Updates and application migration support in an ODMG temporal extension

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

LSR-IMAG Laboratory, University of Grenoble,
BP 72, 38402 St Martin d'Hères, France
Marlon.Dumas@imag.fr

Abstract. A substantial number of temporal extensions to data models and query languages have been proposed. However, little attention has been paid to the migration of data and applications from a "snapshot" DBMS to a temporal extension of it. In this paper, we analyze this issue and precisely formulate some requirements related to it. We then present a temporal extension of the ODMG's object database standard fulfilling these requirements. Throughout this presentation, we underscore the importance of providing adequate update and interpolation modalities in achieving application migration support.
Keywords: temporal databases, application migration, schema evolution, temporal updates, object-oriented databases, ODMG

1 Introduction

Temporal databases aim at integrating time as a primitive concept in DBMS. Research in this area has given birth to a substantial number of temporal data models, most of which are actually extensions of existing "snapshot" ones. A common rationale for temporally enhancing snapshot data models rather than designing temporal ones from scratch, is that the resulting models can be easily integrated into existing systems. The applications built on top of these systems may then rapidly benefit from the added technology.

However, the smooth migration of applications running on top of snapshot database systems to temporal extensions of them, imposes some compatibility requirements. Surprisingly, this issue has been neglected by the temporal database community until relatively recently, when notions such as *seamless integration of time* [7] and *temporal upward compatibility* [1, 12] have been studied.

The first part of this paper precisely defines some requirements related to data and application migration towards temporal DBMS extensions, with an emphasis on object-oriented ones. The main originality of the adopted approach is to view this problem as a particular case of schema evolution.

We then present a temporal extension of ODMG's object database model [3] fulfilling the above migration requirements. This extension directly stems from the TEMPOS object-oriented temporal database framework [6, 4]. It defines a set of abstract datatypes for handling temporal values and histories, and provides temporal extensions of the main ODMG components: the object model, the schema definition language and the query language. In this paper we focus on the update and interpolation modalities defined by the extended object model, and put forward their major role in fulfilling the migration requirements.

The paper is organized as follows. Section 2 introduces the concepts related to application migration towards temporal DBMS. Section 3 overviews the proposed extension of ODMG's model. A description of the update operators provided by this extension is given in section 4. Section 5 shows how application migration is achieved. Finally, section 6 concludes and discusses future works.

2 Application migration requirements and related works

Following [7, 1], we are interested on specifying migration requirements between pairs of data models defined as follows.

Definition 1 (*data model*). A data model M is a quadruplet $(\mathsf{D}, \mathsf{Q}, \mathsf{U}, [\![\,]\!]_M)$ 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 $\mathsf{u} \in \mathsf{U}$, a query statement $\mathsf{q} \in \mathsf{Q}$ and a database instance db $\in \mathsf{D}$, $[\![\mathrm{u(db)}]\!]_M$ yields a database instance, and $[\![\mathrm{q(db)}]\!]_M$ 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 successively introduce two levels of migration requirements: *upward compatibility* and *temporal transition 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 [1, 12].

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

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

□

Notice that in this latter expression, updates and queries are parameterized by an instant. This is because, in some temporal data models, the semantics of queries and updates depend on the instant at which they are issued.

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, updates include object creations and deletions, as well as updates to object properties. We will examine all these update operators in section 4, and define temporal extensions of them.

To illustrate upward compatibility, consider an ODMG compliant DBMS managing a database about document loans in a library. Upward compatibility states that if the ODMG DBMS is replaced by a temporal extension of it, the application programs accessing these data may be left intact. This implies that the set of database instances recognized by the extension is a super-set of those recognized by the original DBMS, and that the query and update statements have identical semantics in the original DBMS and in the temporal extension.

Now, suppose that once the legacy applications run on the temporal extension, it is decided that the history of the loans should be kept, but in such a way that legacy application programs may continue to run (at worst they should be recompiled), while new applications should perceive the property as being "historical". In the sequel we call this requirement *temporal transition support*.

While upward compatibility is achieved by adding new concepts and constructs to a model without modifying the existing ones, temporal transition support is more difficult to achieve. Indeed, [1] shows that almost none of the existing temporal extensions of SQL, including TSQL2, satisfy this requirement.

The same remark holds for existing object-oriented temporal extensions, and in particular for those concerning ODMG (e.g. TAU [8] and T_ODMG [2]). For example, consider a class Document with a property loaned_by. In TAU, 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 [14] (another ODMG temporal extension) does not exhibit this latter problem. However, in achieving temporal transition 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, leading to cumbersome query expressions. 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.

We propose 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 groups: those which view data under schema S, and those which view it under schema S'. Therefore, the problem of temporal transition support is seen as a particular case of schema evolution, and techniques developed in this context apply. More precisely, our approach to temporal transition support is based upon the notion of *bi-accessible temporal data model* defined as follows.

Definition 3 (*Bi-accessible temporal data model*). A bi-accessible temporal data model is a pair of data models ($\mathsf{M_S}$,$\mathsf{M_T}$), $\mathsf{M_S} = (\mathsf{D_S}, \mathsf{Q_S}, \mathsf{U_S}, [\![\,]\!]_{M_S})$ and $\mathsf{M_T} = (\mathsf{D_T}, \mathsf{Q_T}, \mathsf{U_T}, [\![\,]\!]_{M_T})$, such that:

1. $\mathsf{M_S}$ is a snapshot data model upward compatible with respect to temporal data model $\mathsf{M_T}$

2. for any $\mathsf{db_s} \in \mathsf{D_S}$, there exists a $\mathsf{db_t} \in \mathsf{D_T}$, such that, 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 $\mathsf{d_1}, \ldots \mathsf{d_n}, \mathsf{d_{n+1}}$:
$[\![\mathsf{q^{dn+1}(u_n^{dn}(} \ldots \mathsf{u_2^{d2}(u_1^{d1}(db_s))))}]\!]_{M_S} = [\![\mathsf{q^{dn+1}(u_n^{dn}(} \ldots \mathsf{u_2^{d2}(u_1^{d1}(db_t))))}]\!]_{M_T}$

□

The process of mapping $\mathsf{db_s}$ into $\mathsf{db_t}$ is usually termed *database adaptation* or *database conversion* in the schema evolution terminology. We elaborate on this issue in section 5 where we also introduce a view mechanism simulating bi-accessibility on top of the proposed ODMG extension.

3 A temporal extension of ODMG's object model

In this section, we overview the proposed extension of ODMG's model. First we briefly present the temporal datatypes. Next, we show how the basic abstractions of ODMG's model are temporally extended. For more details, the reader may refer to the bibliography on TEMPOS [6, 4], upon which this extension is based.

3.1 Temporal values and histories

TEMPOS is based upon a discrete, linear and bounded time model in which the time line is structured into *time units*. A time unit defines the precision at which time is observed in a particular context. Common units include *year*, *month* and *day*. The set of time units is extensible.

Based on this temporal structure, the TEMPOS model defines three basic temporal datatypes: *Instant*, *Duration*, and *Set of instants*. Intervals are viewed as particular cases of sets of instants.

The History Abstract DataType (ADT) models functions 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. Two selectors of the History ADT (respectively TDomain and SDomain) allow to retrieve these two components of a history.

A set of algebraic operators is defined on histories. These operators include restrictions, joins, extended set operators, grouping operators, aggregations and operators for reasoning about succession in time. In addition, a language for describing patterns of histories has been defined in [4].

3.2 Temporal support at the property level

In ODMG, a property is defined as an attribute or traversal path of a *relationship* attached to some class. For instance, possible properties of an Employee class include salary and department. As classes, properties may be instantiated and instances of properties are attached to objects. More precisely, an object is made up of a unique identifier and a collection of *property instances*. Each of these property instances has a *value* attached to it which may be *accessed* and *updated* through predefined methods defined over the Property interface.

In the proposed extension, a property is *temporal* if the successive values of its instances are recorded, or else *fleeting* if only the most recent value is recorded. When a property is temporal, each of its instances has a *history* attached to it, whose granularity is fixed by the *observation unit* of the property.

As in ODMG, a type is attached to a property. In the case of a fleeting property, the type models the domain of possible values of an instance of this property. In the case of a temporal property, it models the domain of the possible *structural values* of the histories attached to the instances of this property.

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 [11]. The corresponding properties are refered to as *valid-time* or *transaction-time* properties.

The *observation temporal domain* (or *observation domain* in short) of a temporal property instance, is the set of instants during which the property is observed for a given object.

As stated above, each temporal property instance has a history. The temporal domain of this history is equal to the observation domain of the property instance itself. Its structural values are either defined by some update, or derived from the values provided by the user by means of a *semantic assumption*.

More precisely, each temporal property instance has an *effective history*, corresponding to the input timestamped values attached to it. The effective history is contained in (but not necessarily equal to) the property instance's history. The difference between these two histories is called the *potential history* (i.e. the part of the history calculated using the semantic assumption). In the sequel, the temporal domain of a property instance's effective history is called its *effective temporal domain*, or *effective domain* in short.

We distinguish three particular semantic assumptions depending on the intended calculation mode of the potential history (see figure 1):

- *Discrete*: the structural values of the potential history are all equal to the neutral value of the structural type (e.g. `0` for integers, `nil` for objects) or to some *padding value* provided by the user. This is the case of the production of some product in a factory: the period of time during which the production is defined (its observation domain) may be known in advance (e.g. all weekdays). However, at some days, it may happen that there is no production (e.g. due to a strike), so that the effective history is undefined for those days.
- *Stepwise*: structural values are "stable" between two instants in the effective domain (e.g. a property instance modeling an employee's salary). As for 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 assumption does not provide one (e.g. if the smallest instant in the effective domain is different from the smallest instant in the observation domain).
- *Linearly interpolated*: this kind of interpolation applies only to numerically-valued properties. Between two "successive" instants in the effective history, the structural value varies linearly.

Other assumptions may be defined depending on the needs of applications, and the characteristics of the involved structural type.

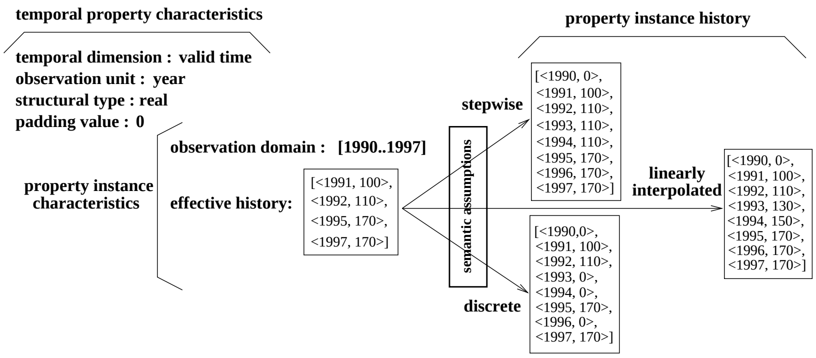

Fig. 1. effective history and semantic assumptions

3.3 Temporal support at the class extent level

As properties, classes may be fleeting or temporal. Declaring a class as temporal attaches a "lifespan" (called *observation domain*) to each object of the class. Thereby, it allows to determine the set of objects in the class extent which are observed at a given instant.

Conceptually, the observation domain of a temporal object, is the set of instants during which the information conveyed by this object is relevant to the applications and thus observed. The notion of "observation" may either be defined with respect to transaction-time, or to valid-time.

For instance, consider a class Product modeling the product types provided by a company. If the class is declared as temporal (either with respect to valid-time or transaction-time), then the observation domain could model the time when a particular product is produced. In addition, if the class is transaction-time, 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 produced in reality.

In accordance with ODMG, the *extent* of a class (whether fleeting or temporal) is the set of all instances of this class having been created and not deleted. In the case of temporal classes, we distinguish the *extent* of the class as defined above, from the *observed extent* at an instant i, which is the subset of the extent consisting of all objects whose observation domain contains i.

A given application may either manipulate the whole extent of a class, or the extent at some instant. For instance, a snapshot application accessing a temporal class, is likely to be interested only in the observed extent of the class at the current instant, as discussed in section 5.2.

4 Database updates and accesses

4.1 Temporal property instances' evolution

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 the property instance, the value given as parameter.

The set of updating and accessing operators on temporal property instances is more complex. These operators are classified depending on whether they are intended to modify the observation domain or the effective history, and depending on the temporal dimension to which they apply.

Evolution of transaction-time property instances Since transaction-time is intended to model the evolution of the database, the observation domain of transaction-time property instances evolve automatically with respect to the database time (i.e. the system clock). Conceptually, the current instant is added to the observation domain of a transaction-time property instance at each system clock tick. This automatic evolution of the observation 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 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 observation domain evolves with the system clock. Otherwise, it does not evolve at all. Operators turn_on and turn_off allow to switch between these two states.

An overloaded version of ODMG's set_value operator allows to update the effective history of a transaction-time property instance. Operation set_value(v) applied to a transaction-time property instance, replaces its effective history by a new history, identical to the old one except that it maps the current instant to value v. If needed, the growth status of the property instance is turned on.

Evolution of valid-time property instances Unlike transaction-time properties, the observation domain of a valid-time property instance does not evolve automatically with the system clock. Instead, an operator set_odomain is provided, which destructively replaces the observation domain of the property instance by the set of instants given as parameter. Since the observation domain should contain the effective domain, this operator may force some modifications on the effective history. For instance, if the constraint is violated after some update to the observation domain, the effective history is restricted to fit inside the new observation domain.

The primitive operator for updating the effective history of a temporal property instance is set_ehistory which replaces the effective history of the temporal property by the one given as parameter. The standard set_value operator is also supported. Given a valid-time property instant VTPI, VTPI.set_value(v) replaces VTPI's effective history by a new history, identical to the old one except that it maps the current instant to value v.

To enforce the inclusion constraint relating the effective domain of a temporal property instance and its observation domain, set_ehistory modifies, when necessary, the observation domain of the property instance to which it applies.

Accessing temporal property instances The primitive access operators for both valid-time and transaction-time property instances are get_ehistory and get_history. The former retrieves the effective history of the property instance. The latter builds a history from the observation domain and the effective history using the corresponding semantic assumption[1] as depicted in figure 1 page 6.

Operator get_value is also defined on temporal property instances: it retrieves the structural value of the property instance's history at the current instant.

4.2 Temporal objects' observation domain evolution

The observation domain of transaction-time objects evolves automatically with the system clock in a similar way as that of transaction-time property instances.

More precisely, each transaction-time object has a growth status, which may be On or Off. Conceptually, while a transaction-time object is On, the current instant is added to its observation domain at each system clock tick. Operators turn_on and turn_off allow to modify the value of the growth status of a transaction-time object. When an object is created, its growth status is On.

Regarding valid-time objects, the unique operator provided for updating the observation domain is set_odomain, which sets the observation domain to be the set of instants given as parameter.

4.3 Example

Consider a class Product with a valid-time stepwise attribute price, having structural type real. The valid-time observation domain of an object of class Product models the time when the product is produced (at the granularity of the day), while the observation domain of an instance of property price models the time when its price is defined. Table 1 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 or equal to it (resp. less than or equal).

5 Achieving temporal transition support

Whenever temporal support is incorporated into previously fleeting classes and/or properties, temporal transition support demands that applications may continue to access the database as if no temporal support had been added, or else, take into account this schema modification. This situation is a rather simple case of schema evolution. Our approach to handle it is similar to those described in [9, 10]. More precisely, three steps are followed. Firstly, the schema is modified. Next, the database is converted to fit the new schema (see section 5.1). Lastly, an updatable "snapshot" view of the database is introduced. Contrarily to [10] and other related works where the issue of schema evolution is studied in a broader setting, we do not employ a complex view mechanism. Instead, we propose an *ad hoc* view mechanism based on the update operators on properties defined in the previous section, and on the notion of *access mode* (see section 5.2).

[1] Transaction-time properties have a stepwise semantic assumption.

Scenario	A product is introduced on 1/4/98 with price 50
Operation	P = new Product; P.set_odomain([1/4/98..]); P.price.set_ehistory({⟨1/4/98, 50⟩})
Result	P.price.get_history() = {⟨[1/4/98..], 50⟩}
Scenario	On 1/6/98, the product's price raises to 60
Operation	P.price.set_ehistory(P.price.get_ehistory() $\cup_+$ {⟨1/6/98, 60⟩})
Result	P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..], 60⟩}
Scenario	The product is not longer produced starting from 6/8/98
Operation	P.set_odomain(P.get_odomain() ∩ [..5/8/98])
Result	P.get_odomain() = [1/4/98..5/8/98] P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..5/8/98], 60⟩}
Scenario	The product is produced again starting from 1/1/99; its price is not observed then
Operation	P.set_odomain(get_odomain(P) ∪ [1/1/99..])
Result	P.get_odomain() = {[1/4/98..5/8/98], [1/1/99..]} P.price.get_history() unchanged
Scenario	The product's price is observed starting from 1/2/99 and until 31/3/99. Its value at 1/2/99 is 80
Operation	P.price.set_odomain(P.price.get_odomain() ∪ [1/2/99..31/3/99]) P.price.set_ehistory(P.price.get_ehistory() $\cup_+$ {⟨1/2/99, 80⟩})
Result	P.get_odomain() unchanged P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..5/8/98], 60⟩, ⟨[1/2/99..31/3/99], 80⟩}

Table 1. Update scenario

5.1 Database instance adaptation

The algorithm below describes how a database instance is adapted to a schema modification which adds temporal support, i.e. which transforms fleeting properties or fleeting classes into temporal ones. This conversion operator should be applied to all existing objects in the database whenever the schema is modified[2]. Throughout the algorithm, some functions such as classOf and properties are used to retrieve metadata about the modified schema.

This algorithm puts forward the fact that temporal transition support is only applicable to stepwise-varying properties. This is because the idea behind temporal transition support is that when the "current" value of a temporal property is modified, the new current value assigned to it should remain constant until the next modification. Such a characteristic is inherent to stepwise properties, and *a fortiori*, to transaction-time properties since they evolve stepwisely.

Algorithm: *Object conversion operator*

```
Input and procedure 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 */
  O_copy: Object; /* variable used to store a copy of O */
```

[2] To avoid the complexity of this process on large databases, deferred conversion techniques may be applied. However, this issue is out of the scope of this paper

```
Procedure:
  O_copy := O.copy(); /* a copy of O is temporarily stored */
  Modify the structure of object O to fit the new schema;
  if classOf(O) in modified_classes then
    if (temporalDimension(classOf(O)) = transaction-time) then
      O.turn_on()
    else O.set_odomain([current_instant()..])
  for p in properties(classOf(O))
    if (p in modified_properties) {
      if (temporalDimension(p) = transaction-time) then
        O.p.turn_on();
      else if (semanticAssumption(p) = stepwise) then
        O.p.set_odomain([current_instant()..])
      O.p.set_value(O_copy.p.get_value())
    }
    else O.p = O_copy.p
```

5.2 Access modes

Usually, object-oriented programming and querying languages only provide one construct for accessing property instances, and one for updating them. For instance, in C++, the only way to access the value of an attribute is through the "dot" operator, whereas updating is performed through constructs of the form o.p = v. The proposed temporal ODMG extension 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 snapshot mode: temporal property instances are snapshot-valued and their value is defined by the structural value of their history at the current instant given by the system clock. In addition, any reference to the extent name of a temporal class 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 snapshot 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 section 4). Similarly, updates in this mode are handled by the set_value operator. In the temporal mode, get_history and set_ehistory are used instead.

In fact, the notion of access mode simulates a view mechanism: applications using the snapshot mode access a "snapshot" view of the database, while those using the temporal mode access the "temporal" view.

Different access modes may be attached to any two applications accessing the same database. As a result, the access mode is a parameter of each application session. The snapshot mode is the default. This design choice is crucial to ensure temporal transition support, since it entails that existing applications are automatically classified as "snapshot" during the migration process.

To illustrate the role of access modes from the querying viewpoint, consider a class Product whose extent is named TheProducts, and having two attributes trademark and price with types *string* and *real* respectively. Now suppose that at some time during the life of the application, the class Product as well as its attribute price are declared as temporal (trademark remains a fleeting property). Now consider the following OQL query:

```
select struct(t : b.trademark, p : b.price) from TheProducts as b
```

In the temporal mode, this query has type $bag{<}struct{<}t : string, p : History{<}real{>}{>}{>}$ and retrieves for each product type ever sold, the history of its prices. Conversely, if the snapshot mode is assumed, the query type is $bag{<}struct{<}t : string, p : real{>}{>}$ and it retrieves the trademarks of all currently sold product types with their corresponding prices.

6 Conclusion and future work

The main contributions of this paper include the precise formulation of requirements related to legacy code migration toward temporal DBMS, and a proposal of an ODMG temporal extension integrating these requirements.

The formulated requirements are *upward compatibility* and *temporal transition support*. The former states that a database may be transparently migrated from a DBMS to a temporal extension of it. The latter allows legacy code to remain usable when temporal support is added to some components of a snapshot database schema, and may therefore be seen as a schema evolution issue. These notions were first proposed in the relational framework in [1, 12].

The proposed ODMG temporal extension is based on the TEMPOS framework [6, 4]. It includes a set of ADT modeling temporal values and histories as well as temporal extensions of the main ODMG components. Temporal transition support is ensured by separating the notion of *temporal property* from that of *history* and by dividing applications into snapshot and temporal: a temporal property may have a historical value in the context of a temporal application and a snapshot value in a non-temporal one. Update primitives on temporal properties are defined in such a way that updates performed by temporal and non-temporal applications are mutually consistent.

The proposal is being implemented on top of the O_2 DBMS. Up to now, we have implemented the temporal types as a library of classes. The OQL extension on the other hand, has been implemented by means of a pre-processor. This pre-processor recognizes specific statements for setting the access mode of a query, thereby integrating the notion of bi-accessibility. Current efforts aim at integrating this notion into each of the programming language bindings defined by the ODMG standard (i.e. SmallTalk, C++ and Java bindings).

Having formulated temporal transition support as a schema evolution problem, it is straightforward to identify more sophisticated, yet useful alternative definitions of this notion. For instance, in our definition, only a single temporal schema is managed at a time. This works fine when the process of converting non-temporal data into temporal one is carried out in a single step. However, more complex situations may arise. For instance, suppose that a non-temporal schema S is modified into a temporal schema S', and that subsequently S' is modified to add temporal support to some of its non-temporal components, yielding schema S". In our approach, two views are managed at the end of this process: one with schema S and another with schema S". Hence, applications developed under schema S' are not supported! We believe that generalizing our approach to handle this kind of situations is an interesting perspective.

References

1. J. Bair, M. Böhlen, C.S. Jensen, and R.T. Snodgrass. Notions of upward compatibility of temporal query languages. Technical Report TR-6, Time Center, 1997.
2. 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.
3. 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.
4. 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.
5. O. Etzion, S. Jajodia, and S.M. Sripada, editors. *Temporal Databases: Research and Practice.* Springer Verlag, LNCS 1399, 1998.
6. M.-C. Fauvet, J.-F. Canavaggio, and P.-C. Scholl. Modelling histories in object DBMS. In *proc. of the 8th Int. Conference on Database and Expert Systems Applications (DEXA)*, Toulouse (France), September 1997. Springer Verlag. LNCS 1308.
7. S. K. Gadia and S. S. Nair. Temporal databases: a prelude to parametric data. In Tansel et al. [13].
8. I. Kakoudakis and B. Theodoulidis. The TAU Temporal Object Model. Technical Report TR-96-4, TimeLab, University of Manchester (UMIST), 1996.
9. R. Peters and M.T. Ozsu. An axiomatic model of dynamic schema evolution in objectbase management systems. *ACM TODS*, 22(1), 1997.
10. Y. G. Ra and E. A. Rundensteiner. A transparent schema evolution based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, pages 600–624, September 1997.
11. R. T. Snodgrass and I. Ahn. A taxonomy of time in databases. In *proc. of ACM SIGMOD*, May 1985.
12. R.T. Snodgrass, M. Böhlen, C. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In Etzion et al. [5].
13. A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodggrass, editors. *Temporal Databases.* The Benjamins/Cummings Publishing Company, 1993.
14. TOOBIS ESPRIT Project. TODM, specification and design. Deliverable T31TR.1, MATRA CAP SYSTEMES – O2 Technology, December 1996.