A representation-independent temporal extension of ODMG's Object Query Language

Marie-Christine Fauvet, Marlon Dumas and Pierre-Claude Scholl
LSR-IMAG, Université Joseph Fourier, Grenoble
BP 72, 38402 St Martin d'Hères (France)
E-mail: Marie-Christine.Fauvet@imag.fr

Abstract

TEMPOS is a set of models providing a framework for extending database systems with temporal functionalities. Based on this framework, an extension of the ODMG's object database standard has been defined. This extension includes a hierarchy of abstract datatypes for managing temporal values and histories, as well as temporal extensions of ODMG's object model, schema definition language and query language. This paper focuses on the latter, namely TEMPOQL. With respect to related proposals, the main originality of TEMPOQL is that it allows to manipulate 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. In fact, TEMPOQL goes further in preserving OQL's structure, by generalizing most standard OQL constructs to deal with histories. The overall proposal has been fully formalized both at the syntactical and the semantical level and implemented as a preprocessor.

Keywords: temporal databases, temporal data models, temporal query languages, time representation, object-oriented databases, ODMG, OQL

1 Introduction

Temporal Databases aim at integrating time as a primitive concept in DBMS. Over the past two decades, research in this area has been quite prolific regarding temporal extension proposals to data models and query languages. Whereas in the relational framework these works have led to the consensus temporal query language TSQL2 [Sno95b], as well as two proposals to the ISO SQL3 standardization committee [Mel97, SBJS98], equivalent results are missing in the object-oriented framework. Early attempts to define temporal extensions of object-oriented data models (see [Sno95a] for a survey) failed to become widely accepted, essentially because they were not based on a standard underlying data model. As the ODMG's [CBB+97] proposal appeared and started to be adopted by the major object DBMS vendors, the idea of temporally extending it arose. Up to now and to our knowledge, three such extensions have been proposed, namely TAU, TOOBIS and T_ODMG, respectively reported in [KT96], [TOO96] and

[BFGM98]. However, we argue that these proposals neglect at least some of the following three important aspects[1]: (i) representation independence as to clearly separate the abstract and the representational views of temporal concepts and constructs; (ii) formal semantics as to avoid ambiguities generated by the richness and complexity of temporal concepts and to serve as a basis for efficient implementation; (iii) temporal migration support as to ensure a seamless transition of applications running on top of a non-temporal system to a temporal extension of it

TEMPOS (Temporal Extension Models for Persistent Object Servers) [SFC98, DFS98, DFS99] is a general framework for designing DBMS temporal extensions integrating the above three features. It includes a set of abstract datatypes for handling temporal values and histories, and a generic model for integrating temporal support both at the class extent and the property level.

Based on TEMPOS, an extension of the ODMG's object database standard has been designed. This paper focuses on the query language, namely TEMPOQL, defined as part of this work.

The paper is organized as follows. Section 2 defines the notion of representation independence and discusses related works on this issue. In order to make the paper self-contained, we review in section 3 the basic concepts of the TEMPOS model, previously presented in [SFC98, DFS98]. Next, in section 4, we introduce TEMPOQL and include some example queries to illustrate it. Before concluding in section 6, we describe in section 5 the prototype that has been developed to validate the proposal's feasibility.

2 Motivations and related works

2.1 Point-based vs. interval-based temporal data models

A temporal association is as a piece of data locating a fact on 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 [Cho94]. An interval-based association states that a fact holds over some interval, e.g. *"The stock price raised by 50% between 1995 and 1999"*, without entailing that the fact holds at each instant in the interval. On the other hand, a point-based association states that a fact holds at some instant observed with some precision (e.g. *"The salary of an employee is 800 at January 98"*).

Temporal data models may be classified into point-based or interval-based, depending on the kind of temporal associations that they manage [BBJ98]. 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 it 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. SQL/Temporal [SBJS98]) or because they distinguish point-

[1] To some extent, this remark also applies to other temporal object models (e.g. [GO93, RS93, SN97])

based from interval-based associations (e.g. TOOBIS [TOO96]). It is worth noting that no temporal data model is actually purely interval-based. This is because the vast majority of temporal applications deal with point-based associations. Based on this argument, TEMPOS has been designed as a point-based data model, though we expect to extend it to deal with interval-based associations as well.

2.2 Representation independence in point-based data models

In temporal data models, related temporal associations are grouped into temporal relations or into object or attribute histories.

Point-based temporal relations or histories may be represented in several ways. For instance, it is possible to attach an instant timestamp to every tuple in a temporal relation at the logical level [Tom98]. An alternative is to group several value-equivalent tuples into a single one, timestamped either by a set of instants (also called *temporal element*) [GN93], 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: 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, leading to some undesirable tensions between query expressions and their intended semantics [Tom98].

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)
```

This example shows a clear difference in the level of abstraction between the query expression in natural language and in TOQL: in the former, 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 examples of such references (according to the informal semantics provided in [TOO96]). In addition, the notion of "since the first time" is not explicitly expressed in terms of "since" and "first time" in the above formulation. This mismatch reflects a lack of operators on histories allowing 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 temporal operators, or for query expression, is an undesirable feature in a temporal data model, and especially in an object-oriented one, since it goes against encapsulation. Instead, specific representation independent operators should be provided, covering the fundamental temporal reasoning paradigms: succession, simultaneity, granularity change, etc. This approach is different from those adopted in [RS93], [Sno95b], [TOO96] and [SN97], and more or less similar to those described in [WD93] and [GBE+98].

Nevertheless, we do not mean that setting 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* [Sno95b], so that temporal query languages should also provide facilities for reasoning on a particular representation of histories.

The above considerations also apply to the modeling of temporal values. For instance, in most temporal data models, sets of instants are modeled as sets of disjoint non-contiguous intervals, termed *temporal elements*. While this representation is 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 that a temporal data model is representation independent if (i) it defines all operators on temporal values and associations independently of any particular 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 includes a time model, which defines a hierarchy of simple and complex datatypes for modeling temporal values, and a historical model, which provides representation independent algebraic operators on histories.

3.1 Overview of the time model

TEMPOS is based upon a discrete, linear and bounded time model in which the time line is structured in a *multi-granular* way by means of *time units* [WJS95]. A time unit is a partition of the time line into convex sets: the elements of this partition are then seen as atomic *granules* and every point of the time line is approximated by the granule which contains it.

If a mapping can be established between each granule of a time unit u1 and a set of consecutive granules of another unit u2, then u2 is said to be *finer than* u1 (u2 $\prec$ u1), or symmetrically, u1 is said to be *coarser than u2*. For instance, the time unit *month* is finer than *year* but coarser than *day*. The finer than relation is used to manage conversions between temporal values at different granularities.

Based on this temporal structure, the TEMPOS model defines a wide variety of temporal datatypes are defined (e.g. instants, durations, sets of instants), which may be expressed at any time unit.

The set of operators defined over these types are independent both of the involved units and of any particular representation as to fulfill the requirement stated above. In particular, a temporal sequence (TSequence) is abstractly defined as a finite, chronologically ordered sequence of instants observed at some granularity, with no assumption on its representation. Intervals are then seen as particular cases of temporal sequences, whose characteristics allow to define specific operators over them.

For a detailed description of these temporal ADT the reader may refer to [SFC98, DFS99].

3.2 Histories and their representations

At an abstract level, a *history* is defined as a function from a finite set of instants observed at a fixed granularity, 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. The following notations are used: T1 $\rightarrow$ T2 stands for the type of all functions with domain T1 and codomain T2. $\{$T$\}$ and [T] respectively denote the type of sets of T and sequences of T. $\langle$T1, T2, ..., Tn$\rangle$ designates the type of tuples whose ith component is of type Ti $(1 \leq i \leq n)$; tuple components may be labeled using the notation $\langle$L1: T1, L2: T2, ..., Ln: Tn$\rangle$. T1$<$T2$>$ denotes an instantiation of the parameterized type T1 with type T2. $\langle$v1, v2, ..., vn$\rangle$ denotes a tuple value whose ith component is vi $(1 \leq i \leq n)$. Finally, if x has a temporal type, then Unit (x) denotes its observation unit.

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

type History$<$T$>$ = Instant $\rightarrow$ T
TDomain: History$<$T$>$ $\rightarrow$ TSequence /* *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 of H at instant I* */
/* *precondition: I* $\in$ *TDomain (H)* */
SDomain: History$<$T$>$ $\rightarrow$ $\{$ T $\}$ /* *structural domain* */

In addition to these, five other selectors are defined on histories, corresponding to the well-known aggregate operators HSum, HAvg, HMax, HMin and HCount. This latter retrieves the cardinality of a history expressed as a duration.

Histories may be represented in several ways, mainly by means of collections of timestamped values, termed *chronicles*. Concretely, a history may be represented by at least three kinds of chronicles:

- IChronicle (instant-based representation): chronologically ordered sequence 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$].
- XChronicle ("maximal" interval-based representation): chronologically ordered, coalesced sequence of interval-timestamped values, e.g. [$\langle$[1..2], v1$\rangle$, $\langle$[4..4], v1$\rangle$, $\langle$[5..7], v2$\rangle$, $\langle$[8..8], v3$\rangle$, $\langle$[9..10], v1$\rangle$].
- DChronicle (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$\rangle$ $\}$.

These representations are sometimes useful for query expression, so that specific operators are defined that cast either of these representations into histories and vice-versa [DFS99]. Here below we only describe the operator allowing to cast histories into XChronicles.

XChronicle: History$<$T$>$ $\rightarrow$ [$\langle$tvalue: Instant, svalue: T$\rangle$]
/* *XChronicle (h)* = $[C_1, \ldots, C_n] \Rightarrow \forall k \in [1..n\text{-}1]$,
$C_k.tvalue < C_{k+1}.tvalue \wedge (C_k.tvalue \text{ meets } C_{k+1}.tvalue \Rightarrow C_k.svalue \neq C_{k+1}.svalue)$ */

3.3 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*. This classification is closed under composition.

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 $\rightarrow$ boolean) $\rightarrow$ 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 $\rightarrow$ History<T> /* $h\ \Gamma_{in}\ S = \{\langle I,v\rangle \mid \langle I,v\rangle \in h \wedge I \in S\}$ */

The intra-point Map pointwisely applies a function to each structural value of a history.

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

The *intra-point* temporal join allows to combine 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: 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); h1 $*_\cup$h2 is similar, except that it attaches structural values of the form $\langle$v, Nil$\rangle$ or $\langle$Nil, v$\rangle$, to those instants where one of the argument histories is defined while the other is not. More precisely:

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

Inter-point operators, on the other hand, include groupings, and operations dealing with succession in time.

There are two grouping operators in TEMPOS: UGroup and DGroup. UGroup (h, u2), h being at granularity u1 (u1 $\prec$ u2), divides up history h into groups according to unit u2. The result is a history at granularity u2 of histories at granularity u1 whose values at instant i are the temporal restriction of h to the interval expand (i, u1)[2]. This is illustrated in figure 1.

On the other hand, DGroup (h, d) yields all sub-histories of h of 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 included in the temporal domain of h. Formally:

[2]Operation expand (i, u) maps an instant i observed at some unit u2, to an interval at a finer unit u1.

UGroup: History<T>, Unit $\rightarrow$ History<History<T>> *\/* precondition: Unit (h) $\prec$ u *\/*

\/ UGroup (h, u) =*

$\{ \langle I, subh \rangle \mid \exists\, I' \in TDomain\ (h),\ approx\ (I', u) = I \wedge h\, \Gamma_{in}\ expand\ (I, Unit\ (h)) = subh \}$ */

DGroup: History<T>, Duration $\rightarrow$ History<History<T>>*\/* precondition: Unit (h) = Unit (d) = Unit (I) *\/*

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

DGroup 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"*.

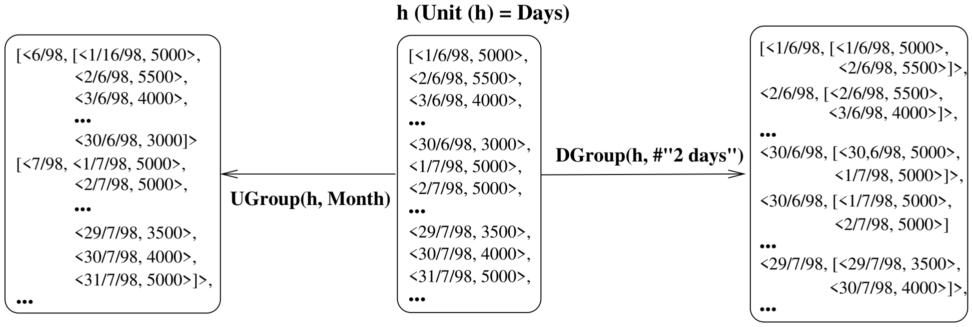

Figure 1: *Unit-based and duration-based temporal grouping*

To reason about successive values of histories and their correlations, TEMPOS provides four algebraic operators, namely AfterFirst, 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. On the other hand, BeforeFirst (h, P), 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 = BeforeFirst (h, P) $\cup$ AfterFirst (h, P) (which are disjoint). Similar remarks apply to AfterLast and BeforeLast.

We give the specification of AfterFirst and BeforeFirst. The other two are defined similarly.

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>

/* $BeforeFirst\ (h, P) = \{ \langle I, v \rangle \mid \langle I, v \rangle \in h \wedge \neg\exists \langle I', v' \rangle \in h\ (P\ (v') \wedge I \geq I') \}$ */

3.4 Boolean pattern-matching

Queries about succession in time intensively deal with the sequenced structure of histories. For this reason, TEMPOS offers a language for describing "patterns" of histories based on this sequenced structure.

A similar language has been proposed in [CG94]. The main originality of the TEMPOS approach, with respect to this latter, lies on the use of regular expression operators. Indeed, the TEMPOS pattern definition language integrates classical logics with timed regular expressions. The former are used to build up atomic formulae and to combine them into complex boolean conditions for expressing properties about the structural value of a history at a given instant, while the latter are used to reason about succession in time. This language allows to express :

- Sequencing (followed by): for example, the pattern "production $<$ 100 followed by production $>$ 200" characterizes *factory items whose production is less than 100 immediately before being greater than 200*.
- Repetition (several) allows to reason on repetition of a pattern: for example, the pattern "production $<$ 100 followed by (several production $>$ 200) followed by production $<$ 100" is used to retrieve factory items whose production was less than 100, then raised above 100, and subsequently became less than 100 again.
- Duration constrained repetition: for example, the pattern "(production = 0) at least ♯"2 days"" characterizes items whose production was null during more than 2 days.

The syntax of the pattern matching language follows. The integers following each rule indicate its relative priority.

<pattern> ::= <pattern> **followed by** <pattern> (1)
<pattern> ::= **several** <pattern> (2)
<pattern> ::= <pattern> **during** [<comparison_operator>] <query> (3)
/* "p during d" ≡ pattern "p" has a duration equal to "d" */
<pattern> ::= <pattern> **or** <pattern> (4)
<pattern> ::= <pattern> **and** <pattern> (5)
<pattern> ::= **not** <pattern> (6)
<pattern> ::= * /* ≡ "several (true)" */
<pattern> ::= (<pattern>)
<comparison_operator> ::= < | <= | ...

The corresponding semantics is formally described in [DFS98]. In this latter work, we also introduce a boolean pattern-matching operator Match, which checks if a given history contains at least one occurrence of a pattern.

4 TEMPOQL

Basically TEMPOQL adds some types to OQL (e.g. time unit, instant, interval, history) together with some constructs over them. Most of these constructs are counterparts of the operators defined in the time and historical models. Notice that the term *construct* is used to designate operators of the query language, so as to distinguish them from the abstract operators on histories introduced above, which are actually defined independently of OQL.

As a working example to illustrate TempOQL's constructs, we consider a class Product with a "non-temporal" property trademark of type string and two properties price and production whose values are histories observed at the granularity of the day, and having structural values of type real and integer respectively. The extent of class Product is named TheProducts.

In the rest of the section, we give a formalization of the main constructs of the language, and illustrate its ease of use and expressive power through a set of example queries. These examples are organized so as to put forward the temporal reasoning paradigms addressed by each of the constructs of the language: simultaneity, granularity change, duration constraints, succession in time, continuity and change. Although this classification is quite informal, we believe that it may be useful during query expression, specially when identifying the operators that may get involved in a given context.

4.1 Formalization of temporal constructs

In the following, we describe most of the constructs for handling histories in TEMPOQL. For each construct, we give in figures 2 and 3, its typing rules, its syntax and its semantics, using a notation similar to that of [RS97], which provides a complete formalization of OQL queries. More precisely, the formalization of a construct is made up of four parts:

– Context: it describes subtyping 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.

– Syntax: in a BNF-like notation with terminal symbols typeset in boldface.

– Typing: a typing rule 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'.

– Semantics: described in terms of expressions involving operators of the TEMPOS historical model. The semantics of a query is parameterized by a valuation function which determines the values of free symbols in the query. $\nu[\mathrm{x} \leftarrow \mathrm{v}]$ 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.

4.2 Restriction

As in the relational algebra, one of the most elementary operations on histories is the extraction of some portion of them based on a given condition. In TEMPOQL, this is expressed through the during and when constructs (see Figure 2a and 2b respectively), as illustrated below.

Q.1 : **Restriction according to the temporal domain**

For each product, retrieve its trademark and the history of its production during 1997.

```
/* query type: bag<struct<T: string, P: History<real>>>  */
select struct (T: p.trademark, P: p.production during @"1997") from TheProducts as p
  /* symbol @ is used to introduce an instant litteral. Notice that the instant at the granularity of the
  year @"1997", is automatically expanded into interval [@"1/1/1997"...@"31/12/1997"]. */
```

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: $[\![\mathrm{q_1} \text{ during } \mathrm{q_2}]\!]_\nu = \begin{cases} [\![\mathrm{q_1}]\!]_\nu \ \Gamma_{in}\ [\![\mathrm{q_2}]\!]_\nu & \text{if } \tau_2 \text{ subtype of TSequence} \\ [\![\mathrm{q_1}]\!]_\nu \ \Gamma_{in} & \text{if } \tau_2 \text{ subtype of Instant and} \\ \text{expand } ([\![\mathrm{q_2}]\!]_\nu, \text{Unit } ([\![\mathrm{q_1}]\!]_\nu)) & \text{Unit } ([\![\mathrm{q_1}]\!]_\nu) \prec \text{Unit } ([\![\mathrm{q_2}]\!]_\nu) \end{cases}$

(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: $[\![\mathrm{q_1} \text{ as x when } \mathrm{q_2}]\!]_\nu = [\![\mathrm{q_1}]\!]_\nu \ \Gamma_{if}\ \lambda \mathrm{v} \bullet [\![\mathrm{q_2}]\!]_{\nu[\mathrm{x} \leftarrow \mathrm{v}]}$

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

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

Semantics: $[\![\text{join } (\mathrm{l_1}{:}\ \mathrm{q_1}, \mathrm{l_2}{:}\ \mathrm{q_2}, \ldots\ \mathrm{l_n}{:}\ \mathrm{q_n})]\!]_\nu = [\![\mathrm{q_1}]\!]_\nu *_\cap [\![\mathrm{q_2}]\!]_\nu, *_\cap \ldots *_\cap [\![\mathrm{q_n}]\!]_\nu$

(c) *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: $\dfrac{q_1 :: \tau_1 < integer >,\ q_2 :: \tau_2 < integer >}{\mathrm{q_1}\ \theta\ \mathrm{q_2} :: History < integer >}$

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

(d) *Generalizing arithmetic operators on integers to two historical arguments*

Context: $\tau < integer >$ is subtype of $History < integer >$ and $\theta \in \{ +, -, *, \text{div}, \text{mod} \}$

Syntax: <query> ::= <query> θ <query>

Typing: $\dfrac{q_1 :: \tau < integer >,\ q_2 :: integer}{\mathrm{q_1}\ \theta\ \mathrm{q_2} :: History < integer >}$

Semantics: $[\![\mathrm{q_1}\ \theta\ \mathrm{q_2}]\!]_\nu = \text{Map } ([\![\mathrm{q_1}]\!]_\nu, \lambda \mathrm{v} \bullet \mathrm{v}\ \theta\ [\![\mathrm{q_2}]\!]_\nu)$

(e) *Generalizing arithmetic operators to one historical and one integer argument*

Figure 2: *Semantics of* TEMPOQL*'s intra-point constructs*

Q.2 : **Restriction according to the structural domain**

For each product, retrieve its trademark and the history of its prices when it was greater than 100.

```
/* query type: bag<struct<T: string, P: History<float>>>  */
select struct (T: p.trademark, P: p.price as aprice when aprice > 100) from TheProducts as p
```

These restriction operators do not express any temporal reasoning paradigm by themselves, but rather when composed with other operators as shown in the following paragraphs.

4.3 Reasoning on simultaneity: 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 2c provides a formal definition of the join construct. The construct ojoin is defined in a similar way. Recall that the structural values of a history resulting from an outer join may contain references to null values. Therefore, if the structural values of the argument histories are of non-nullable literal types τ_1 and τ_2 (e.g. integer and real), then the structural values of the result are of type $struct{<}label1 : nullable\_\tau_1, label2 : nullable\_\tau_2{>}$ (see [CBB+97] for details on nullable types in ODMG).

Since both of these constructs build pairs of synchronous values taken by two histories, they allow 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 product is greater than that of another product.

Q.3 : **Temporal join**

When was the product with trademark T1 produced in larger quantity than product with trademark T2?

```
/* query type: TSequence  */
element (
   select tdomain ( join (prod1: p1.production, prod2: p2.production) as j when (j.prod1 > j.prod2) )
   /* "tdomain" retrieves the temporal domain of a history */
   from TheProducts as p1, TheProducts as p2 where p1.Trademark ="T1" and p2.Trademark = "T2")
```

In this query, ojoin could be used to take into account that when T2 has no production, the production of T1 may be considered as being greater than that of T2 :

```
element (select tdomain (ojoin (prod1: p1.production, prod2: p2.production)
                         as oj when oj.prod1 != nil and (oj.prod2 = nil or (oj.prod1 > oj.prod2)))
         from TheProducts as p1, TheProducts as p2
         where p1.Trademark ="T1" and p2.Trademark = "T2")
```

4.4 Reasoning on simultaneity: 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 gen-*

eralization principle[3]: given an N-ary operator θ: $\tau_1, \ldots \tau_n \rightarrow \tau_{n+1}$, an operator θ: History<τ_1>, ... History<τ_n> $\rightarrow$ History<τ_{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 figure 2d). 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:

```
element ( select tdomain ((p1.production > p2.production) as g when g)
from TheProducts as p1, TheProducts as p2 where p1.trademark = "T1" and p2.trademark = "T2")
```

In the case of binary operators, the generalization principle can be applied as soon as one of the arguments is a history (see figure 2e), as illustrated by the following query.

Q.4 : **Pointwise generalization of OQL constructs**

Retrieve the VAT-included price history of each product, assuming that the VAT is 20% (the prices given by the price *attribute do not include the VAT).*

```
/* query type: struct<product: Product, vat_price: History<float>> */
select struct (product: p, vat_price: p.price * 1.2) from TheProducts as p
```

Similarly, OQL's "dot" operator on structured types and objects, is extended 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.

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. An ongoing effort aims at applying the generalization principle to the select ... from ... where construct on collections.

4.5 Granularity change and constant duration constraints

Temporal grouping is the process of dividing the contents of a given history into several ones, based on some criteria. As discussed, in section 3.3, in TEMPOS there are two elementary kinds of temporal grouping: a unit-based and a duration-based. In TEMPOQL, these two operators are embedded into a single construct as described in figure 3. Notice that the choice of either of the operators in the semantics of the construct, depends on the type of the query appearing in the group by clause. Notice also that

[3]This principle has been used in dataflow programming languages (e.g. LUSTRE [CPHP87]), and in some temporal relational query languages [SBJS98, GBE+98].

keyword partition is used in the map and having clauses to refer to the sub-histories generated by the grouping.

The unit-based grouping is usually involved when transforming a history from a given granularity to a coarser one. In query **Q.5** below, this operator is used to transform histories at the granularity of the Day, into histories at the granularity of the Month.

Q.5 : **Aggregations and unit-based grouping**

For each product, and for each month when this product was produced, retrieve the number of days during which its production was greater than 100 on that month.

```
/* query type: bag<struct<trademark: string, production: History<Duration>>>  */
select struct (trademark: p.trademark,
               production: map hcount (partition) /* partition is a history at the granularity of the day. */
               /* "hcount" retrieves the cardinality of a history expressed as a duration (see 3.3) */
                           on p.production as prod when prod > 100
                           group by month)
from TheProducts as p
```

Duration-based partitioning on the other hand, is classically involved in calculation of moving-window aggregations, as in query **Q.6** below. More generally, it may be used to express that a given condition holds during a fixed duration as in *"which bottles had a constant price during a period of at least three months*.

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

For each product, and for each 10-day period during which the total production of this product is greater than 1000, retrieve its average production on that period.

```
/* query type: bag<struct<trademark : string, avgprod : History<real>>>  */
select struct (trademark: p.trademark,
               avgprod: /* The following sub-query retrieves 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 > 1000 */
                           map havg (partition)
                           on p.production as prod
                           group by #"10 days" having hsum (partition) > 1000
from TheProducts as p
```

4.6 Reasoning about succession in time

The afterfirst beforefirst, afterlast and beforelast constructs are straightforward adaptations of the corresponding operators on histories. Their syntax is similar to that of the when construct.

These constructs allow to reason about succession in time. For instance, in the following query, the history of the production of each product, is restricted to those instants *following* the first instant when a given condition holds.

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

Semantics: $[\![\text{map } q_1 \text{ on } q_2 \text{ as x when } q_3 \text{ group by } q_4 \text{ having } q_5]\!]_\nu$ =

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

Figure 3: Semantics of "group by"

Q.7 : **Succession in time: splitting histories**

For each product, retrieve its trademark and its production history since the last time its production was smaller than 100.

```
/* query type: set<struct<trademark: string, hprod: History<unsigned long>> */
select struct (trademark: p.trademark, hprod: p.production as prod afterlast prod ≤ 100)
from TheProducts as p
```

By composing these four constructs among themselves and with the other constructs of the language, it is possible to express queries involving complex sequences of events. However, the complexity of these query expressions rapidly increases with the complexity of the involved sequence. In addition, expressing duration constraints in this way involves composing these constructs with the "group by" construct discussed above, leading to still more complicated query expressions.

Our approach to tackle this complexity is to embed into TEMPOQL the pattern-description language described in section 3.4 and the corresponding Matches boolean operator. In this way, queries involving selection of histories based on a sequence of events related by duration constraints, may be expressed in an elegant way. In addition, they can be efficiently evaluated as discussed in 5.

Q.8 : **Succession in time: pattern-matching**

Which clients ordered bottle-type T1, then afterwards bottle-type T2, and at most three days after ordering T2, ordered T1 again? We suppose that T1 and T2 are variables referencing some bottle-types.

```
select c from TheClients as c where c.orders as ords matches
        T1 in ords followed by * followed by T2 in ords followed by * during "♯ 3 days"
                  followed by T1 in ords
```

4.7 Reasoning on continuity and change

While history splitting operators and pattern-matching ones are suitable for expressing "succession" (events coming one after the other in time), they cannot express continuity and change. In particular, they cannot express the queries *"how many times did the price of bottle-type T1 raised"*, nor *"what is the longest interval of time during which the price of bottle-type T1 remained constant"*.

It is at this point that it becomes necessary to manipulate histories through their representation as collections of "maximal" interval-timestamped values. In TEMPOQL, this is done using the xchronicle construct, which casts a history into a collection of interval-timestamped values as discussed in section 3.2.

Q.9 : **Reasoning on a "maximal" interval-based representation**

For each product, give its trademark and the longest time interval (s) during which its production was greater than 100.

```
/* query type: set<string, bag<Interval>> */
select struct (trademark: p.trademark, period: xsmax.tvalue)
from TheProducts as p, xchronicle (p.production as prod when prod > 100) as xsmax
where hcount (xsmax.tvalue) = max (select hcount (xs.tvalue)
                                   from xchronicle (p.production as prod when prod > 100) as xs)
```

5 Implementation

5.1 Overall architecture

TEMPOS has been implemented on top of the object-oriented DBMS O_2. This implementation consists of a library of classes corresponding to the ADT hierarchies defined in the the time and historical models, and of two preprocessors implementing respectively TEMPODL and TEMPOQL. A temporal metadata manager accounts for the communication between these two preprocessors. An ongoing effort aims at designing and implementing an API of this module compatible with the meta-object interface defined by ODMG 2.0. Figure 4 illustrates the prototype architecture.

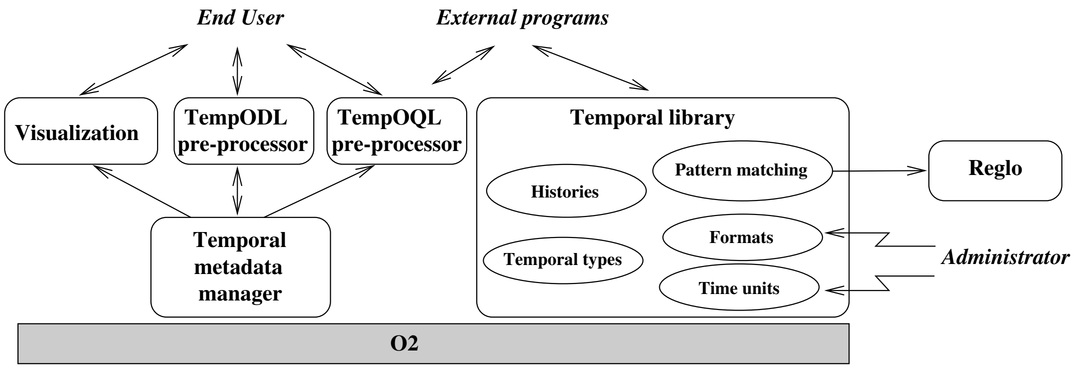

Figure 4: *Prototype architecture*

The library of temporal classes has been entirely implemented in the O_2's proprietary database programming language O_2C, while the preprocessors and the metadata manager have been primarily implemented in C using code generation tools such as Lex and Yacc.

The visualization module referenced in the architecture is part of an ongoing work on applying visualization paradigms to the interactive analysis of data resulting from temporal queries. As the preprocessor, this module uses metadata related to the temporal classes and properties.

Another module referenced in the architecture implements the pattern-matching operator described in section 3.4. We discuss the implementation of this module in section 5.3.

The TEMPOS prototype has been used to implement 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.

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 3.3).

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 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).

We believe that the above considerations illustrate the necessity of extending the current ODMG object model to allow user-defined parametric classes, as discussed in [Ala97].

5.3 Evaluation of the pattern-matching operator

In the next paragraphs, we describe the algorithm that we have implemented for evaluating the boolean pattern-matching operator Match (see section 3.4). 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 [Ray96] 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[4], 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 3.4).

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.

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

6 Conclusion and future work

We have presented the query language associated to the TEMPOS temporal database model described in [SFC98, DFS98]. This model unifies many of the concepts and facilities recognized as necessary to temporally extend existing DBMS, and integrates them into a temporal extension of the ODMG standard.

The proposed query language, namely TEMPOQL, is an extension of ODMG's Object Query Language (OQL). It 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 [RS93, TOO96, SN97], queries on histories are expressed by applying iterators over collections of interval-timestamped values representing them.
- It integrates some novel features 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 [Sno95b]), 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 [CPHP87]), TEMPOQL extends the semantics of standard OQL constructs to deal with histories.

Foremost, TEMPOQL has been fully formalized both at the syntactical and the semantical levels and implemented as a preprocessor. This implementation has been used to test the suitability of the proposed extensions in regard to concrete applications' requirements.

Currently, we are planning to extend this work in several directions:

- Query evaluation: studying the integration of temporal evaluation techniques, and specially temporal access methods into the TEMPOS architecture.
- User interaction: developing and applying visualization paradigms and tools to ease the interactive analysis and manipulation of data resulting from temporal queries.
- Expressive power: Characterize families of queries expressible or inexpressible in TEMPOQL.

Acknowledgments: we wish to thank Jean-François Canavaggio for his participation in the initial design and implementation of TEMPOS.

References

[Ala97] 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.

[BBJ98] 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.

[BFGM98] 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.

[CBB+97] R.G.G. Cattell, D. Barry, D. Bartels, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[CG94] 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.

[Cho94] J. Chomicki. Temporal Query Languages: A Survey. In *proc. of the International Conference on Temporal Logic*, Bonn, DE, 1994.

[CPHP87] 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.

[DFS98] 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.

[DFS99] M. Dumas, M.-C. Fauvet, and P.-C. Scholl. TEMPOS: A Temporal Database Model Seamlessly Extending ODMG. Research report 1013-I-LSR-7, LSR-IMAG, Grenoble (France), March 1999. 45 pages.

[EJS98] O. Etzion, S. Jajodia, and S.M. Sripada, editors. *Temporal Databases: Research and Practice*. Springer Verlag, LNCS 1399, 1998.

[GBE+98] R. Güting, M. Böhlen, M. Erwig, C. Jensen, Lorentzos N, M. Schneider, and M. Vazirgianis. A foundation for representing and querying moving objects. Technical Report 238, FerUniversität das Hagen (Germany), 1998.

[GN93] S. K. Gadia and S. S. Nair. Temporal databases: a prelude to parametric data. In Tansel et al. [TCG+93].

[GO93] 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.

[KT96] I. Kakoudakis and B. Theodoulidis. The TAU Temporal Object Model. Technical Report TR-96-4, TimeLab, University of Manchester (UMIST), 1996.

[Mel97] J. Melton, editor. *SQL/Temporal*. ISO/IEC JTC1/SC21/WG3 DBL-LGW-013, 1997.

[Ray96] 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.

[RS93] E. Rose and A. Segev. TOOSQL - a temporal object-oriented query language. In Springer Verlag, editor, *proc. of the 12th International Conference on the Entity-Relationship Approach - ER'93, LNCS 823*, 1993.

[RS97] 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.

[SBJS98] R.T. Snodgrass, M. Böhlen, C. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In Etzion et al. [EJS98].

[SFC98] P.-C. Scholl, M.-C. Fauvet, and J.-F. Canavaggio. Un modèle d'historique pour un SGBD temporel. *TSI*, 17(3), mars 1998. Numéro thématique "Bases de données".

[SN97] 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.

[Sno95a] 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.

[Sno95b] R. T. Snodgrass, editor. *The TSQL2 temporal query language*. Kluwer Academic Publishers, 1995.

[TCG+93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodggrass, editors. *Temporal Databases*. The Benjamins/Cummings Publishing Company, 1993.

[Tom98] D. Toman. Point-based temporal extensions of SQL and their efficient implementation. In Etzion et al. [EJS98].

[TOO96] TOOBIS ESPRIT Project. TOQL specification. Deliverable T33TR.1, University of Athens, National Technical University of Athens, 01-PLIFORIKI S.A., O2 Technology, December 1996.

[WD93] G.T.J. Wuu and U. Dayal. A uniform model for temporal and versioned object-oriented databases. In Tansel et al. [TCG+93].

[WJS95] X. S. Wang, S. Jajodia, and V. S. Subrahmanian. Temporal modules : an approach toward federated temporal databases. *Information Systems*, 82, 1995.