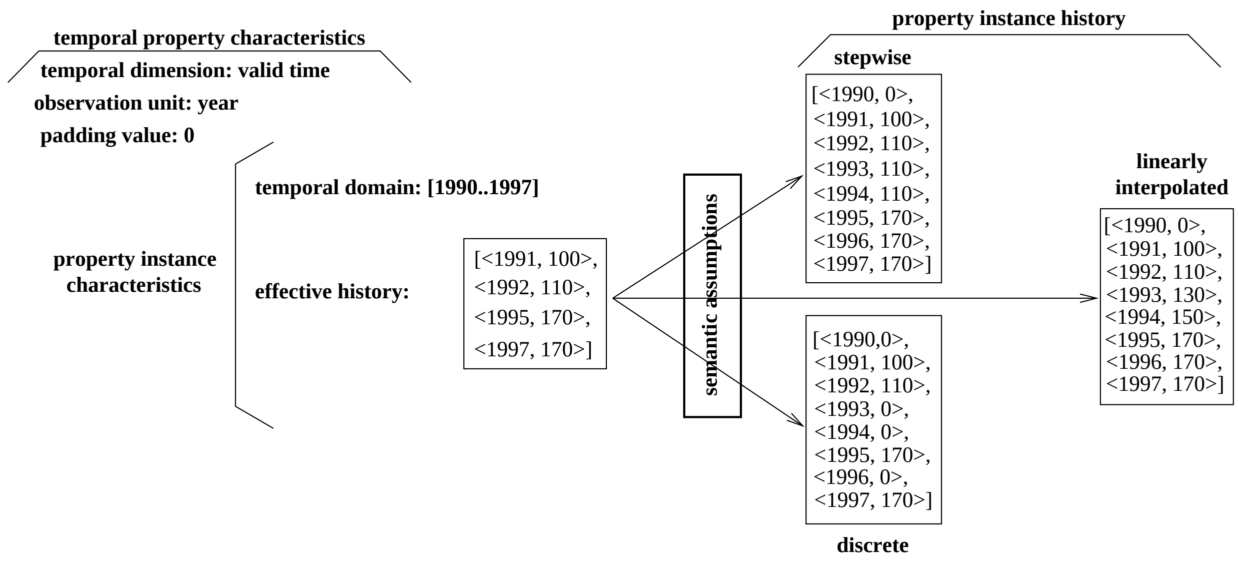


Figure 4: a valid-time property instance's history is calculated dynamically based on the effective history and the semantic assumption

We distinguish three particular semantic assumptions depending on the intended calculation mode of the potential history (see figure 4).

- *Discrete*: the structural value of the potential history is equal to the neutral value of its structural type (e.g. 0 for integers, nil for objects). This is the case of the production of some product in a factory: the period of time during which the production is defined (i.e. its temporal domain) may be known in advance (e.g. all week-days), but at some days, it may be that there is no inputted value (e.g. due to a strike), so that the effective history is not defined for those days. A *padding value* may be attached to a discrete property, to override the use of the neutral element of the structural type as the structural value of the potential histories. This facility is similar to that of attaching "initialization" values to properties in ODMG's ODL.
- *Stepwise*: structural values are "stable" between two instants in the effective temporal domain (e.g. a property instance modeling an employee's salary). As with discrete properties, a padding value is attached to a stepwise property to set the structural value of its instances at those instants for which the stepwise semantic assumption does not provide one (e.g. when the smallest instant in the effective domain is not equal to the smallest instant in the temporal domain).
- *Linearly interpolated*: this kind of interpolation applies only to numerically-valued properties. Between two successive instants in the effective domain, the structural value varies linearly. A padding value may also be attached to this kind of properties.

Transaction-time properties have a stepwise semantic assumption. In addition, they have the peculiarity that the temporal domains of their instances may evolve with the system clock. Consequently, each time that a transaction-time property instance is accessed, its temporal domain is computed by replacing its upper bound by the current instant[4]. The overall process is depicted in figure 5.

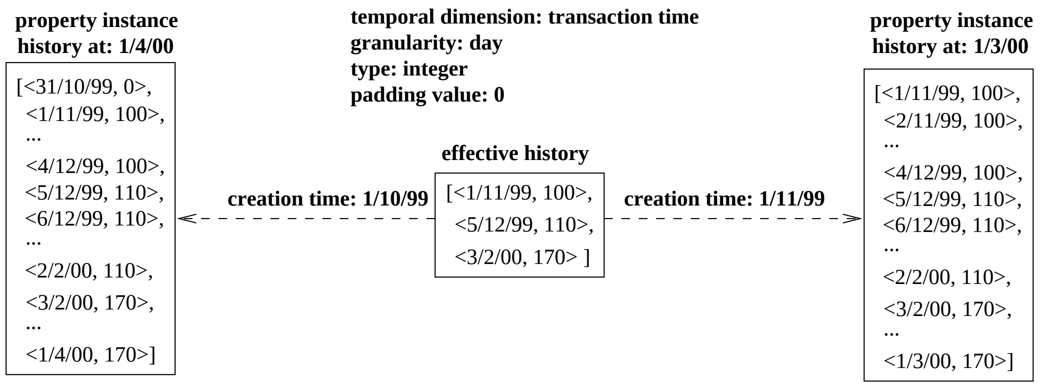


Figure 5: a transaction-time property instance's history is calculated dynamically based on the effective history, the creation time and the current time

Temporal binary relationships In the ODMG's data model, a *relationship* is defined implicitly by the declaration of a pair of inverse properties attached to the class(es) participating in the relationship. TEMPOS extends this notion of inverse properties to model *temporal relationships*. More precisely, if P1 and P2, respectively attached to classes C1 and C2 are the inverse properties defining a temporal relationship, then the constraints listed below hold. The following notations are used: extent(C) is the set of all instances of class C, and o.P is the history of property instance P attached to object o.

- if P1 and P2 are both single-valued then:

  (∀ o1 ∈ extent(C1), ∀ i ∈ TDomain(o1.P1) i ∈ TDomain(SValue(o1.P1, i).P2)
  ∧ o1 = SValue(SValue(o1.P1, i).P2, i))
  ∧ (∀ o2 ∈ extent(C2), ∀ i ∈ TDomain(o2.P2) i ∈ TDomain(SValue(o2.P2, i).P1)
  ∧ o2 = SValue(SValue(o2.P2, i).P1, i))

- If P1 is single-valued and P2 is multi-valued then:

  (∀ o1 ∈ extent(C1), ∀ i ∈ TDomain(o1.P1) i ∈ TDomain(SValue(o1.P1, i).P2))
  ∧ o1 ∈ SValue(SValue(o1.P1, i).P2, i)
  ∧ (∀ o2 ∈ extent(C2), ∀ i ∈ TDomain(o2.P2), ∀ o1 ∈ SValue(o2.P2, i) i ∈ TDomain(o1.P1)
  ∧ o2 = SValue(o1.P1, i))

  The case where P1 is multi-valued and P2 is single-valued is symmetric to this latter.
- If P1 and P2 are both multi-valued then:

  (∀ o1 ∈ extent(C1), ∀ i ∈ TDomain(o1.P1), ∀ o2 ∈ SValue(o1.P1, i)
  i ∈ TDomain(o2.P2) ∧ o1 ∈ SValue(o2.P2, i))
  ∧ (∀ o2 ∈ extent(C2), ∀ i ∈ TDomain(o2.P2), ∀ o1 ∈ SValue(o2.P2, i) i ∈ TDomain(o1.P1)
  ∧ o2 ∈ SValue(o1.P1, i))

3.2.2 Temporality at the class level

As properties, classes may be fleeting or temporal. A temporal class keeps track of its extent, by associating to each of its instances the set of instants at which it is *observed*, either with respect to valid or transaction time. Instances of temporal classes are called *temporal objects*.

[4]Unless the property is "turned off" as discussed later.

In accordance with ODMG, the *extent* of a class (whether fleeting or temporal) is defined as the set of all instances of this class having been created and not deleted. Due to the "append-only" semantics of transaction-time (i.e. no information may be lost), an object of a transaction-time class may not be deleted from its extent[5]. For this reason, the operator delete is overloaded when applied to transation-time objects as discussed later.

In addition to the notion of extent, two other notions are introduced that apply to temporal classes and their instances: observation temporal domain and observed extent.

Definition 5 (*Observation temporal domain*). A valid-time (respectively transaction-time) object, has a valid-time (resp. transaction-time) *observation temporal domain* attached to it, which is an arbitrary set of instants. □

Definition 6 (*Observed extent*). Given a temporal class, the *observed extent* of this class at an instant i, is the subset of the extent consisting of all objects whose observation domain contains i. □

Conceptually, the observation temporal domain (or *observation domain* in short) of a valid-time or transaction-time object, is the set of instants at which the information conveyed by this object is observed. The notion of "observation" may either be defined with respect to transaction-time (when is the information conveyed by an object observed in the database?), or to valid-time (when is the information about the entity modeled by an object observed?).

For instance, consider a class Product modeling the product types produced and sold by a company. If the class is declared as temporal (either with respect to valid-time or transaction-time), then the observation domain could be used to model the time when a particular product is produced, or the time when it is sold. Suppose now that the observation domain models the time when the bottle is produced. If the class is transaction-time, then the value of the observation domain of an object of this class captures the time when the database knows that a product is produced, whereas if the class is valid-time, it models the time when the corresponding product is actually produced in reality.

Temporal support on properties and class extents are orthogonal, i.e., a fleeting class may have transaction-time or valid-time properties, and reciprocally, a valid-time class or transaction-time class may have fleeting properties.

3.2.3 Updating and accessing temporal property instances

In ODMG, there is one "access" and one "update" operator for property instances, respectively get_value, which retrieves the value of the property instance, and set_value which assigns to it the value given as parameter.

In the context of temporal property instances, the set of updating and accessing operators is more complex, due to the variety of temporal characteristics attached to them and the need to achieve bi-accessibility (see section 2). These operators are classified depending on whether they are intended to modify the temporal domain or the effective history, and depending on the time dimension (valid-time or transaction-time) to which they apply.

Evolution of the temporal domain of transaction-time property instances Since transaction-time is intended to model the evolution of the database, the temporal domain of transaction-time properties instances evolve automatically with respect to the database time (i.e. the system clock). Conceptually, the current instant is added to the temporal domain of a transaction-time property instance at each system clock tick. This automatic evolution of the temporal domain can be overridden at any time, e.g. to model the fact that the property instance is not observed during some period of time. This is achieved through the notion of *growth*

[5]In fact, the same remark applies to any object participating in a transaction-time relationship or referenced by a transaction-time attribute

status, which takes one of two values: On or Off. If the value of the growth status of a transaction-time property instance is On, its temporal domain evolves with the system clock. Otherwise, it does not evolve at all. Operators turn_on and turn_off on transaction-time property instances, allow to switch between these two states.

Evolution of the temporal domain of valid-time property instances Unlike transaction-time properties, the temporal domain of a valid-time property instance does not evolve automatically with the system clock. Instead, an update operator set_temporal_domain is provided, which destructively replaces the temporal domain of the property instance by the TSequence given as parameter. Since the temporal domain must always contain the effective domain, this operator may force some modifications on the effective history, i.e. if the constraint is violated after some update to the temporal domain, the effective history is restricted to fit inside the temporal domain.

Evolution of the effective history of transaction-time property instances In the case of transaction-time properties, the effective history of a temporal property instance may only be modified by an overloaded version of ODMG's set_value operator. More precisely set_value(v) applied to a transaction-time property instance TTPI, replaces the effective history of TTPI by a new history, identical to the old one except that it maps the current instant to value v. If necessary, the growth status of the property instance is turned on.

Evolution of the effective history of valid-time property instances The primitive operator for updating the effective history of a temporal property instance is set_effective_history which destructively replaces the effective history of the temporal property by the one given as parameter. In order to achieve temporal transitioning support, the standard set_value operator is also supported, with the following semantics (VTPI is a valid-time property instance):

VTPI.set_value(v) $\equiv$ VTPI.set_effective_history(
  VTPI.get_effective_history() $\cup_+$ IHistory(bag(tuple(tvalue:current_instant(),svalue:v))))

Operator get_effective_history retrieves the effective history of a property instance, and current_instant is a function yielding the instant associated to the system clock.

Accessing temporal property instances The primitive access operators for both valid-time and transaction-time property instances are get_effective_history and get_history. The former simply retrieves the effective history of the property instance, while the latter builds a history from the temporal domain and the effective history using the corresponding semantic assumption as depicted in figures 4 and 5.

To achieve temporal transitioning support, the ODMG get_value access operator is also supported on temporal property instances. Its semantics in this context is defined as follows (TPI is a temporal property instance):

TPI.get_value() $\equiv$ SValue(TPI.get_history(), current_instant())

Figure 6 describes the interfaces of temporal property instances.

3.2.4 Temporal objects' observation domain evolution

Evolution of transaction-time observation domains The observation domain of transaction-time objects evolves automatically with the system clock in a similar way as the temporal domains of transaction-time property instances. More precisely, each transaction-time object has a growth status, which may be On

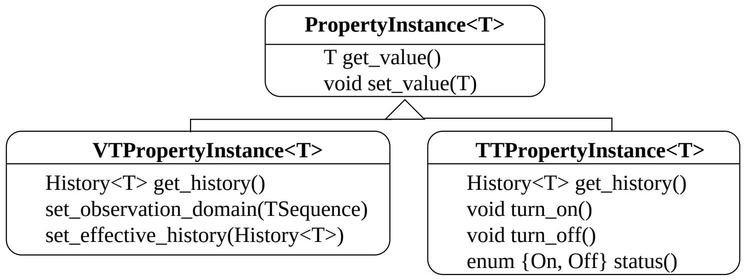


Figure 6: interfaces for temporal property instances

or Off. Conceptually, while a transaction-time object is on, the current instant is added to its observation domain at each system clock tick.

When a transaction-time object is created, its growth status is on. If the operator delete is called on it, the growth status turns off, but the object remains in the extent of its class. Subsequently, the growth status can be turned on again by calling the operator revive. There is no operator for suppressing a transaction-time object from its class extent. This is in line with the append-only semantics of transaction time.

Evolution of valid-time observation domains When a valid-time object is created, its observation domain is set to be the interval [current_instant()..], that is the interval starting at the current instant, and extending up to the largest instant recognized by the system. This observation domain can be subsequently modified using operator set_odomain. This operator destructively sets the observation domain of the object to be the temporal sequence given as parameter.

The operator delete, when applied on valid-time objects, does not physically deletes the object from its extent. Instead, it sets the upper bound of the object's observation domain, to be the current instant. That is:

$$\text{O.delete()} \equiv \text{O.set\_odomain(O.get\_odomain()} \cap \text{[..current\_instant()])}$$

Where get_odomain is an operator which returns the observation domain of an object, whether valid-time or transation-time.

The above definition of the operator delete on valid-time objects is crucial for ensuring temporal transitioning support as discussed in section 3.3.

To enable the physical deletion of a valid-time object, an operator named destroy is provided. This operator simply supresses the valid-time object from its class extent, as does the operator delete when applied on non-temporal objects.

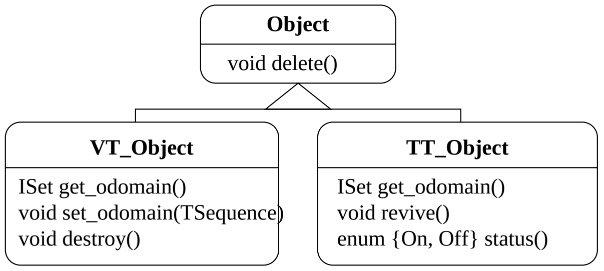


Figure 7: Interfaces for temporal classes

Figure 7 presents the proposed temporal extensions to ODMG's predefined Object interface. Appendix A, provides two detailed examples of update scenarios, which illustrate the use of the operators appearing in these interfaces.

3.3 Achieving temporal transitioning

After a database schema is modified to add temporal support to some of its classes and/or properties, applications may either continue to access the database as if there was no temporal component in the schema, or else, take into account the schema modification.

The following two paragraphs describe how objects are adapted after this kind of schema modification (i.e. how the schema change is reflected in the instances), and how the temporal and non-temporal "views" of the database may be accessed by applications.

3.3.1 Database instance adaptation

To specify how a database instance is adapted to a schema modification which adds temporal support to non-temporal schema components, an object conversion operator is defined. Conceptually, this conversion operator should be applied to all existing objects in the database instance whenever the schema is modified.

The algorithm below implements the object conversion.

Algorithm 1: *Object conversion operator*

```
/* inputs and local variables */
modified_classes: set of classes; /* classes to which temporal support is added */
modified_properties: set of properties; /* properties to which temporal support is added */
o: object; /* object to be converted */
copy_of_o: object; /* used to store a copy of o */
/* procedure */
copy_of_o := o.copy(); /* makes a copy of o */
if classOf(o) in modified_classes then
  if (temporalDimension(o) = transaction-time) then o.turn_on();
  else o.set_odomain([current_instant()..])
for p in (properties(classOf(o))
  if (p in modified_properties) then {
    if (temporalDimension(p) = transaction-time) then o.p.turn_on();
    else if (semanticAssumption(p) = stepwise) then
      o.p.set_observation_domain([current_instant()..])
    o.p.set_value(copy_of_o.p.get_value())
  }
  else o.p = copy_of_p.p /* property remains fleeting */
```

As shown by this algorithm, temporal migration support is only ensured for transaction-time and valid-time stepwise properties. This is because the idea behind temporal migration support is that when the "current" value of a temporal property is modified, the new current value assigned to it should remain constant. Such a characteristic is proper to stepwise properties.

3.3.2 Access modes

Object-oriented programming and querying languages generally only provide one construct for accessing property values, and one for updating them. For instance, in C++, the only way of accessing the value of an attribute is through the "dot" operator, whereas updating is performed through constructs of the form o.p = v. TEMPOS, on the other hand, provides several update and access primitives for each type of temporal property instance. The notion of *access mode* that we introduce in this section, establishes which updating or accessing operator on temporal properties is to be used depending on the application context. Two access modes are provided:

- The upward compatible mode: temporal property instances are snapshot-valued: their value is defined by the structural value of their history at the current instant. In addition, any reference to the extent name of a temporal class (whether valid-time or transaction-time) retrieves the observed extent at the current instant.

- The temporal mode: temporal property instances are history-valued, and no filtering is performed when accessing a temporal class extent (i.e. all objects in the extent are retrieved).

Concretely, in the upward compatible mode, whenever a temporal property is accessed either from a program or from a query, the value associated to this property is retrieved through the get_value operator (see 3.2.3). Similarly, updates in this mode are handled by the set_value operator. In the temporal mode, get_history and set_effective_history are used instead.

In addition, if the operator delete is applied over a temporal object, this object is still visible by the temporal applications, since it is still present in the extent of its class. However, it is not visible by the applications which are in upward compatible mode, since its observation domain does not contain the current instant, and consequently, it does not appear in the observed extent at the current instant (see section 3.2.4).

Different access modes may be attached to any two applications accessing the same database. As a result, the access mode should be implemented on a particular system as a parameter of each application session. The upward-compatible mode is the default in TEMPOS. This design choice is crucial to ensure temporal migration support.

4 Querying and browsing

4.1 Application example

We consider an application example dealing with a factory's assembly lines and employees. Each assembly line is identified by a number and is associated to the history of its daily production (quantity and quality). An assembly line is under the responsibility of an employee called its supervisor. For each employee, the application keeps track of its salary, and of the assembly line to which (s)he is assigned.

The schema of the corresponding temporal database is given below. The schema definition language that we use is TEMPODL, an extension of ODMG's ODL integrating the concepts of the TEMPOS model [18]. Temporal classes and properties are those preceded by the keyword valid.

```
valid granularity Day class AssemblyLine (extent TheLines, key lineNumber) {
  attribute string lineNumber;
  valid stepwise Day relationship Supervisor supervisor inverse Supervisor::supervises;
  valid stepwise Day relationship set<Workers> workers inverse Worker::worksIn;
  valid stepwise Day attribute short production;
  valid stepwise Day attribute float quality;
}
valid granularity Day class Employee (extent TheEmployees, key name) {
  attribute string name;
  valid stepwise Day attribute float wage;
}
valid granularity Day class Worker extends Employee (extent TheWorkers) {
  valid stepwise Day relationship AssemblyLine worksIn inverse AssemblyLine::workers;
}
valid granularity Day class Supervisor extends Employee (extent The Supervisors) {
  valid stepwise Day relationship AssemblyLine supervises inverse AssemblyLine::supervisor;
}
```

4.2 Querying temporal objects

TEMPOQL adds some types to OQL such as time unit, instant, interval and history, together with some language constructs for manipulating them. In the sequel, we introduce some of the salient constructs on histories provided by TEMPOQL, and illustrate them though examples taken from the application presented above. Throughout this section, we assume that the application that issues these queries is in the temporal mode.

4.2.1 Formalization of TEMPOQL

TEMPOQL's constructs are formalized using a notation similar to that of [39], which provides a complete formalization of OQL. The formalization of a construct is made up of four parts:

- A *context* which describes constraints on the types appearing in the typing part, as well as some constraints on sub-queries (such as a variable being free in a sub-query). Some of the typing preconditions will make reference to subtypes of History and Instant, although none of such subtypes are actually defined by the model. This is to achieve some extensibility, by allowing TEMPOQL constructs to be applicable to user-defined subtypes of History and Instant.
- A *syntax* given in a BNF-like notation with terminal symbols typeset in boldface.
- The *typing* rules for the construct using the notation $\frac{\text{premise}}{\text{implication}}$. The notation q::t means that the query q has type t, while q[x::t']::t means that query q has type t assuming that variable x has type t'.
- The *semantics* described in terms of expressions involving operators of the TEMPOS historical model. The semantics of a query is parametrized by a valuation function which determines the values of free symbols in the query. The notation $\nu[\mathrm{x} \leftarrow \mathrm{v}]$ denotes the valuation equal to ν except that it assigns value v to symbol x. The preconditions that apply to the operators defining the semantics of a construct (in particular those related to the observation units of the argument histories), also apply to the construct itself.

As an example, the formalization of the restriction operators on histories is given in figure 8. For the formalization of the other TEMPOQL's constructs used throughout this section, the reader may refer to appendix C.

4.2.2 Range and domain restrictions

The during construct builds a history, by restricting a given history to those instants lying in a given temporal sequence, as illustrated in the following query.

Q.1: **Domain restriction**
For each assembly line, give its number and the history of its production during 1997.

```
/* type: bag<struct<L: string, P: History<short>>> */
select struct (L: li.lineNumber, P: li.production during @“1997”)
from TheLines as li
  /* The instant @“1997” is automatically expanded into an interval at the granularity of the day i.e.
  [@“1/1/1997”...@“31/12/1997”]. */
```

As shown in figure 8(a), the semantics of the during construct is defined in terms of the Γ_{in} operator on histories.

The when construct on the other hand, builds a history by restricting a given history to those instants where its value satisfies a given condition. The semantics of this construct is defined in terms of the Γ_{if} operator on histories as detailed in figure 8(b). The following query illustrates its use.

Q.2: **Range restriction**
For each assembly line, retrieve its number and the set of instants when its production is greater than 100.

```
/* type: bag<struct<L: string, P: TSequence>> */
select struct(L: li.lineNumber, P: tdomain(li.production as p when p > 100)) from TheLines as li
```

The tdomain operator retrieves the set of instants at which the history given as parameter is defined. In the above example, this is the set of instants during which the history of the line's production fulfills the

Context: $\tau_1<\tau_1'>$ is a subtype of $History<\tau_1'>$;
τ_2 is a subtype of TSequence or a subtype of Instant
Syntax: <query> ::= <query> **during** <query>
Typing: $\dfrac{q_1 :: \tau_1<\tau_1'>,\ q_2 :: \tau_2}{q_1 \text{ during } q_2 :: History<\tau_1'>}$
Semantics: $[\![\text{q}_1 \text{ during } \text{q}_2]\!]_\nu = \begin{cases} [\![\text{q}_1]\!]_\nu\ \Gamma_{in}\ [\![\text{q}_2]\!]_\nu & \text{if } \tau_2 \text{ subtype of TSequence} \\ [\![\text{q}_1]\!]_\nu\ \Gamma_{in} & \text{if } \tau_2 \text{ subtype of Instant and} \\ \text{expand}([\![\text{q}_2]\!]_\nu, \text{Unit}([\![\text{q}_1]\!]_\nu)) & \text{Unit}([\![\text{q}_1]\!]_\nu) \prec \text{Unit}([\![\text{q}_2]\!]_\nu) \end{cases}$

(a) *during: history restriction according to temporal values*

Context: $\tau<\tau'>$ is a subtype of $History<\tau'>$; variable x is free in q_2
Syntax: <query>:= <query> **as** <identifier> **when** <query>
Typing: $\dfrac{q_1 : \tau<\tau'>,\ q_2[x : \tau'] : boolean}{q_1 \text{ as x when } q_2 : History<\tau'>}$
Semantics: $[\![\text{q}_1 \text{ as x when } \text{q}_2]\!]_\nu = [\![\text{q}_1]\!]_\nu\ \Gamma_{if}\ \lambda\text{v} \bullet [\![\text{q}_2]\!]_{\nu[\text{x}\leftarrow\text{v}]}$

(b) *when: history restriction according to structural value*

Figure 8: Semantics of TEMPOQL's restriction constructs

condition given in the when clause. The when construct can be combined with a generalized projection construct on histories called map, in a similar way that the where is coupled with the select in plain OQL. The semantics of the map/when construct on histories is given in appendix C, figure 14.

4.2.3 Temporal joins

The constructs join and ojoin correspond to the inner and outer temporal joins on histories. Their syntax is similar to that of the struct construct in plain OQL. Figure 15 in appendix C, gives the formal definition of the join construct (ojoin is defined in a similar way, and its semantics is defined in terms of the $*_\cup$operator on histories).

Since both of these constructs build pairs of synchronous values taken by two histories, they allow one to express "simultaneity". For instance, in the following query, the join construct is composed with the when one, so as to express that, at some instant, the production of a assembly line is greater than that of another assembly line.

Q.3: **Restriction on structural values and temporal join**
When was the production of assembly line L1 greater than the production of assembly line L2?

```
/* type: TSequence */
flatten (select tdomain ( join (p1: L1.production, p2: L2.production) as j when (j.p1 > j.p2)
        from TheLines as L1, TheLines as L2
        where L1.lineNumber="L1" and L2.lineNumber="L2")
```

In the above example query, ojoin could be used to take into account that when L2 has no production, the production of L1 may be considered as being greater than that of L2 :

```
/* type: TSequence */
flatten (select tdomain (ojoin (p1: L1.production, p2: L2.production)
                                as oj when oj.p1 != nil and (oj.p2 = nil or (oj.p1 > oj.p2)))
          from TheLines as L1, TheLines as L2
          where L1.lineNumber ="L1" and L2.lineNumber = "L2")
```

4.2.4 Pointwise generalization of OQL constructs

For each OQL construct, TEMPOQL provides a counterpart construct on histories which seamlessly generalizes it. The semantics of the "extended" operator is defined using the following *pointwise generalization principle*[6]: given an N-ary operator θ: $\tau_1, \ldots \tau_n \rightarrow \tau_{n+1}$, an operator θ: History$<\tau_1>, \ldots$ History$<\tau_n> \rightarrow$ History$<\tau_{n+1}>$ is defined such that: $\forall$ i $\in$ TDomain ($h_1 \cap \ldots \cap h_n$)   SValue ($h_1\ \theta \ldots \theta\ h_n$, i) = SValue ($h_1$, i) $\theta \ldots \theta$ SValue (h_n, i)

For instance, given two queries h1 and h2 both retrieving histories of integers, and an arithmetic operator θ, query h1 θ h2 retrieves the history obtained by applying operator θ to synchronous values of h1 and h2 (see annexe C, figure 16). The same holds for comparison and boolean operators.

Using the generalization principle, it is possible to express query **Q.3** in a more concise way:

Q.3: Pointwise generalization of arithmetic operators

```
element (select tdomain ((L1.production > L2.production) as b when b)
            from TheLines as L1, TheLines as L2 where L1.lineNumber = "T1"and L2.lineNumber = "T2")
```

Notice that in this query, the expression L1.production > L2.production retrieves a history of booleans. This history is then restricted to those instants when its value is "true".

OQL's "dot" operator on structured types and objects, is similarly generalized to deal with histories. More precisely, let h be a query yielding a history whose structural values are objects with some attribute a, then query h.a yields a history with the same temporal domain as h, obtained by projecting each structural value of h over attribute a. More precisely, if q is a query retrieving a history of objects of class C, and P is a temporal property over class C, then:

$$[\![\text{q.P}]\!]_\nu = \{ \langle \text{i, o'} \rangle \mid \langle \text{i, o} \rangle \in [\![\text{q}]\!]_\nu \wedge \langle \text{i, o'} \rangle \in \text{o.P} \}$$

The use of this "temporal navigation" operator is illustrated below.

Q.4: **Pointwise generalization of the navigation operator**

When did the assembly line supervised by employee X had a quality-weighted production greater than that of the assembly line supervised by Y?

```
/* type: TSequence */
element(select domain(supX.supervises.production * supX.supervises.quality
                      >
                      supY.supervises.production * supY.supervises.quality
                      as b when b)
          from TheSupervisors as supX, TheSupervisors as supY
          where supX.name = "X" and supY.name = "Y")
```

The generalization principle applies to collection types as well, i.e. OQL collection expressions (forall, exists, sum, avg, etc.) are generalized in TempOQL to apply to histories of collections. For instance, let h be a query retrieving the history of courses followed by a student, then expression sum (h) retrieves, for each instant in the temporal domain of h, the amount of courses followed by the student at that time.

[6]This principle has been used in dataflow programming languages (e.g. LUSTRE [10]), and in some temporal relational query languages [45, 31].

4.2.5 Aggregations and grouping

All OQL's aggregation functions on collections are extended to deal with subtypes of History. The only exception is count which is actually renamed to duration when applied to a history, i.e. duration(h) yields the cardinality of the temporal domain of history h. The semantics of these extensions are obvious from the definition of the corresponding operators on histories and we therefore omit them.

The map ... on ... when construct defined on histories is extended with a group by and a having clauses, accounting for grouping (see appendix C, figure 17). The grouping criteria may be a temporal unit or a duration. Similar remarks to those formulated for the map ... on ... when construct apply, and the when and the having clauses are optional. Keyword partition may be used in the map and having clauses to refer to the sub-histories generated by the group by clause.

Q.5: **Change of granularity**

For each assembly line, and for each month when it has a production, retrieve the number of days in those month, when its production quantity is greater than 100.

```
/* type: bag<struct<L: AssemblyLine, P: History<Duration>>> */
select struct (L: L, P: map duration (partition)
                        /* partition is a history at the granularity of the day. */
                        on L.production as p when p > 100
                        group by month)
from TheLines as L
```

Q.6: **Aggregations and duration-based grouping**

For each assembly line, and for each 10-day period during which the total production of this line is greater than 10000, retrieve its average production on that period.

```
/* type: bag<struct<L: string, avgP : History<real>>> */
select struct(L: L.lineNumber, avgP : map avg(partition) on L.production as p
                                      group by #"10 days"having sum(partition) > 10000)
from TheLines as L
/* The result of the sub-query introduced by the map clause, is a history at the granularity of the day: for
a given day, the average production of the 10-day period starting on that day is retrieved, provided that
the total production on that period is > 10000 */
```

4.2.6 Reasoning on chronicle-based representations

TEMPOQL allows to set a particular representation of histories. By iterating on the resulting chronicle, it is then possible to express temporal queries by explicitly referencing the timestamps associated to history values as in TSQL2 and TOOBIS–TOQL.

Three constructs to cast histories into chronicles are provided: ichronicle, xchronicle and dchronicle. They respectively yield an ordered list of instant-timestamped values, an ordered list of coalesced interval-timestamped values, and a set of temporal sequence-timestamped values. In appendix C, figure 18 we describe the semantics of xchronicle construct.

Q.7: **Reasoning on XChronicle-based representation**

For each assembly line, give its number and the longest time period(s) during which its production was greater than 100.

```
/* type: set<string, bag<Interval>> */
select struct (L: L.lineNumber, period: xsmax.tvalue)
from TheLines as L, xchronicle (L.production as p when p > 100) as xsmax
where duration (xsmax.tvalue) = max (select duration (xs.tvalue)
                                     from xchronicle (L.production as p when p > 100) as xs)
```

4.2.7 Reasoning about succession in time

The afterfirst beforefirst, afterlast and beforelast constructs are straightforward adaptations of the corresponding operators on histories (see appendix C, figure 19). Their syntax is similar to that of the when construct.

Q.8: **Succession in time - splitting histories**
For each assembly line give its number and its production history up to the last time its production was smaller than 100.

```
/* type: set<L: string, P: History<unsigned long>> */
select struct (L: L.lineNumber, P: L.production as b beforefirst b ≥ 100)
from TheLines as L
```

4.3 Pattern-matching queries

The construct Matches defined in appendix B, allows to formulate rather complex queries about succession in time in an elegant way.

Q.9: **Pattern-matching**
Retrieve the workers who once moved from assembly line L1 to L2 where they stayed for at most 3 months before moving to L3?

```
/* type: bag<Workers> */
select w from TheWorkers as w
where w.assemblyLine as u
        matches (u.name = “L1”followed by u.name = “L2” during > #“3 months” followed by u.name = “L3”)
```

4.4 Pointwise temporal object browsing

The pointwise browser is based on an extension of existing object browsing techniques such as PESTO’s “synchronous” navigation [9]. This extension is designed to specifically address time-related users’ tasks such as:

- Analyze data about the supervisor and workers of each assembly line at a given date.
- Compare at different dates, a given worker’s wage with respect to that of the supervisor of the assembly line to which he is assigned.
- Find out whether the composition of a given assembly line (equipment plus workers) considerably changes when its supervisor does.

4.4.1 Overview

The pointwise browser interface (see figure 9) is made up of two parts: a *time-line window* and a tree of *snapshot windows*. A snapshot window displays either a non-temporal object[7] or a snapshot of a temporal object at a given instant. The instant with respect to which the object snapshots are determined is the same for all the windows in the tree, and is subsequently called the *reference instant*. The reference instant is constrained to reside within a given interval called the *temporal browsing range*.

The role of the time-line window is to fix the reference instant. At the beginning of a session, the reference instant is at the middle of the temporal browsing range. Its position varies thereafter according to the user interactions with the sliders and buttons composing the time-line window. In its simplest form, the time-line window is composed of a slider (called the *main slider*) and four buttons placed at the ends of this

[7]An object is *temporal* if it owns at least one temporal property and *non-temporal* otherwise.

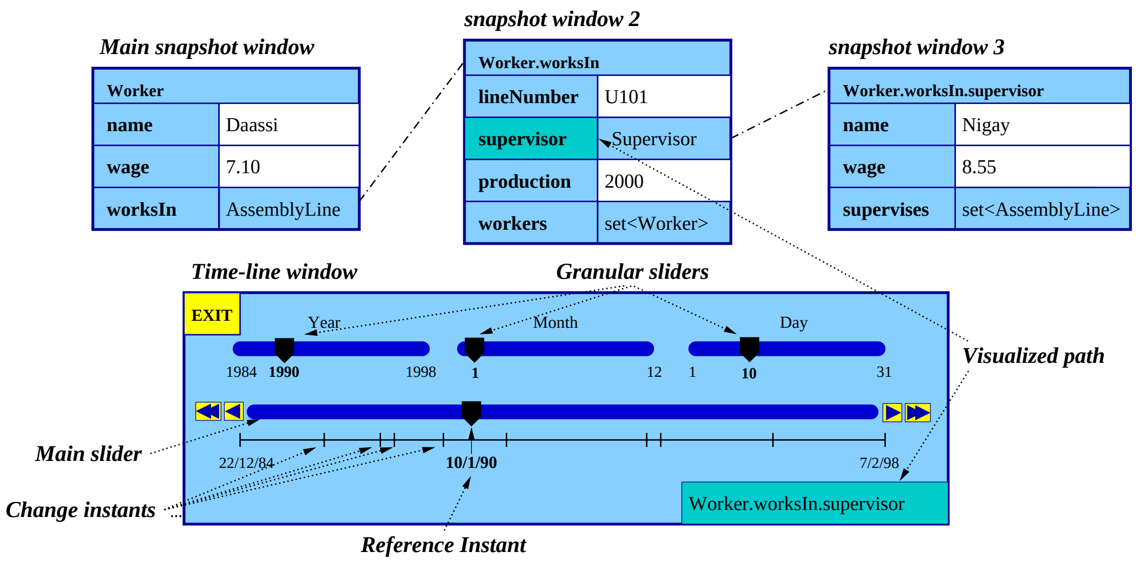


Figure 9: The pointwise temporal object browser. The schema of the underlying database is the one given in section 4.1. In particular, properties Worker::wage and Worker::worksIn are temporal.

slider. Two of the buttons (labeled by simple arrows), allow the user to move the reference instant forward or backward by one unit. The other pair of buttons (labeled with double-arrows), allow to move the reference instant to the next/previous instant where the value of a given navigation path (called the *visualized path*) changes. The instants at which the value of the visualized path changes are called *change instants*. Change instants are visually represented as vertical marks lying within a horizontal line just beneath the main slider.

In figure 9, the change instants are those when the supervisor of worker "Daassi" changes, whether this change is due to the fact that this worker is assigned to a new assembly line and that this assembly line has a different supervisor than the previous one, or to the fact that the supervisor of the assembly line to which this worker is assigned changes.

In addition to the main slider, the time-line window may additionally contain several *granular sliders*, which allow the user to move the reference instant with different "steps" according to a given calendar. For instance, if the reference instant is a date, and that the user specifies the calendar Year/Month/Day (as in figure 9), three granular sliders appear in the time-line window: the first one allows one to move the reference instant with a step of a year, the second one with a step of a month, and the third one with a step of a day (within the limits of a given month).

Snapshot windows are structured as forms containing one line per property of the visualized object or object snapshot[8]. Each line is composed of two buttons: the left one labeled with the name of the property, and the right one labeled with its value at the reference instant[9]. The value of a non-temporal property is always the same regardless of the reference instant. The value of a temporal property at a given instant is equal to the value of its history at that instant, which is itself defined as follows:

- The value at instant I, of a history represented as an instant-timestamped collection of objects, is equal to the object within this collection whose timestamp is equal to I. If no such object exists, the history's value is null.
- The value at instant I, of a history represented as an interval-timestamped collection of objects, is equal to the object within this collection whose timestamp contains instant I. If no such object exists, the history's value is null.

[8]For the time being, we restrict our examples to snapshot windows displaying single objects or object snapshots. We will discuss afterwards how collections are accommodated.

[9]If the value of a property is not printable (i.e. its type is not integer, string, etc.), the name of its class is used as its label.

All buttons within a snapshot window are clickable, except those which denote literal values (i.e. integers, reals, string, and characters). For instance, in figure 9 all the buttons within the snapshot windows are clickable, except the white-colored ones.

At the beginning of a session, there is a single snapshot window. Other snapshot windows are incrementally added to the tree according to the user interactions with the clickable buttons denoting object references which appear within existing forms. The object displayed by a given snapshot window other than the main one, is equal to the object referenced by the button from which this window was opened. For instance, the configuration shown in figure 9 is obtained by displaying the worker named "Daassi" and successively clicking on the buttons labeled AssemblyLine and Supervisor.

The user may also click on the buttons labeled with property names (i.e. the buttons on the left column of a form). The semantics of this interaction is that the selected property becomes the visualized path expression and the set of "change instants" attached to the time-line window are updated accordingly. For instance, clicking on the button labeled production on window 2 of figure 9, sets the visualized path to be Worker.worksIn.production instead of Worker.worksIn.supervisor. The vertical marks drawn on the line just below the main slider, and the label appearing in the low-right corner of the time-line window are then modified accordingly.

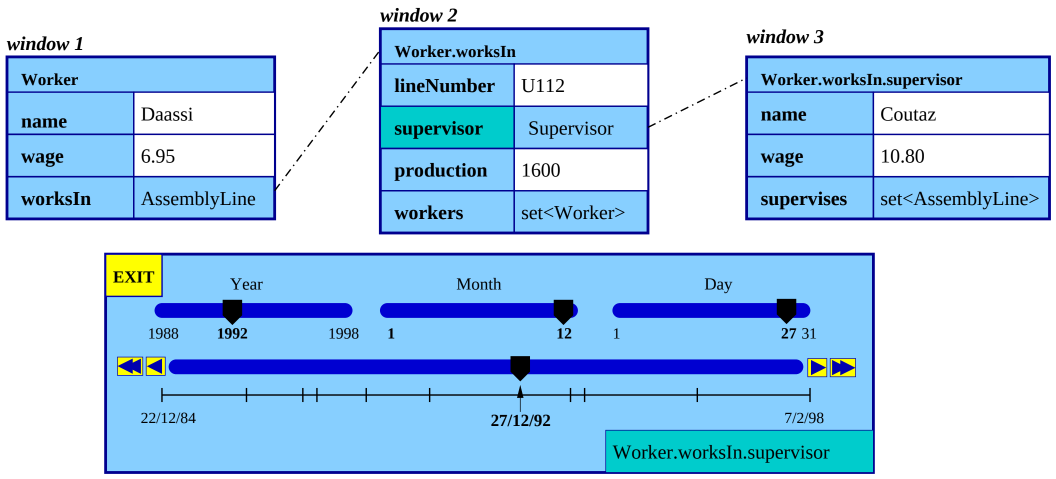


Figure 10: Modification of the reference instant upon the configuration given in figure 9.

Whenever the user modifies the reference instant, the new reference instant is notified to the main snapshot window (see Figure 10). Upon receiving this notification, the main window computes the snapshot at the new reference instant, of the object that it displays, and updates its appearance so as to reflect this new snapshot. During this process, if the value of a temporal property changes, the new value is transmitted to its dependent window if any. Finally, the main window propagates the notification of the new reference instant to all its dependent windows, and the above process is carried out recursively.

As stated before, the reference instant ranges through an interval called the temporal browsing range. This range is taken to be the smallest interval containing all the instants when at least one of the temporal properties of the object displayed by the main snapshot window (subsequently called the *main object*) is defined. For example, if the main object is a worker W, such that:

W.salary = set(struct(timestamp: [1..4], value: 10.0), struct(timestamp: [6..9], value: 12.0))
and
W.worksIn = set(struct(timestamp: [2..4], value: X), struct(timestamp: [6..8], value: Y)).

(where X and Y are two assembly lines), then the temporal browsing range is taken to be interval [1..9].

Notice that the above definition entails that the main object is temporal, since otherwise, the browsing range would be empty.

4.4.2 Pointwise browsing in the presence of null-valued properties

Heretofore, we have implicitly assumed that all properties displayed within snapshot windows have non-null values. However, null-valued properties within a snapshot window may arise in two cases:

- The value of the property at the reference instant was actually set to "null" through an update (this can occur whether the property is temporal or not).
- The history of a temporal property is not defined at the reference instant, in which case we consider that its value is null. This situation can occur in the middle of a pointwise browsing session, since the temporal browsing range may include instants in which some of the temporal properties of the main object are defined while others are not.

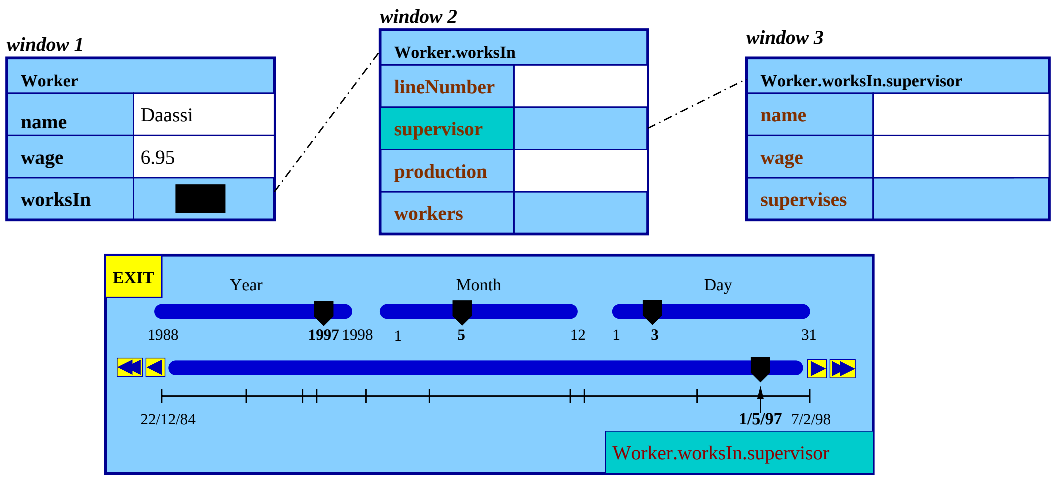


Figure 11: Modification of the reference instant upon the configuration given in figure 10. Windows 2 and 3 become inactive as a result of this interaction.

As in O_2Look [37], we visually denote a null value through a filled rectangle. However, this does not solve all the problems arising from nulls. Indeed, suppose that the worker displayed in figure 10 is not assigned to any assembly line on 1/5/97 (i.e. there is no element in its history whose timestamp contains this date). If the reference instant is set to this date, the value of property worksIn becomes null, and something has to be done with its dependent forms (i.e. windows 2 and 3 in figure 10).

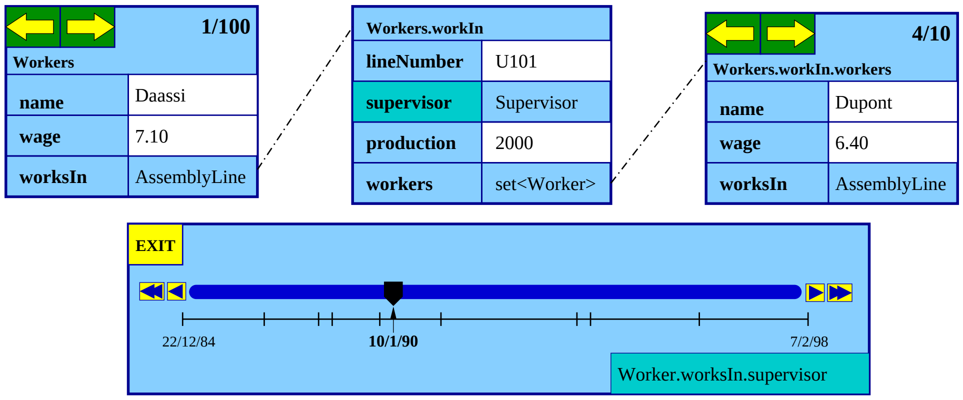
In our approach, if further a modification of the reference instant, one of the properties displayed by a snapshot window becomes null, and if this property has a snapshot window attached to it, then all the windows in the sub-tree stemming from this property become *inactive*. Inactivity of a snapshot window can be visually rendered in at least two ways:

- Hide the window (and redisplay it when it becomes active again).
- Modify the appearance of some elements within the window, e.g. by graying out the labels denoting property names and erasing the labels denoting property values. In this case, labels should be restored as the window becomes active again. Figure 11 illustrates this approach.

We believe that the second approach is to be preferred, since hiding and redisplaying windows violates the screen stability ergonomic property.

4.4.3 Pointwisely browsing collections of temporal objects

To accommodate collections, we augment the pointwise browser with the concept of *synchronous navigation* as defined in object browsers such as PESTO [9]. Basically, a collection of temporal objects is displayed in the same way as a single one, except that the corresponding snapshot window contains a couple of arrow-labeled buttons on top of it. This window displays the snapshot at the reference instant, of one of the objects within the collection. Clicking on either of the arrows allows one to switch to the next or the previous object in the collection (see figures 12(a) and 12(b)).



1/100
Workers

name	Daassi
wage	7.10
worksIn	AssemblyLine

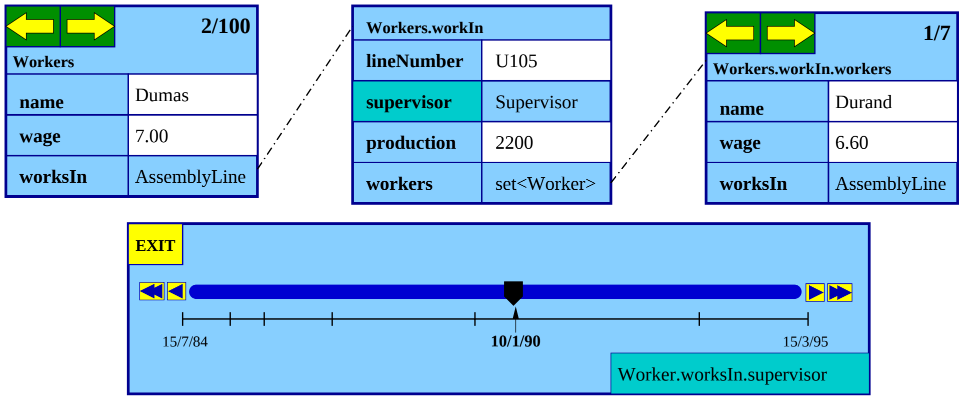
Workers.workIn

lineNumber	U101
supervisor	Supervisor
production	2000
workers	set<Worker>

4/10
Workers.workIn.workers

name	Dupont
wage	6.40
worksIn	AssemblyLine

(a) Configuration 1



2/100
Workers

name	Dumas
wage	7.00
worksIn	AssemblyLine

Workers.workIn

lineNumber	U105
supervisor	Supervisor
production	2200
workers	set<Worker>

1/7
Workers.workIn.workers

name	Durand
wage	6.60
worksIn	AssemblyLine

(b) Configuration 2

Figure 12: Pointwisely browsing a collection of temporal objects. Configuration 2 is the result of clicking on the right arrow of the main snapshot window in configuration 1.

As before, the temporal browsing range is defined with respect to the object visualized by the main snapshot window. Therefore, when this object changes, the browsing range is recomputed. This is the reason why the time-line is redrawn when transitioning from configuration 1 (see figure 12(a)) to configuration 2 (see figure 12(b)). Notice also that during this transition, the change instants are also re-computed. Under some circumstances this computation involves a relatively large amount of data. For instance, consider the example of figures 12(a) and 12(b) and suppose that the visualized path expression is Workers.worksIn.supervisor.wage (which means that the path Workers.worksIn.supervisor is displayed), computing the change instants then involves the following histories:

- The history of the employee's assembly lines.
- The history of the supervisors of each line where the visualized employee has ever worked.

- The history of the wages of each supervisor appearing within any of the histories referenced in the previous item.

In order to ensure an acceptable response time when transitioning from one object to another, a good history join algorithm should be used, and the involved histories must be well clustered on disk. Studying these two issues is therefore an interesting perspective to the work reported here.

4.5 Comparison with related works

Data visualization has received little attention within the temporal database research domain[10]. A notable exception to this remark is [35], which specifies a 3D interface for browsing temporal relational databases. In this approach, a temporal relation is represented as a sequence of time-indexed planes, each one displaying a table denoting a snapshot of the relation. Although the interface is not detailedly described, it seems that the interaction devices are limited to two scroll-bars: one for browsing through the records of a relation snapshot, and the other for navigating across the time dimension.

In the area of information visualization, many techniques for graphically displaying and browsing temporal data have been designed [4, 52]. Most of these techniques are oriented towards quantitative time series, i.e. periodical series of numerical data items. [36] adapts some concepts developed in these works to design an interface for visualizing legal and medical personal records involving non-quantitative temporal data such as histories of texts and of complex objects.

The pointwise temporal object browser is an extension of data browsers such as KIVIEW [34], O_2Look [37], ODEVIEW [16], SUPER [17], and PESTO [9]. In particular, the technique used to navigate through collections is based on the concept of synchronous navigation proposed in KIVIEW, and refined in ODEVIEW and PESTO. Nevertheless, the pointwise browser considerably differs from all the above ones, since it treats time as a dimension per se. Indeed, the pointwise browser allows one to orthogonally navigate through the following three dimensions :

- Through the objects composing a collection using the arrow-labeled buttons on top of each snapshot window denoting a collection.
- Through object relationships using the clickable buttons within the right columns of a snapshot window, whether this snapshot window denotes a single object or a collection of objects.
- Through the time dimension using the components of the time-line window.

Moreover, the pointwise browser takes into account the impacts of null-valued properties during synchronous navigation, whereas this issue is completely neglected in the above proposals.

5 Implementation

TEMPOS has been implemented as a prototype on top of the object-oriented DBMS O_2. This prototype has been used to develop several concrete applications. In particular, we have implemented an application dealing with the management of annotated time series for economical analysis, and an application concerning the analysis of the behavior of people over time in a ski resort [26]. Furthermore, in another work we have adapted this implementation to the management of video annotations [23].

[10]On the contrary, several visual query languages for temporal databases have been proposed (e.g. [28]). We do not discuss them since the functionalities addressed by these proposals are beyond the scope of the present paper.

5.1 Overall architecture

Figure 13 depicts the prototype architecture. It essentially consists of a library of classes corresponding to the ADT hierarchies defined in the the time and historical models, two preprocessors implementing respectively TEMPODL and TEMPOQL, and a visualization module. A temporal metadata manager accounts for the communication between the two preprocessors.

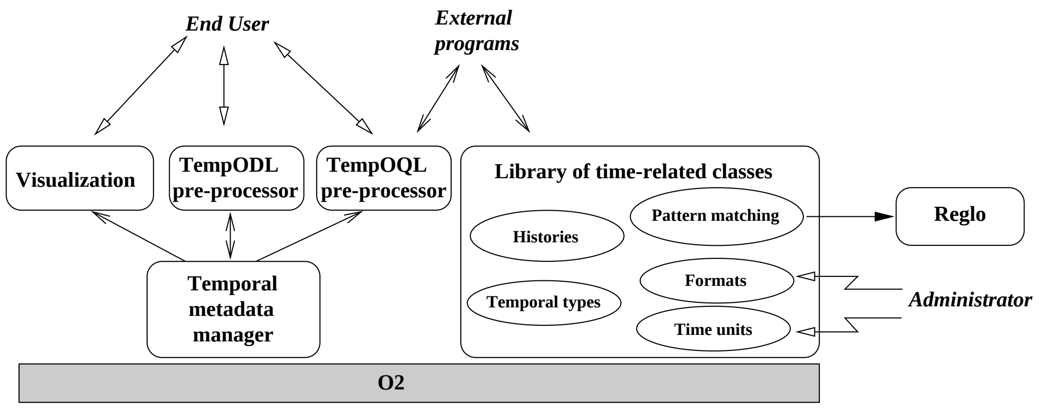


Figure 13: Prototype architecture

The library of time-related classes has been primarily implemented in the O_2's database programming language O_2C (which is translated into C by a preprocessor provided by the O_2 system), while the preprocessors and the metadata manager have been implemented in C using code generation tools such as Lex and Yacc. These classes can be used from C++ and Java programs using the corresponding bindings provided by the O_2 system. One of the modules of the library of time-related classes implements the pattern-matching operator described in section 3.1.4. We discuss the implementation of this module in section 5.3.

The visualization module implements the pointwise temporal object browser, and some other visualization techniques that we are developing as part of an ongoing work on temporal visual data analysis. The pointwise browser is implemented in C using the XForms toolkit[11], and the other techniques are being implemented in Java using JavaSwing. As the preprocessor, this module uses metadata related to the temporal classes and properties defined in the database's schema.

5.2 Implementation issues related to the lack of parametric classes in ODMG

Perhaps, the major problems that we faced during the design and implementation of the TEMPOS prototype, were those related to the lack of parametric classes in the O_2 model (which is true of the ODMG object model as well). Indeed, the History datatype could be naturally mapped into a parametric class.

One of the solutions that we envisaged, is to generate a class for each kind of history involved in an application (e.g. one for histories of integers, another for histories of floats, etc.). However, in realistic situations, this rapidly leads to a high proliferation of classes. In addition, some operators, such as the temporal joins, cannot be satisfactorily implemented using this approach, since the structural value type of the resulting history intrinsically depends on that of the argument histories (see section 3.1.3).

Instead, we decided to partially simulate parametric classes by exploiting the preprocessors included in the architecture. In this approach, a single non-parametric class History, corresponding to histories whose structural values are of type Object (the top of the ODMG's class hierarchy), is first implemented. Then, during schema definition, each history-valued attribute is declared as being of type History by the TEMPODL preprocessor, but its exact type specification is stored in the temporal metadata manager. Since this

[11]http://world.std.com/˜xforms

metadata manager is accessed by the TEMPOQL preprocessor, this latter knows the exact type of the histories involved in a query. With this knowledge, the TEMPOQL preprocessor adds explicit downcastings in the translated query expression, whenever the structural value of a history is involved. In this way, the user of TEMPOQL manipulates histories as if they were parametrically typed.

The above solution has several drawbacks. First, adding explicit downcastings in the translated queries introduces a burden during query evaluation, since the OQL interpreter performs a dynamic type checking whenever an object is downcasted. Second and foremost, the above solution does not take into account that the database objects (and in particular the histories contained in a temporal database) are not only accessible through the query language, but also, through any of the programming language bindings. As a result, in the current TEMPOS implementation, the typing of histories has to be coded into the application programs (through downcastings and explicit dynamic type checkings).

The above considerations illustrate the necessity of extending the current ODMG object model to support user-defined parametric classes, as discussed in [1].

5.3 Implementation of the pattern-matching operator

In the next paragraphs, we describe the algorithm that we have implemented for evaluating the boolean pattern-matching operator Matches (see section 3.1.4). Basically, this algorithm proceeds in three steps.

First, the involved pattern is mapped into a regular expression over boolean variables. During, this translation, a correspondence table between these boolean variables and the atomic formulae appearing in the pattern is also built.

Second, the resulting regular expression is translated into an automaton whose inputs are boolean streams. For performing this translation, we chose the technique developed in [38] and materialized in a tool called Reglo. This technique was chosen over classical automata-generation techniques for three main reasons:

- This technique assumes that the alphabet of the regular expression is composed of boolean variables and that the input of the generated automata is a vector of boolean streams (which exactly match our needs), whereas classical techniques assume that the alphabet of the regular expression is a set of tokens, and that the input of the generated automaton is a single stream of tokens.
- The size of the generated (non-deterministic) automaton is linear on the size of the involved regular expression[12], and the time complexity of the generation algorithm is also linear. Thereby, the exponential space and time complexity of classical deterministic automata generation is avoided.
- The technique has been extended to efficiently deal with exponentiation operators (i.e. counters). This feature is fundamental for implementing the duration-constrained repetition operator of the pattern-matching language (see section 3.1.4).

Finally, once the automata built up, it is executed by successively providing it as inputs, the boolean values taken at each instant in the involved history, by the atomic formulae appearing in the pattern. Obviously, to evaluate these atomic formulae, the structural values of the history need to be accessed. This process terminates either when the automaton outputs true, or when the history has been completely scanned. Notice that it is not necessary to evaluate the atomic formulae twice for two successive instants, unless the value of the history changes in between. This remark leads to a straightforward yet important optimization, specially when the involved history varies stepwisely (e.g. the price of a product).

Taking into account the above optimization, the evaluation of the third step of the algorithm for a temporal pattern comprising n distinct atomic formulae, and whose structural value changes s times, involves at most $n \times s$ atomic formulae evaluations. Since the complexities of the first two steps are linear on the size of the pattern, this is also the worst-case complexity of the overall algorithm.

[12]This is not always true when negations or conjunctions are involved in the pattern

6 Conclusion

TEMPOS is a comprehensive temporal database framework which synthesizes and unifies most of the concepts, requirements and functionalities, recognized as necessary to temporally extend existing DBMS. This framework is composed of a time model, a history model, a temporal object model, a query language, and a visual browser.

The time model defines a set of abstract datatypes modeling time values expressed with respect to an extensible set of time units. These types are provided with a rich collection of arithmetic and comparison operators.

The history model provides an abstract datatype dedicated to the notion of history, and a wide variety of representation independent operators over it. These operators, together with those defined on the types of the time model, form the basis for TEMPOQL, the proposed extension of ODMG's OQL.

The temporal object model, which extends ODMG's object model, fulfills two important requirements related to legacy code migration: upward compatibility and temporal transitioning support. The former states that a database may be transparently migrated from an ODMG system to a temporal extension of it. The latter allows non-temporal legacy code to remain usable even after a database schema is modified to add temporal support to some of its components. Temporal transitioning support is ensured by clearly separating temporal properties from the history of their values: a temporal property may have a historical value in the context of a temporal application and an "snapshot" value in the context of a non-temporal one. In addition, update operators on temporal properties are defined in such a way that updates done by non-temporal applications are compatible with those performed by temporal ones. The concept of "now", which has lead to many confusions in previous temporal data models [14], is naturally modeled by dynamically generating a history from a now-relative temporal property.

The TEMPOQL query language offers facilities to express, in a unified framework, classical temporal queries such as restriction, join and grouping, together with operators for reasoning about succession in time. With respect to related proposals, the main originalities of TEMPOQL are :

- It is representation-independent in the sense that histories are primarily manipulated through constructs whose semantics is not tight to a particular representation, whereas in [40, 51, 47], queries on histories are expressed by applying iterators on collections of interval-timestamped values representing them.
- It integrates some novel temporal query operators such as a boolean pattern-description language, and two algebraic history grouping operators. While these latter operators are present in most existing temporal query languages (e.g. TSQL2 [42]), they have never been, to our knowledge, defined algebraically in this context. The algebraic nature of these operators in TEMPOQL, allows to easily compose them with the other operators of the algebra.
- By applying the pointwise generalization principle that has been used in the design of some dataflow programming languages (e.g. LUSTRE [10]), TEMPOQL extends the semantics of standard OQL constructs to deal with histories.

Regarding data visualization, TEMPOS integrates a novel technique for browsing temporal object databases. This technique orthogonally supports three kinds of navigation: (i) navigation through time, (ii) navigation via object relationships, and (iii) navigation within the elements of a collection.

All the above models and languages have been formalized at the syntactical and the semantical level, and a prototype on top the O_2 DBMS has been developed on the basis of this formalization. This prototype has been used to develop several applications from various contexts (GIS, time series, multimedia).

As a future work, we envisage two main research avenues: designing user interfaces for visually mining temporal and spatio-temporal data, and modeling moving objects over constrained networks. Regarding the first issue, a project has already been started in our research team, as reported in [20].

References

[1] S. Alagic. The ODMG model: does it makes sense? In *Proc. of the Int. Conference on Object-Oriented Programming Systems, Langages and Applications (OOPSLA)*, Atlanta, GA (USA), October 1997.

[2] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), November 1983.

[3] J. Bair, M. Bohlen, C.S. Jensen, and R.T. Snodgrass. Notions of upward compatibility of temporal query languages. Technical Report TR-6, Time Center, 1997.

[4] J. Bertin. *Graphics and Graphic Information Processing*. Walter de Gruyter & Co, Berlin, 1981.

[5] E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extending the ODMG object model with time. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.

[6] M. Bohlen, R. Busatto, and C.S. Jensen. Point-based versus interval-based temporal data models. In *Proc. of the 14th Int. Conference on Data Engineering*, pages 192–200, 1998.

[7] J.-F. Canavaggio. TEMPOS, un modèle d'historiques pour un SGBD temporel. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), novembre 1997.

[8] J.-F. Canavaggio and M. Dumas. Manipulation de valeurs temporelles dans un SGBD à objets. In *actes du XV congrès INFORSID*, Toulouse, juin 1997. English version at ftp://ftp.imag.fr/pub/labo-LSR/STORM/PUBLICATIONS/1997/mtv.ps.gz.

[9] M. Carey, L. Haas, V. Maganty, and J. Williams. PESTO : an integrated query/browser for object databases. In *Proc. of the Int. Conference on Very Large Databases (VLDB)*, Mumbai, India, August 1996.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE : a declarative language for programming synchronous systems. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, Munchen, Germany, 1987.

[11] R.G.G. Cattell and D. Barry, editors. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, January 2000.

[12] T. Cheng and S. Gadia. A pattern matching language for spatio-temporal databases. In *Proc. of the 3rd International Conference on Information and Knowledge Management (CIKM)*, Gaithersburg, MD (USA), November 1994.

[13] J. Chomicki. Temporal Query Languages: A Survey. In *Proc. of the International Conference on Temporal Logic*, Bonn, DE, 1994.

[14] J. Clifford, C. Dyreson, T. Isakowitz, C. Jensen, and R. Snodgrass. On the semantics of "Now" in databases. *ACM Transactions on Database Systems*, 22(2):171 – 214, June 1997.

[15] J. Clifford and A. Rao. A simple general structure for temporal domains. In *Conference on Temporal Aspects in Information Systems*, Sophia-Antipolis, May 1987. AFCET.

[16] S. Dar, N.H. Gehani, H.V. Jagadish, and J. Srinivasan. Queries in an object-oriented graphical interface. *Journal of Visual Languages and Computing*, 6(1):27 – 52, 1995.

[17] Y. Dennebouy, M. Andersson, A. Auddino, Y. Dupont, E. Fontana, M. Gentile, and S. Spaccapietra. SUPER: visual interfaces for object + relationship data models. *Journal of Visual Languages and Computing*, 6(1):27 – 52, 1995.

[18] M. Dumas. TEMPOS: une plate-forme pour le développement d'applications temporelles au dessus de SGBD à objets. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Juin 2000.

[19] M. Dumas, C. Daassi, M.C. Fauvet, and L. Nigay. Pointwise temporal object database browsing. In *Proc. of the ECOOP Symposium on Objects and Databases*, Sophia Antipolis, France, June 2000. 15 pages.

[20] M. Dumas, C. Daassi, M.C. Fauvet, L. Nigay, and P.C. Scholl. Interactively exploring temporal object databases. In *Actes des 16e Journées Bases de Données Avancées (BDA)*, Blois, Octobre 2000.

[21] M. Dumas, M.-C. Fauvet, and P.-C. Scholl. Handling temporal grouping and pattern-matching queries in a temporal object model. In *Proc. of the CIKM International Conference*, Bethesda, MD (USA), November 1998.

[22] M. Dumas, M.-C. Fauvet, and P.-C. Scholl. Updates and application migration support in an ODMG temporal extension. In *Proc. of the 1st International Workshop on Evolution and Change in Data Management*, Paris, France, November 1999. Springer-Verlag, LNCS No 1727. 12 pages.

[23] M. Dumas, R. Lozano, M.-C. Fauvet, H. Martin, and P.-C. Scholl. A sequence-based object-oriented model for video databases. *Multimedia Tools and Applications*, 2001. To appear.

[24] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 2. Elsevier Science Publisher, 1990.

[25] O. Etzion, S. Jajodia, and S.M. Sripada, editors. *Temporal Databases: Research and Practice*. Springer Verlag, LNCS 1399, 1998.

[26] M.-C. Fauvet, S. Chardonnel, M. Dumas, P.-C. Scholl, and P. Dumolard. Analyse de données géographiques : application des bases de données temporelles. *Revue Internationale de Œomatique*, 8(1-2), novembre 1998.

[27] M.-C. Fauvet, M. Dumas, and P.-C. Scholl. A representation independent temporal extension of ODMG's Object Query Language. In *actes des Journées Bases de Données Avancées*, Bordeaux, France, Octobre 1999. 20 pages.

[28] S. Fernandes, U. Schiel, and T. Catarci. Visual query operators for temporal databases. In *Proc. of the 4th Int. Workshop on Temporal Representation and Reasoning (TIME)*, May 1997.

[29] S. K. Gadia and S. S. Nair. Temporal databases: a prelude to parametric data. In Tansel et al. [48].

[30] I. A. Goralwalla and M. T. Ozsu. Temporal extensions to a uniform behavioral object model. In *Proc. of the 12th International Conference on the Entity-Relationship Approach - ER'93, LNCS 823*. Springer Verlag, 1993.

[31] R. Guting, M. Bohlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgianis. A foundation for representing and querying moving objects. Technical Report 238, FerUniversität das Hagen (Germany), 1998.

[32] I. Kakoudakis and B. Theodoulidis. The TAU Temporal Object Model. Technical Report TR-96-4, TimeLab, University of Manchester (UMIST), 1996.

[33] N. A. Lorentzos. The Interval-extended Relational Model and its application to valid-time databases. In Tansel et al. [48].

[34] A. Motro, A. D'Atri, and L. Tarantino. KIVIEW: An object oriented browser. In *Proc. of the Int. Conference on Expert Database Systems*, Vienna, Virginia (USA), April 1988. Benjamin Cummings.

[35] P. Papapanagiotou and B. Theodoulidis. ERT/vql: A visual environment for querying and manipulating temporal database applications. Technical Report TR-94-5, Timelab, UMIST, 1994.

[36] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Schneiderman. LifeLines: Visualizing Personal Histories. In *proc. of the ACM CHI conference*, Vancouver, Canada, April 1996.

[37] D. Plateau, P. Borras, D. Leveque, J.C. Mamou, and D. Tallot. Building user interfaces with Looks. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *The story of O_2*. Morgan Kaufmann, 1992.

[38] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *Proc. of the 23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[39] H. Riedel and M.H. Scholl. A formalization of ODMG queries. In *Proc. of the 7th Int. Conference on Data Semantics (DS-7)*, Leysin, Switzerland, October 1997. Chapman & Hall.

[40] E. Rose and A. Segev. TOOSQL - a temporal object-oriented query language. In *Proc. of the 12th International Conference on the Entity-Relationship Approach - ER'93*, 1993.

[41] R. T. Snodgrass. Temporal object-oriented databases: a critical comparison. In W. Kim, editor, *Modern database systems. The object model, interoperability and beyond*, chapter 19. Addison Wesley, 1995.

[42] R. T. Snodgrass, editor. *The TSQL2 temporal query language*. Kluwer Academic Publishers, 1995.

[43] R. T. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proc. of ACM SIGMOD*, May 1985.

[44] R.T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, July 1999.

[45] R.T. Snodgrass, M. Bohlen, C. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In Etzion et al. [25].

[46] A. Sotiropoulou, M. Souillard, and C. Vassilakis. Temporal extension to ODMG. In *Proc. of the Workshop on Issues and Applications of Database Technology (IADT)*, Berlin, Germany, July 1998.

[47] A. Steiner and M.C. Norrie. Implementing temporal databases in object-oriented systems. In *Proc. of the 5th International Conference on Databases for Advanced Applications*, Melbourne, Australia, April 1997.

[48] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodggrass, editors. *Temporal Databases*. The Benjamins/Cummings Publishing Company, 1993.

[49] D. Toman. Point-based temporal extensions of SQL and their efficient implementation. In Etzion et al. [25].

[50] TOOBIS ESPRIT Project. TODM, specification and design. Deliverable T31TR.1, MATRA CAP SYSTEMES – O2 Technology, December 1996.

[51] TOOBIS ESPRIT Project. TOQL specification. Deliverable T33TR.1, University of Athens, National Technical University of Athens, 01-PLIFORIKI S.A., O2 Technology, December 1996.

[52] E. Tufte. *The visual display of quantitative information*. Graphics Press, 1984.

[53] X. S. Wang, S. Jajodia, and V. S. Subrahmanian. Temporal modules : an approach toward federated temporal databases. *Information Systems*, 82, 1995.

[54] G.T.J. Wuu and U. Dayal. A uniform model for temporal and versioned object-oriented databases. In Tansel et al. [48].

A Updating temporal properties

Consider a simple application managing information about products produced and sold by a company. To model it, a class Product with a valid-time attribute price is introduced. This property is observed at the granularity of the day, has structural type real, and evolves stepwisely. The valid-time observation of an object of class Product models the time when the product is produced (at the granularity of the day), while the temporal domain of an instance of property price models the time when its price is defined. The following table illustrates a possible update scenario. The notation [i..] (resp. [..i]) designates the interval containing all instants having the same granularity as i and greater than (resp. less than) or equal to it.

Event	A new assembly line is created on 1/4/98; it is supervised by S1.
Operation	L = new AssemblyLine;
	L.supervisor.set_odomain([1/4/98..]);
	L.supervisor.set_effective_history({⟨1/4/98, S1⟩})
Result	L.supervisor.get_history() = {⟨[1/4/98..], S1⟩}
	∀ i ∈ [1/4/98..] (S1.supervises.get_history().get_value(i) = L)
Event	From 1/6/98, S2 is the supervisor of L
Operation	L.supervisor.set_effective_history(L.supervisor.get_effective_history()
	$\cup_+$ {⟨1/6/98, S2⟩})
Result	L.supervisorr.get_history() = {⟨[1/4/98..31/5/98], S1⟩, ⟨[1/6/98..], S2⟩}
	∀ i ∈ [1/6/98..] (S2.supervises.get_history().get_value(i) = L)
	∀ i ∈ [1/6/98..] (S1.supervises.get_history().get_value(i) = NULL)
Event	From 6/8/98, the assembly line is suspended.
Operation	L.set_odomain(L.get_odomain() ∩ [..5/8/98])
Result	L.get_odomain() = [1/4/98..5/8/98]
	L.supervisor.get_history() = {⟨[1/4/98..31/5/98], S1⟩,
	⟨[1/6/98..5/8/98], S2⟩}
	∀ i ∈ [6/8/98..] (S2.supervises.get_history().get_value(i) = NULL)
Event	From 1/1/99 the assembly line is reintroduced, but it is not in motion.
Operation	L.set_odomain(L.get_odomain() ∪ [1/1/99..])
Result	L.get_odomain() = {[1/4/98..5/8/98], [1/1/99..]}
	L.supervisor.get_history() unchanged
Event	From 1/2/99 to 31/3/99 the assembly is in motion, supervised by S3.
Operation	L.supervisor.set_odomain (L.supervisor.get_odomain() ∪ [1/2/99..31/3/99])
	L.supervisor.set_effective_history(L.supervisor.get_history() $\cup_+${⟨1/2/99, S3⟩})
Result	L.get_odomain() unchanged
	L.supervisor.get_history() = {⟨[1/4/98..31/5/98], S1⟩,
	⟨[1/6/98..5/8/98], S2⟩, ⟨[1/2/99..31/3/99], S3⟩}
	∀ i ∈ [1/2/99..31/3/99] (S3.supervises.get_history().get_value(i) = L)

Consider another simplified application dealing with the observation of the courses followed by students in a School. We choose to model these data by means of a transaction-time class Student with a transaction-time property follows whose structural type is set<Course> (the class Course is not described here). Both the observation domain of objects of class Student and the temporal domain of instances of property follows, are observed at the granularity of the day. The following table describes a possible update scenario that may occur in the context of this application and how it is handled using the operators described bellow.

Date **Event** **Operation** **Result**	1/9/98 A new student register at the school S = new Etudiant S.status() = On ; S.follows.status() = On S.follows.get_history() = $\{\langle$[1/9/98..1/9/98], $\{\ \}\rangle\}$
Date **Event** **Operation** **Result**	3/9/98 The student enrolls in Math and Chemistry courses S.follows.set_value($\{$Math, Chemistry$\}$) S.get_odomain() = [1/9/98..3/9/98]; S.follows.get_history() = $\{\langle$[1/9/98..2/9/98], $\{\ \}\rangle$, $\langle$[3/9/98..3/9/98], $\{$Math, Chemistry$\}\rangle\}$
Date **Event** **Operation** **Result**	8/10/98 The student quits S.delete () S.status() = Off; S.follows.status() = Off; S.get_odomain() = [1/9/98..7/10/98]; S.follows.get_history() = $\{\langle$[1/9/98..2/9/98], $\{\}\rangle$, $\langle$[3/9/98..7/10/98], $\{$Math, Chemistry$\}\rangle\}$
Date **Event** **Operation** **Result**	1/2/99 The student reinstates; he enrolls in Logics and Databases courses S.revive (); S.follows.set_value($\{$Logics, Databases$\}$) S.status() = On ; S.follows.status() = On; S.get_odomain() = $\{$[1/9/98..3/9/98], [1/2/99..1/2/99]$\}$; S.follows.get_history() = $\{\langle$[1/9/98..2/9/98], $\{\}\rangle$, $\langle$[3/9/98..8/10/98], $\{$Math, Chemistry$\}\rangle$, $\langle$[1/2/99..1/2/99], $\{$Logics, Databases$\}\rangle\}$

B Semantics of the pattern description language

B.1 Abstract syntax and semantics

The syntax of the atomic formulae of the language is defined as in multi-sorted first-order logics, over an alphabet made up of constants, functions (which model object properties), variables and predefined relations ($<$, $=$, etc.). Other well-formed formulae (wff) are built from atomic formulae by using propositional operators ($\wedge$, $\vee$ and $\neg$) and the following temporal operators:

- Sequence: if f1 and f2 are wff then f1;f2 is a wff.
- Repetition: if f is a wff, θ a comparison operator ($<$, $=$, etc), n an integer term and d is duration term, then $(\mathsf{f})^{\mathsf{n}}$, $(\mathsf{f})^{+}$ and $(\mathsf{f})^{\theta \mathsf{d}}$ are wff too. The first two of these operators are counterparts of the corresponding operators on regular expressions. The third one adds a duration constraint on the patterns.

The semantics of the language is similar to that of Extended Temporal Logics (ETL) [24]. It is defined over pairs $\mathcal{H} =< \mathcal{T}, \mathcal{S} >$ made up of a temporal domain $\mathcal{T}$ (a set of instants at a given granularity), and a $\mathcal{T}$-indexed sequence of valuation functions $\mathcal{S}$ that assign values to constant, function and relation symbols for each instant in the temporal domain. Function and relation symbols are global [24], which means that they have the same interpretation over all valuation functions in $\mathcal{S}$, whereas constants may be either local (i.e. their interpretation may vary depending on the instant considered) or global. Numerical litterals (e.g. 1, 2, 3, ...) are treated as global constants whereas constants denoting structural values of a history are local.

In the following, f, f1 and f2 represent wff. The semantics of wff in a given model $\mathcal{H}$, between two instants i and j (j $>$ i) belonging to $\mathcal{T}$ and under a valuation ν of variables, is given by the satisfaction relation defined below. Intuitively, $\mathcal{H}, \nu, i, j \models$ f, iff there is an occurrence of the pattern defined by f starting at i and ending at j.

- $\mathcal{H}, \nu, i, i \models$ f (for an atomic formulae f) iff $\mathcal{S}_i, \nu \models f$ following the definition of $\models$ in first-order logics
- $\mathcal{H}, \nu, i, j \models f1 \vee f2$ iff $\mathcal{H}, \nu, i, j \models f1$ or $\mathcal{H}, \nu, i, j \models f2$
- $\mathcal{H}, \nu, i, j \models \neg f$ iff not $\mathcal{H}, \nu, i, j \models f$
- $\mathcal{H}, \nu, i, j \models f1; f2$ iff there is a $k \in \mathcal{T}$, $i \leq k < j$, such that $\mathcal{H}, \nu, i, k \models f1$ and $\mathcal{H}, \nu, k+1, j \models f2$
- $\mathcal{H}, \nu, i, j \models (f)^1$ iff $\mathcal{H}, \nu, i, j \models f$
- $\mathcal{H}, \nu, i, j \models (f)^n$ $(n > 1)$ iff there exists a $i \leq k < j, k \in \mathcal{T}$ such that $\mathcal{H}, \nu, i, k \models f$ and $\mathcal{H}, \nu, k+1, j \models (f)^{n-1}$
- $\mathcal{H}, \nu, i, j \models (f)^+$ iff there is an $n \geq 1$ such that $\mathcal{H}, \nu, i, j \models (f)^n$
- $\mathcal{H}, \nu, i, j \models (f)^{\theta d}$ iff there is an $n \geq 1$ such that $\mathcal{H}, \nu, i, j \models (f)^n$ and $(i - j)\ \theta\ d$
- As usual, $f1 \wedge f2 \equiv \neg(\neg f1 \vee \neg f2)$

B.2 Concrete syntax

The pattern description language is embedded into TEMPOQL through the following construct:

```
<query> ::= <query> as <identifier> matches <pattern>
```

The semantics of this construct is defined in terms of the Match boolean operator on histories defined in section 3.1.4. The non-terminal <pattern> is defined by the following rules, whose relative priorities are indicated through integers:

```
<pattern> ::= <query> (0) /* an atomic formula */
<pattern> ::= <pattern> followed by <pattern> (1)
<pattern> ::= several <pattern> (2)
<pattern> ::= <pattern> during [ <comparison_operator> ]  <query> (3)
<pattern> ::= <pattern> or <pattern> (4)
<pattern> ::= <pattern> and <pattern> (5)
<pattern> ::= not <pattern> (6)
<pattern> ::=  *
<pattern> ::= ( <pattern> )
<comparison_operator> ::= < | <= | ...
```

The following equivalences define the mapping from this concrete syntax to the abstract syntax.

p1 and p2 $\equiv p1 \wedge p2$	p1 or p2 $\equiv p1 \vee p2$	not p $\equiv \neg p$
p1 followed by p2 $\equiv p1; p2$	several p $\equiv p^+$	p during θ d $\equiv p^{\theta d}$ $(\theta \in \{<, >, \ldots\})$
p during d $\equiv p^{=d}$	* $\equiv$ true$^+$	

C TEMPOQL's formalization

Context: $\tau_2{<}\tau_2'{>}$ is a subtype of $History{<}\tau_2'{>}$; variable x is free in q_1 and q_3

Syntax: <query> ::= **map** <query> **on** <query> as <identifier> **when** <query>

Typing: $$\frac{q_2 :: \tau_2{<}\tau_2'{>},\ q_1[x :: \tau_2'] :: \tau_1\ q_3[x :: \tau_2'] :: boolean}{\text{map } q_1 \text{ on } q_2 \text{ as x when } q_3 :: History{<}\tau_1{>}}$$

Semantics:

$[\![\text{map } q_1 \text{ on } q_2 \text{ as x when } q_3]\!]_\nu = \text{Map}([\![q_2]\!]_\nu\ \Gamma_{if}\ \lambda v \bullet [\![q_3]\!]_{\nu[x \leftarrow v]}, \lambda w \bullet [\![q_1]\!]_{\nu[x \leftarrow w]})$

Figure 14: map: history projection

Context: $\tau_1<\tau_1'>, \ldots\ \tau_2<\tau_2'>, \ldots\ \tau_n<\tau_n'>$ are respectively subtypes of $History<\tau_1'>$, $History<\tau_2'>, \ldots\ History<\tau_n'>$; $l_1, l_2, \ldots\ l_n$ are valid labels for structured types.

Syntax: <query>:= **join (**<identifier> **:** <query> {,<identifier> **:** <query> } >

Typing: $$\frac{q_1 : \tau_1<\tau_1'>,\ q_2 : \tau_2<\tau_2'>,\ \ldots q_n : \tau_n<\tau_n'>}{\text{join } (l_1 : q_1, l_2 : q_2, \ldots, l_n : q_n) : History<struct\ (l_1 : \tau_1', l_2 : \tau_2', \ldots l_n : \tau_n')>}$$

Semantics: $\llbracket \text{join } (l_1\text{: } q_1, l_2\text{: } q_2, \ldots\ l_n\text{: } q_n) \rrbracket_\nu = \llbracket q_1 \rrbracket_\nu *_\cap \llbracket q_2 \rrbracket_\nu, *_\cap \ldots *_\cap \llbracket q_n \rrbracket_\nu$

Figure 15: join: merging histories

Context: $\tau_1<integer>$ and $\tau_2<integer>$ are subtypes of $History<integer>$ and $\theta \in \{ +, -, *, \text{div}, \text{mod} \}$

Syntax: <query> ::= <query> θ <query>

Typing: $$\frac{q_1 :: \tau_1<integer>,\ q_2 :: \tau_2<integer>}{q_1\ \theta\ q_2 :: History<integer>}$$

Semantics: $\llbracket q_1\ \theta\ q_2 \rrbracket_\nu = \text{Map } (\llbracket q_1 \rrbracket_\nu *_\cap \llbracket q_2 \rrbracket_\nu, \lambda\langle v1,v2 \rangle \bullet v1\ \theta\ v2)$

Figure 16: Generalization of the arithmetic operators on integers to two historical arguments

Context: $\tau_2<\tau_2'>$ is a subtype of $History<\tau_2'>$; variable x is free in q_3;
variable partition is free in q_1 and q_5; τ_4 is a subtype of $Unit$ or $Duration$

Syntax: <query> ::= **map** <query> **on** <query> **as** <identifier> **when** <query>
group by <query> **having**<query>

Typing: $$\frac{\begin{array}{c} q_2 :: \tau_2<\tau_2'>,\ q_1[partition :: History<\tau_2'>] :: \tau_1,\ q_3[x :: \tau_2'] :: boolean \\ q_4 :: \tau_4,\ q_5[partition :: History<\tau_2'>] :: boolean \end{array}}{\text{map } q_1 \text{ on } q_2 \text{ as x when } q_3 \text{ group by } q_4 \text{ having } q_5 :: History<\tau_1>}$$

Semantics: $\llbracket \text{map } q_1 \text{ on } q_2 \text{ as x when } q_3 \text{ group by } q_4 \text{ having } q_5 \rrbracket_\nu =$

$$\begin{cases} \text{Map (UGroup (}\llbracket q_2 \rrbracket_\nu\ \Gamma_{if}\ \lambda v \bullet \llbracket q_3 \rrbracket_{\nu[x\leftarrow v]}, \llbracket q_4 \rrbracket_\nu) \\ \quad \Gamma_{if}\ \lambda w \bullet \llbracket q_5 \rrbracket_{\nu[\text{partition}\leftarrow w]}, \lambda y \bullet \llbracket q_1 \rrbracket_{\nu[\text{partition}\leftarrow y]}) & \text{if } \tau_4 \text{ subtype of Unit} \\ \text{Map (DGroup (}\llbracket q_2 \rrbracket_\nu\ \Gamma_{if}\ \lambda v \bullet \llbracket q_3 \rrbracket_{\nu[x\leftarrow v]}, \llbracket q_4 \rrbracket_\nu) \\ \quad \Gamma_{if}\ \lambda w \bullet \llbracket q_5 \rrbracket_{\nu[\text{partition}\leftarrow w]}, \lambda y \bullet \llbracket q_1 \rrbracket_{\nu[\text{partition}\leftarrow y]}) & \text{if } \tau_4 \text{ subtype of Duration} \end{cases}$$

Figure 17: group by: temporal grouping

Preconditions: $\tau<\tau'>$ is a subtype of History$<\tau'>$

Syntax: <query>:= **xchronicle (**<query>**)**

Typing: $$\frac{q : \tau<\tau'>}{\text{xchronicle } (q) : list<struct<tvalue : Interval, svalue : \tau'>>}$$

Semantics: $\llbracket \text{xchronicle (q)} \rrbracket_\nu = \text{XChronicle } (\llbracket q \rrbracket_\nu)$

Figure 18: XChronicle: transforming a history into a collection of interval-timestamped objects

Preconditions: $\tau<\tau'>$ is a subtype of $History<\tau'>$; variable x is free in q_2

Syntax: <query>:= <query> **as** <identifier> **afterfirst** <query>

Typing: $$\frac{q_1 : \tau<\tau'>,\ q_2[x : \tau'] : boolean}{q_1 \text{ as x afterfirst } q_2 : History<\tau'>}$$

Semantics: $\llbracket q_1 \text{ as x afterfirst } q_2 \rrbracket_\nu = \text{AfterFirst } (\llbracket q_1 \rrbracket_\nu, \lambda v \bullet \llbracket q_2 \rrbracket_{\nu[x\leftarrow v]})$

Figure 19: afterfirst: succession in time - splitting histories