# Extending the ODMG Object Model with Time

*Elisa Bertino* [1] *Elena Ferrari* [1] *Giovanna Guerrini* [2] *Isabella Merlo* [2]

[1] Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39/41 - I20135 Milano, Italy
{bertino,ferrarie}@dsi.unimi.it

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso 35 - I16146 Genova, Italy
{guerrini,merloisa}@disi.unige.it

**Abstract.** Although many temporal extensions of the relational data model have been proposed, there is no comparable amount of work in the context of object-oriented data models. Moreover, extensions to the relational model have been proposed in the framework of SQL standards, whereas no attempts have been made to extend the standard for object-oriented databases, defined by ODMG. This paper presents $T$_ODMG, a temporal extension of the ODMG-93 standard data model. The main contributions of this work are, thus, the formalization of the ODMG standard data model and its extension with time. Another contribution of this work is the investigation, on a formal basis, of the main issues arising from the introduction of time in an object-oriented model.

## 1 Introduction

Temporal databases [16] provide capabilities for efficiently managing not only the current values of data, but also the entire history of data over time. Several database applications, such as the ones in the medical and scientific domains or the ones in the business and financial context, require support for historical data. In most cases, data of interest to those applications are meaningful only if related to time.

Most of the research and development efforts in the area of temporal databases have been carried out in the context of the relational model. A temporal extension, known as TSQL-2 [13], of the SQL-92 relational database standard, has been proposed and issues related to temporal relational data models and query languages have been extensively investigated from the theoretical point of view [17].

By contrast, research on temporal object-oriented databases is still in an early stage. Although various object-oriented temporal models have been proposed [12], there is no amount of work comparable to the work reported for the relational model. One of the main reasons for the delay in the development of temporal object-oriented data models is that, until recently, there was no standard for object-oriented databases. Recently, a proposal for an object database standard has been formulated by ODMG [4]. The ODMG standard includes a reference object model (ODMG Object Model), an object definition language (ODL) and an object query language (OQL). The definition of such a standard represents an important step toward the widespread diffusion of object-oriented DBMSs. Indeed, most of the success of relational databases, as well as of its temporal extensions, is undoubtly due to the existence of a well-accepted standard. The acceptance of the SQL standard ensures a high degree of portability and interoperability between systems, simplifies learning new relational DBMSs, and represents a wide endorsement of the relational approach.

For all the above reasons, in this paper we propose $T$_ODMG, a temporal extension of the ODMG standard object data model. Thus, the goal of our work is not to develop a new temporal object-oriented data model. Rather our goal is to incorporate temporal features into the ODMG object model. To the best of our knowledge, this is the first extensive attempt of complementing the ODMG data model with temporal capabilities.

The ODMG data model provides a very rich set of modeling primitives. Objects are characterized by a state and a behavior. The state of an object is represented by the values of its properties. Properties can be either attributes, like in most object-oriented data models, or relationships. Relationships are declared between two object types and induce a pair of traversal paths between the two types. Moreover, the ODMG data model supports the notion of key, that is, a set of properties whose values uniquely identify each object within a class extent.

The development of a temporal extension of a semantically rich data model, like the one proposed by ODMG, entails several interesting issues that we address in this paper. First the rich set of data types provided by ODMG has been complemented with a set of temporal types, which allows to handle in a uniform way temporal and non temporal domains. Then, we have added the temporal dimension to both object attributes and relationships. A relevant feature provided by $T$_ODMG is the support for temporal, immutable and static object properties. A temporal property is a property whose value may change over time, and whose values at different times are recorded in the database. An immutable property is a property whose value cannot be modified during the object lifetime, whereas a static property is a property whose value can change over time, but whose past values are not meaningful, and are thus not stored in the database. In addition, we have investigated how the notions of extent, key and object consistency should be modified to deal with the temporal dimension.

Another important property of the $T$_ODMG data model we propose is that it extends the ODMG data model in an "upward compatible" way, that is, non-temporal and temporal data coexist in an integrated way. This is very important if applications should migrate to a temporal conext.

A further contribution of this paper is that the addition of time in the ODMG standard data model has been carried out on a formal basis. This has required to first provide a formalization of the main components of the ODMG model, which was not provided in [4].

The temporal object model presented in this paper is based on the $T$_Chimera formal data model proposed by us in [2]. Such model represents the starting point of our current work. However the work reported in this paper substantially extends the previous work. First ODMG is characterized by a richer set of types with respect to the $T$_Chimera one. Second, ODMG supports the notions of relationship and key which are not provided by $T$_Chimera. Finally, the notions of temporal types and the distinction among temporal, static, immutable attributes, that are inherited from $T$_Chimera, obviously need to be revisited in the context of the $T$_ODMG model.

This paper is organized as follows. Section 2 summarizes the main features of the ODMG standard. Section 3 introduces types and values, whereas Sections 4 and 5 deal with classes and objects, respectively. Section 6 discusses existing proposals of temporal object-oriented data models. Finally, Section 7 concludes the paper.

## 2 The object database standard: ODMG-93

In what follows we summarize the main features of the ODMG data model. We refer the interested reader to [4] for the description of the object definition language (ODL) and the object query language (OQL).

The basic modeling concepts in the ODMG data model are the concepts of *object* and *literal*. The main difference between objects and literals is that each object has a unique identifier (*oid*), whereas a literal has no identifier. Literals are sometimes described as being constant or *immutable* since the value of a literal cannot change. By contrast, objects are described as being *mutable*. Changing the values of the attributes of an object, or the relationships in which it is involved, does not change the identity of the object.

Objects and literals can be categorized according to their *types*. Object types can be partitioned into two main groups: atomic object types and collection object types.

Atomic object types are user-defined types (e.g., `Person`, `Employee`). Collection object types represent set, bag, list and array objects. Literal types can be partitioned into three main groups: atomic literal types, collection literal types and structured literal types. Atomic literal types are numbers, characters and so on. Collection literal types represent set, bag, list and array literals. Structured literal types have a fixed number of elements, each of which has a variable name and can contain either a literal value or an object. Moreover ODMG supports user-defined structures implementing records.

The formalization of object and literal types and the differences between collection object types and collection literal types are presented in Subsection 3.1. An object is sometimes referred to as an *instance* of its type.

An object in ODMG is characterized by a *state* and a *behavior*. The state of an object is defined by the values of its *properties*. Properties can be either *attributes* of the object itself or *relationships* among the object and one or more other objects. Typically the value of an object property can change over time. The behavior of an object is defined by the set of *operations* that can be executed on or by the object. All objects of a given type have the same set of properties and the same set of defined operations.

There are two aspects in the definition of a type. A type has an *interface* specification and one or more *implementation* specifications. The interface defines the *external characteristics* of an instance of the type. That is the operations that can be invoked on the object and the state variables whose values can be accessed. By contrast, a type implementation defines the *internal aspects* of the instances of the type.

The distinction between interface and implementation is important, since the separation between these two is the approach according to which ODMG supports encapsulation. Throughout the paper we assume that a type has a single implementation specification. Therefore the terms object type, interface and class are used as synonymous. More precisely, we distinguish between object types and literal types, object interfaces and literal interfaces, while the term class is used only when dealing with objects. Moreover, implementation details are not relevant from a modeling point of view. Thus, we focus on the interface specification, disregarding the implementation specification of a type.

The attribute declarations in an interface define the abstract state of a type. An attribute value is either a literal or an object identifier. Relationships are defined between object types. The ODMG object model supports only binary relationships, i.e., relationships between two types, each of which must have instances that can be referenced by object identifiers. Therefore literal types cannot participate in relationships (since they do not have object identifiers). Relationships in the ODMG object model are similar to relationships in the entity-relationship data model [6]. A binary relationship may be one-to-one, one-to-many, or many-to-many, depending on how many instances of each type participate in the relationship. For instance, *marriage* is an example of one-to-one relationship between two instances of type `Person`. A woman can have a one-to-many *mother_of* relationship with many children. Teachers and students typically participate in many-to-many relationships.

A relationship is implicitly defined by declaring *traversal paths* that enable applications to use the logical connections between the objects participating in the relationship. For each relationship two traversal paths are declared, one for each direction of traversal of the binary relationship. For instance, the relationship between a professor and the courses he/she teaches generates two traversal paths, since a professor *teaches* courses and a course *is taught by* a professor. The `teaches` traversal path is defined in the interface declaration of the `Professor` type. The `is_taught_by` traversal path is defined in the interface declaration of the `Course` type. The fact that the `teaches` and `is_taught_by` traversal paths apply to the same relationship is indicated by an `inverse` clause in both the traversal path declarations.

One-to-many and many-to-many relationships can also be implemented using bags, lists and arrays.

Besides the attribute and relationship properties, the other characteristic of a type is its behavior, which is specified as a set of *operation signatures* (*method* *signatures*). Each signature defines the name of an operation, the name and type of each of its arguments, the types of the value(s) returned.[1] Each operation is associated with a single type and its name must be unique within the corresponding type definition. However, operations with the same name can be defined for different types. The names of these operations are said to be *overloaded*. When an operation is invoked using an overloaded name, a specific operation must be selected for execution. This selection is called *operation dispatching*. An operation may have side effects. Some operations may return no value. The ODMG object model does not include formal specification of the semantics of operations; it is highly implementation dependent. Finally *extents*, *keys* and *supertype* information can be optionally associated with a type.

The extent of a type is the set of all instances of the type within a particular database. If an object is an instance of a type `t`, than it necessarily belongs to the extent of `t`. If type `t` is a subtype of type `t'`, then the extent of `t` is a subset of the extent of `t'`.

In some cases, instances of a type can be uniquely identified by the values they carry for some property or set of properties. These identifying properties are called *keys*. The scope of uniqueness is the extent of the type, thus a type must have an extent in order to have a key.

Similar to most object models, also the ODMG object model includes inheritance based type-subtype relationships. Supertype information is one of the characteristics of types, together with extent naming and key specifications. The type-subtype relationships are commonly represented by oriented graphs; each node represents a type and there exists an arc from a node `t` to a node `t'` if `t'` is the *supertype* of `t` (*subtype*). The type/subtype relationship is sometimes called an *is-a* relationship, or simply an *ISA* relationship, or *generalization-specialization* relationship. The supertype is the more general type; the subtype is the more specialized one.

## 3 Types and values

In this section we formally introduce $T$_ODMG types and values. First, we give a formalization of ODMG types. Then, we extend the ODMG types with temporal types. The resulting model is $T$_ODMG, the temporal extension of the ODMG model. Temporal structured literal types are described in detail. Finally, we define $T$_ODMG legal values. In the following we refer to Table 1 that summarizes the functions used in defining the model. For each function the table reports the name, the signature and the output.[2]

In the following we denote with $\mathcal{OI}$ a set of object identifiers, with $\mathcal{CI}$ a set of class identifiers, that is, interface names, with $\mathcal{AN}$ a set of attribute names, with $\mathcal{RN}$ a set of relationship names and with $\mathcal{MN}$ a set of method names.

[1] Moreover each signature defines the names of *exceptions* (error conditions) the operation can raise. In this context we do not consider exceptions.

[2] The meaning of each function will be clarified in the remainder of the discussion.

| Name | Signature | Output |
| --- | --- | --- |
| $\pi$ | $\mathcal{CI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$ | extent of a class at a given instant |
| $pe\_set$ | $\mathcal{CI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$ | proper extent of a class at a given instant |
| $type$ | $\mathcal{CI} \rightarrow \mathcal{T} \times \mathcal{T}$ | structural property type of a class |
| $h\_type$ | $\mathcal{CI} \rightarrow \mathcal{T} \times \mathcal{T}$ | historical property type of a class |
| $s\_type$ | $\mathcal{CI} \rightarrow \mathcal{T} \times \mathcal{T}$ | static property type of a class |
| $h\_state$ | $\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V} \times \mathcal{V}$ | historical value of an object |
| $s\_state$ | $\mathcal{OI} \rightarrow \mathcal{V} \times \mathcal{V}$ | static value of an object |
| $o\_lifespan$ | $\mathcal{OI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$ | lifespan of an object |
| $c\_lifespan$ | $\mathcal{OI} \times \mathcal{CI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$ | lifespan of an object as a member of a class |

**Table 1.** Functions employed in defining the model

### 3.1 ODMG types

As we mentioned in Section 2, objects and literals can be categorized according to their types. First we give the formal definition of object types. In the remainder, ODMG$\mathcal{T}$ denotes the set of ODMG types. It is formally defined later on in this subsection.

**Definition 1** (*Object Types*). The set of ODMG object types $\mathcal{OT}$ is defined as follows.

$$\mathcal{OT} = \mathcal{AOT} \cup \mathcal{AOST} \cup \mathcal{COT}$$

where:

- $\mathcal{AOT}$ is the set of ODMG *atomic object types*; it is defined as the set of class identifiers $\mathcal{CI}$;
- $\mathcal{AOST}$ is the set of ODMG *atomic object system types*; it is defined as the following set {`Object`, `Collection`, `Set`, `Bag`, `List`, `Array`};
- $\mathcal{COT}$ is the set of ODMG *collection object types*; it is defined as the following set $\mathcal{COT}$ = {`O_Set<t>` | `t` ∈ ODMG$\mathcal{T}$} ∪ {`O_Bag<t>` | `t` ∈ ODMG$\mathcal{T}$} ∪ {`O_List<t>` | `t` ∈ ODMG$\mathcal{T}$} ∪ {`O_Array<t>` | `t` ∈ ODMG$\mathcal{T}$}.[3]

□

As Definition 1 states, the ODMG object types can be partitioned into three sets. The set $\mathcal{AOT}$ denotes the simple user-defined interfaces. The set $\mathcal{AOST}$ denotes a set of predefined system types. Finally, the set $\mathcal{COT}$ denotes the types of objects of type set, bag, etc. The direct supertype of each collection object is

[3] Note that in [4] the notation for collection object types is different. For example, in case of sets `Set<t>` is used to denote both collection literal and collection object set types. We have chosen, for sake of clarity, to distinguish collection object and literal types. Therefore we use `O_Set<t>` to denote the collection object type for sets and `Set<t>` to denote the corresponding collection literal type.

the relative atomic object system type (e.g., `Set` is the direct supertype of each `O_Set<t>` such that $\mathtt{t} \in \mathrm{ODMG}\mathcal{T}$). ODMG also provides collection literal types to define sets, bags, lists and arrays (Definition 2).

Note that a name in $\mathcal{CI}$ can be associated with a collection object type. For instance, the name `EmployeeSet` can be associated with the collection object type `O_Set<Employee>`. Names are associated with collection object types for an easy reference, there is, however, the complete substitutability of a collection object type and the name assigned to it.

The following definition formalizes ODMG literal types.

**Definition 2** (*Literal Types*). The set of ODMG literal types $\mathcal{LT}$ is defined as follows.

$$\mathcal{LT} = \mathcal{ALT} \cup \mathcal{CLT} \cup \mathcal{SLT} \cup \mathcal{ST}$$

where:

- $\mathcal{ALT}$ is the set of ODMG *atomic literal types*; it is defined as the following set $\mathcal{ALT} = \{$`Long`, `Short`, `Unsigned long`, `Unsigned short`, `Float`, `Double`, `Boolean`, `Octet`, `Char`, `String`, `Enum`$\}$;
- $\mathcal{CLT}$ is the set of ODMG *collection literal types*; it is defined as the following set $\mathcal{CLT} = \{\mathtt{Set{<}t{>}} \mid \mathtt{t} \in \mathrm{ODMG}\mathcal{T}\} \cup \{\mathtt{Bag{<}t{>}} \mid \mathtt{t} \in \mathrm{ODMG}\mathcal{T}\} \cup \{\mathtt{List{<}t{>}} \mid \mathtt{t} \in \mathrm{ODMG}\mathcal{T}\} \cup \{\mathtt{Array{<}t{>}} \mid \mathtt{t} \in \mathrm{ODMG}\mathcal{T}\}$;
- $\mathcal{SLT}$ is the set of ODMG *structured literal types*; it is defined as the following set $\mathcal{SLT} = \{$`Date`, `Interval`, `Time`, `TimeStamp`$\}$;[4]
- $\mathcal{ST}$ is the set of ODMG *struct literal types*; it is defined as the following set $\mathcal{ST} = \{\mathtt{Struct}\{\mathtt{t}_1\ p_1; \ldots; \mathtt{t}_n\ p_n\ \} \mid \mathtt{t}_i \in \mathrm{ODMG}\mathcal{T}, p_i \in \mathcal{AN} \cup \mathcal{RN}, i \in [1, n]\}$.

□

An important difference between collection object types and collection literal types is that instances of collection object types have identifiers while instances of collection literal types do not have identifiers. For more details on such features of the ODMG model we refer the reader to [4].

The set of ODMG types $\mathrm{ODMG}\mathcal{T}$ is defined as the union of the set of ODMG object types $\mathcal{OT}$ and the set of ODMG literal types $\mathcal{LT}$. Therefore:

$$\mathrm{ODMG}\mathcal{T} = \mathcal{OT} \cup \mathcal{LT}.$$

We also recall that in the ODMG data model the type `any` stands for any type $\mathtt{t} \in \mathrm{ODMG}\mathcal{T}$, and that `void` stands for "no values" and it is used for side effect operations which return no values.

---

[4] The interfaces of these types can be found in [4]. In the following we only specify the interfaces we are interested in, since they are relevant for our work.

### 3.2 *T*_ODMG types

*T*_ODMG extends the set of ODMG types with a collection of *temporal types*, introduced to handle in a uniform way temporal and non temporal domains. For each ODMG type `t`, a corresponding temporal type is defined through a special constructor *temporal*. Intuitively, instances of type *temporal*(`t`) are partial functions from instances of type `TimeStamp`[5] to instances of type `t`. We elaborate on this informal definition in the following section.

**Definition 3** (*T*_ODMG *Temporal Types*). The set of *T*_ODMG temporal types ODMG$\mathcal{TT}$ is defined as follows.

$$\mathrm{ODMG}\mathcal{TT} = \{temporal(\mathtt{t}) \mid \mathtt{t} \in \mathrm{ODMG}\mathcal{T}\}.$$

□

In *T*_ODMG, temporal types can be used in the definition of collection object, literal and struct types, as stated by the following definitions. In the remainder, $\mathcal{T}$ denotes the set of *T*_ODMG types.

**Definition 4** (*T*_ODMG *Object Types*). The set of *T*_ODMG object types $\mathcal{OTT}$ is defined as follows.

$$\mathcal{OTT} = \mathcal{AOT} \cup \mathcal{AOST} \cup \mathcal{COTT}$$

where:

- $\mathcal{AOT}$ is the set of ODMG *atomic object types*, as specified by Definition 1;
- $\mathcal{AOST}$ is the set of ODMG *atomic object system types*, as specified by Definition 1;
- $\mathcal{COTT}$ is the set of *T*_ODMG *collection object types*; it is defined as the following set $\mathcal{COTT} = \{\mathtt{O\_Set<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{O\_Bag<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{O\_List<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{O\_Array<t>} \mid \mathtt{t} \in \mathcal{T}\}$.

□

**Definition 5** (*T_ODMG Literal Types*). The set of *T*_ODMG literal types $\mathcal{LTT}$ is defined as follows.

$$\mathcal{LTT} = \mathcal{ALT} \cup \mathcal{CLTT} \cup \mathcal{SLTT} \cup \mathcal{STT}$$

where:

- $\mathcal{ALT}$ is the set of ODMG *atomic literal types*, as specified by Definition 2;
- $\mathcal{CLTT}$ is the set of *T*_ODMG *collection literal types*; it is defined as the following set $\mathcal{CLTT} = \{\mathtt{Set<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{Bag<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{List<t>} \mid \mathtt{t} \in \mathcal{T}\} \cup \{\mathtt{Array<t>} \mid \mathtt{t} \in \mathcal{T}\}$;

[5] Type `TimeStamp` will be described in Subsection 3.3.

- $\mathcal{SLTT}$ is the set of $T$_ODMG *structured literal types*; it is defined as the following set $\mathcal{SLTT} = \mathcal{SLT} \cup \{\texttt{TimeInterval}\}$;
- $\mathcal{STT}$ is the set of $T$_ODMG *struct literal types*; it is defined as the following set $\mathcal{STT} = \{\texttt{Struct}\{\texttt{t}_1\ p_1; \ldots; \texttt{t}_n\ p_n\ \} |\ \texttt{t}_i \in \mathcal{T},\ a_i \in \mathcal{AN},\ i \in [1, n]\ \}$.

□

Now we can define the set of $T$_ODMG types $\mathcal{T}$.

**Definition 6** (*T_ODMG Types*). The set of $T$_ODMG types $\mathcal{T}$ is defined as follows.

$$\mathcal{T} = \mathrm{ODMG}\mathcal{TT} \cup \mathcal{OTT} \cup \mathcal{LTT}.$$

□

In the remainder of the discussion we use ODMG$\mathcal{ST}$ to denote the set of static, that is non temporal, $T$_ODMG types. Formally , ODMG$\mathcal{ST} = \mathcal{T} \setminus$ ODMG$\mathcal{TT}$.

### 3.3 Temporal structured literals

In what follows we briefly discuss the interfaces specified for `TimeStamp`, defined as part of the ODMG standard [4], and `TimeInterval`, the new structured literal we propose. For sake of simplicity, in introducing the following interfaces we do not deal with different time zones.

As we will see in Section 6, most models support a linear discrete time structure and a single time dimension, that is, valid time. ODMG structured types provide data structures for managing time. Thus, to adhere to the ODMG model, we have chosen to support time features using the ODMG existing data structures. We will see in Subsection 3.4 that this is conceptually equivalent to supporting a linear discrete time structure.

The structured type `TimeStamp`, whose interface is described in Figure 1, provides the time granularity of our model. In what follows a time instant $t$ is therefore represented as a pair $\langle date, time \rangle$, where *date* is a value of type `Date` that is, a year, a month and a day, whereas *time* is a value of type `Time`, that is, a particular time instant in such date, i.e., hour, minutes and seconds. In the following if `T` has type `TimeStamp` then `T.date` denotes the date and `T.time` denotes the time instant in the day, moreover `T.date.year = T.year` and so on. We do not present here the interface specifications of `Date` and `Time` for lack of space; we refer the interested reader to [4].

Note that we have chosen to model time instants as pairs $\langle date, time \rangle$. We could alternatively have chosen to employ dates as our time granularity.

In order to support time intervals, we introduce a new type `TimeInterval`, whose interface is described in Figure 2. We do not use the type interface `Interval` supplied by ODMG for intervals. Such interface does not implement our abstract notion of interval as a a set of consecutive time instants between two given instants, that is, two values of type `TimeStamp`. Rather it is only used to perform some operations on `Time` and `TimeStamp` objects. The type `TimeInterval` is

```
interface TimeStamp {
   typedef Unsigned short ushort;
   Date date();
   Time time();
   ushort year();
   ushort month();
   ushort day();
   ushort hour();
   ushort minute();
   float second();

   TimeStamp plus(in Interval an_interval);
   TimeStamp minus(in Interval an_interval);

   Boolean is_equal(in TimeStamp t);
   Boolean is_greater(in TimeStamp t);
   Boolean is_greater_or_equal(in TimeStamp t);
   Boolean is_less(in TimeStamp t);
   Boolean is_less_or_equal(in TimeStamp t);
   Boolean is_between(in TimeStamp a_t, in TimeStamp b_t);
   };
```

**Fig. 1.** `TimeStamp` interface

```
interface TimeInterval {
   TimeStamp start();
   TimeStamp end();

   TimeInterval equal(in TimeInterval I);
   Set<TimeInterval> union(in Set<TimeInterval> SI,in TimeInterval I);
   TimeInterval intersect(in TimeInterval I);

   Boolean inclusion(in TimeInterval I);
   Boolean in_interval(in TimeStamp t);
   };
```

**Fig. 2.** `TimeInterval` interface

characterized by two elements `start` and `end`, both of type `TimeStamp`, denoting the starting and ending time of the interval, respectively.

In order to simplify the notation, we use symbols, such as "=", "$\leq$", with the obvious meaning instead of operation names. For example the symbol "=" is used to denote the equality between two structured literals; this implies that the corresponding operation, `equal`, is implemented for the type. Other examples are "$\leq$" for `is_less_or_equal` between values of type `TimeStamp`, "$\cup$" for `union` between time intervals etc.

Let `I` be a variable of type `TimeInterval` such that $\texttt{I.start} = \langle date_1, time_1 \rangle$ and $\texttt{I.end} = \langle date_2, time_2 \rangle$. Thus `I` denotes the set of consecutive time instants, including all time instants between $t_1$, that is the time instant identified by $\langle date_1, time_1 \rangle$, and $t_2$, that is the time instant identified by $\langle date_2, time_2 \rangle$, $t_1$ and $t_2$ included. The constraint imposed on objects of type `TimeInterval` is that $\texttt{I.start} \leq \texttt{I.end}$. A single time instant can be represented as a time interval `I` such that $\texttt{I.start} = \texttt{I.end}$, whereas if $\texttt{I.start} = \texttt{I.end} = \texttt{nil}$, `I` denotes the null interval. The operators `equal` ($=$), `intersect` ($\cap$), `inclusion` ($\subseteq$) have the well-known semantics of interval operations. Moreover, function `in_interval` ($\in$) takes as arguments an instant $t$ and an interval `I` and returns true if $t$ is one of the instants denoted by interval `I`. The `union` ($\cup$) operation takes as input an interval and a set of intervals and returns the set of intervals consisting of the given set plus the given interval. In the following, we often use a set of disjoint intervals $I = \{\texttt{I}_1, \ldots, \texttt{I}_n\}$ as a compact notation for the set of time instants included in these intervals.

### 3.4 *T*_ODMG values

For each predefined atomic literal type $\texttt{t} \in \mathcal{ALT}$, we postulate the existence of a non-empty set of values, denoted as $dom(\texttt{t})$. For instance, the domain of the atomic literal type `Boolean` consists of the two classical boolean values `true` and `false`. The domain of type `TimeStamp` is the set $\mathcal{TIME}$, which is a set of pairs $\langle date, time \rangle$, where *date* is a value of type `Date` and *time* is a value of type `Time`. We can say that $\mathcal{TIME}$ represents the sequence of time instants starting from the first time instant considered in the system until *now*, where *now* is the timestamp corresponding to the current date and hour according to the system clock. Such set is isomorphic to the set of natural numbers $\mathbb{N}$.

It is easy to verify that the set of values of type `Date` is isomorphic to a subset of $\mathbb{N}$ (in the following we denote with $D$ this subset), and that the set of values of type `Time` is obviously a finite set (denoted in the following with $S$).[6] Therefore $\mathcal{TIME}$ is isomorphic to $D \times S$ and, since $D \times S$ is isomorphic to $\mathbb{N}$, $\mathcal{TIME}$ is then isomorphic to $\mathbb{N}$. Thus, we assume time to be discrete.

Of course, since each pair belonging to $\mathcal{TIME}$ is an instance of the ODMG type `TimeStamp`, the operations defined for type `TimeStamp` can be applied on it. Moreover the operations of the type `Date` can be applied to the first component of the pair, whereas the operations of the type `Time` can be applied to the

[6] The number of seconds in a day is a finite number.

second component of the pair. Moreover values of type `TimeInterval` are pairs where each component is of type `TimeStamp`. Then, the set of values of type `TimeInterval` is the cartesian product $\mathcal{TIME} \times \mathcal{TIME}$.

Note that in $T$_ODMG oids in $\mathcal{OI}$ are handled as values. Thus, an object identifier $i$ is a value of an object type in $\mathcal{CI}$. We consider as legal values for an object type $\mathtt{t} \in \mathcal{CI}$ all the oids of objects belonging to the extent of `t` both as instances or as proper instances.[7] The set of objects instances of a class changes dynamically over time. Thus, to define the extension, that is, the set of legal values for $T$_ODMG object types, we introduce a function $\pi$: $\mathcal{CI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$, assigning an extent to each object type, for each instant $t$. For each $\mathtt{t} \in \mathcal{CI}$, $\pi(\mathtt{t}, t)$ is the set of the identifiers of objects that, at time $t$, belong to the extent of `t` as instances. To emphasize the fact that the interpretation of a type can only be given fixing a time instant $t$, we denote the set of legal values for type `t` at time $t$ as $[\![\mathtt{t}]\!]_t$.

The following definition states the set of legal values for each $T$_ODMG type, denoted with $\mathcal{V}$.

**Definition 7** (*Type Legal Values*). $[\![\mathtt{t}]\!]_t$ denotes the extension of type $\mathtt{t} \in \mathcal{T}$ at time $t$.

- $nil \in [\![\mathtt{t}]\!]_t, \forall \mathtt{t} \in \mathcal{T}$;
- $[\![\mathtt{t}]\!]_t = dom(\mathtt{t}), \forall \mathtt{t} \in \mathcal{ALT}$;
- $[\![\mathtt{TimeStamp}]\!]_t = \mathcal{TIME}$;
- $[\![\mathtt{TimeInterval}]\!]_t = \mathcal{TIME} \times \mathcal{TIME}$;[8]
- $[\![\mathtt{t}]\!]_t = \pi(\mathtt{t}, t), \forall \mathtt{t} \in \mathcal{OTT}$;
- $[\![\ \mathtt{Set<t>}\ ]\!]_t = 2^{[\![\mathtt{t}]\!]_t}$;
- $[\![\ \mathtt{Bag<t>}\ ]\!]_t = multiset([\![\mathtt{t}]\!]_t)$;[9]
- $[\![\ \mathtt{List<t>}\ ]\!]_t = \{[v_1, \ldots, v_n] \mid n \geq 0, v_i \in [\![\mathtt{t}]\!]_t, \forall i \in [1, n]\}$;
- $[\![\ \mathtt{Array<t>}\ ]\!]_t = \{[[1, v_1], \ldots, [n, v_n]] \mid n \geq 0, v_i \in [\![\mathtt{t}]\!]_t, \forall i \in [1, n]\}$;
- $[\![\ \mathtt{Struct}\{\mathtt{t}_1\ p_1; \ldots; \mathtt{t}_n\ p_n\ \}]\!]_t = \{\langle v_1 p_1, \ldots, v_n p_n\rangle \mid p_i \in \mathcal{AN} \cup \mathcal{RN},\ v_i \in [\![\mathtt{t}_i]\!]_t, \forall i \in [1, n]\}$;
- $[\![temporal(\mathtt{t})]\!]_t = \{f \mid f : \mathcal{TIME} \rightarrow \bigcup_{t' \in \mathcal{TIME}} [\![\mathtt{t}]\!]_{t'}$ is a partial function such that $\forall t'$ if $f(t')$ is defined then $f(t') \in [\![\mathtt{t}]\!]_{t'}\}$. □

Given an instant $t$ the extensions of predefined basic value types are the elements of their corresponding domains, the extensions of classes are their explicit

[7] According to the usual terminology, an object is a *proper instance* of a class $c$, if $c$ is the most specific class, in the inheritance hierarchy, to which the object belongs to. If an object is a proper instance of a class it is also an *instance* of all the superclasses of $c$.

[8] We do not specify the extensions of all structured literals, `Date`, `Time` and so on, since they are the obvious set of values.

[9] As specified in the definition of the model [4], `Bag` denotes unordered collections of elements that may contain duplicates. In the scientific literature, such data structures are sometimes referred to as *multisets*, thus we denote as $multiset([\![\mathtt{t}]\!]_t)$ the possible multisets whose elements are of type `t`.

extents at time $t$, whereas the set of legal values of the collection literal types are defined recursively in terms of the legal values of their component types. The extension of a temporal type $temporal(\mathtt{t})$ is the set of partial functions from $\mathcal{TIME}$ (i.e, the set of legal values for type `TimeStamp`) to the union of the sets of legal values for type `t` for each instant $t'$ in $\mathcal{TIME}$. The value of a variable of type $temporal(\mathtt{t})$ can thus be represented as a set of pairs $(t, f(t))$, where $f$ is a partial function, $t$ is an element of $\mathcal{TIME}$ and $f(t)$ is the value of function $f$ at time $t$. Usually, the value of a variable of type $temporal(\mathtt{t})$ does not change at each instant. Therefore, its value can be, more concisely, represented as a set of pairs $\{\langle I_1, v_1\rangle, \ldots, \langle I_n, v_n\rangle\}$, where $v_1, \ldots, v_n$ are legal values for `t`, and $I_1, \ldots, I_n$ are time intervals, such that the variable assumes the value $v_i$ for each instant in $I_i$, $i \in [1, n]$. We adopt this representation throughout the paper.

**Example 1** Let $t$ be an instant, $\mathtt{i}_1$ and $\mathtt{i}_2 \in \mathcal{OI}$ such that $\mathtt{i}_1, \mathtt{i}_2 \in \pi(\mathtt{Person}, t)$.

- $\mathtt{10,100} \in [\![\mathtt{Short}]\!]_t$;
- $\{\mathtt{i}_1,\mathtt{i}_2\} \in [\![\ \mathtt{Set{<}Person{>}}\ ]\!]_t$;
- $\langle$ `Bob name;` $\{\langle \mathtt{I}_1\mathtt{,40}\rangle\}$ `score`$\rangle \in [\![$ `Struct`$\{$`string name;` $temporal$(`Short`) `score`$\}$ $]\!]_t$

  where:

  $\mathtt{I}_1\mathtt{.start}= \langle\ \langle \mathtt{1995,4,20}\rangle, \langle \mathtt{00,00,00}\rangle\rangle$

  $\mathtt{I}_1\mathtt{.end}= \langle\ \langle \mathtt{1995,4,25}\rangle, \langle \mathtt{00,00,00}\rangle\ \rangle$

  Note that the value of type `TimeStamp` where the second component of the pair is $\langle \mathtt{00,00,00}\rangle$ denotes the first instant of the day corresponding to the considered date. $\diamond$

## 4 Classes

As we have seen in Section 2 an object type interface or, analogously, a class interface consists of a set of property (attribute and relationship) specifications, and a set of operation specifications.

Each attribute is characterized by its name, its *temporal nature* and its type. Each relationship is characterized by its name, its temporal nature, its type[10] and its inverse. $T$_ODMG supports properties of three different temporal kinds: temporal, immutable and static. A temporal property is a property whose value[11] may change over time, and whose values at different times are recorded in the database. An immutable property is a property whose value cannot be modified during the object lifespan, whereas a static property is a property whose value can change over time, but whose past values are not recorded in the database.

[10] Suppose that a relationship is defined between two classes $C_1$ and $C_2$, then in $C_1$ the type of the relationship is $C_2$; similarly, in $C_2$, the type of the inverse relationship is $C_1$.

[11] In this context by value of a relationship we mean the set of objects involved in the relationship.

In $T_$ODMG a *lifespan* is associated with each class, representing the time interval during which the class has existed. We assume the lifespan of a class to be contiguous.

**Definition 8** (*Class Interface*). A class interface is a 6-tuple $(c, c\_type, lifespan, attr, rel, meth)$, where:

$c \in \mathcal{CI}$ is the class identifier;
$c\_type \in \mathcal{OTT} \cup \mathcal{CI}$ is the class type;
$lifespan \in (\mathcal{TIME} \times \mathcal{TIME})$ is the lifespan of the class;
$attr$ is a set containing an item for each attribute of the class. Each item is a 3-tuple $(a\_name, a\_nature, a\_type)$, where:

- $a\_name \in \mathcal{AN}$ is the attribute name;
- $a\_nature \in \{\texttt{temporal,static,immutable}\}$ is the attribute temporal nature;
- $a\_type \in$ ODMG$\mathcal{ST}$ is the attribute domain type;

$rel$ is a set containing an item for each relationship of the class. Each item is a 4-tuple $(r\_name, r\_nature, r\_type, r\_inv)$, where:

- $r\_name \in \mathcal{RN}$ is the relationship name;
- $r\_nature \in \{\texttt{temporal,static,immutable}\}$ is the relationship temporal nature;
- $r\_type$ can be of two different categories: $r\_type \in \mathcal{CI}$ or $r\_type \in \{\texttt{Set <t>} \mid \texttt{t} \in \mathcal{CI}\} \cup \{\texttt{Bag <t>} \mid \texttt{t} \in \mathcal{CI}\} \cup \{\texttt{List <t>} \mid \texttt{t} \in \mathcal{CI}\} \cup \{\texttt{Array <t>} \mid \texttt{t} \in \mathcal{CI}\}$;
- $r\_inv$ is a pair denoting the inverse traversal path of the relationship, that is, $r\_inv = (r\_inv\_name, r\_inv\_type)$ where:
  - $r\_inv\_name \in \mathcal{RN}$ is the inverse relationship name;
  - $r\_inv\_type \in \mathcal{OTT}$ is the inverse relationship domain type;

$meth$ is a set containing an item for each method of the class. Each item is a pair $(m\_name, m\_sign)$, where:

- $m\_name \in \mathcal{MN}$ is the method name;
- $m\_sign$ is the signature of the method, expressed as:

$$p\_name_1 : \texttt{t}_1 \times \ldots \times p\_name_j : \texttt{t}_j \rightarrow \texttt{t}_{j+1} \times \ldots \times \texttt{t}_n$$

  where $p\_name_i$ and $\texttt{t}_i \in \mathcal{T}$, $i \in [1, j]$, denote names and types of input parameters and and $\texttt{t}_i \in \mathcal{T}$, $i \in [j+1, n]$, denote types of output parameters.[12]

□

[12] Note that in the ODMG model input parameters have names, whereas output parameters do not have names. Moreover note that if n=0 the method has no parameters, the notation $p\_name_1 : \texttt{t}_1 \times \ldots \times p\_name_j : \texttt{t}_j \rightarrow \texttt{void}$ denotes a method with no output parameters and, similarly, $\rightarrow \texttt{t}_{j+1} \times \ldots \times \texttt{t}_n$ denotes a method with no input parameters.

In $T_$ODMG the distinction between the class identifier and the class type is relevant. $T_$ODMG supports two different kinds of object types: atomic object types and collection object types. For what concerns atomic object types, the type of the class coincides with the class identifier.[13] By contrast, the class identifiers of a collection object type is the name assigned to it by the user, while its class type is a type belonging to $\mathcal{COTT}$.

Note that the *attr* and *rel* components of the interface constitute the property specification.

In order to avoid redundancy, we allow only the use of static types in the definition of attribute types and relationship domains. For example, we specify a temporal attribute $a$ of type $temporal(\mathtt{t})$ by means of the 3-tuple $(a, \mathtt{temporal}, \mathtt{t})$ instead of by $(a, \mathtt{temporal}, temporal(\mathtt{t}))$. Given an attribute specification $(a, nat, \mathtt{t})$ we can easily derive the type $\mathtt{t}'$ of $a$. If $nat \in \{\mathtt{temporal}, \mathtt{immutable}\}$ then $\mathtt{t}' = temporal(\mathtt{t})$ if $nat \in \{\mathtt{temporal}, \mathtt{immutable}\}$; if $nat \in \{\mathtt{static}\}$ then $\mathtt{t}' = \mathtt{t}$. We impose the constraint, on attribute specification, that type $\mathtt{t}'$, computed with the method sketched above, must be a legal $T_$ODMG type. For instance, $(a, \mathtt{temporal}, \mathtt{Set}{<}temporal(\mathtt{t}''){>})$ is an invalid attribute specification since $temporal(\mathtt{Set}{<}temporal(\mathtt{t}''){>})$ is not a legal $T_$ODMG type. The same considerations hold for relationships.

We consider immutable attributes as a particular kind of temporal ones, that is, as temporal attributes whose values never change during the object lifespan. Thus, the value of an immutable attribute is modeled by means of a constant function from the temporal domain to the set of legal values for the attribute. Therefore, we can distinguish among temporal, immutable and static attributes. A temporal attribute is an attribute with a temporal type and whose value is a function from a temporal domain; an immutable attribute is an attribute with a temporal type but whose value is a *constant* function[14] from the temporal domain; finally, a static attribute is an attribute with a static type (that is, whose value is not a function from the temporal domain).

As we have seen in Section 2, a relationship can be between two types, each of which must have instances that can be referenced by object identifiers. Moreover to implement one-to-many and many-to-many relationships the relationship domain can be a set, a bag, a list or an array of objects. This is the reason why in the specification of the component *rel* of a class, *r_type* can be an object type or a collection type, whose inner type is an object type.

Similarly to attributes, relationships can be of three different types: temporal, static and immutable. If a relationship $r$ is temporal, we are interested in recording the entire history of the relationship. This means that, for each object, we are interested in keeping track of the objects, or the set (bag, list, array) of objects, which are connected to it through $r$ during each instant of the object lifespan. We can view the value of a temporal relationship, whose *r_type* is a type $\mathtt{t}$, as a partial function, $r$, from $\mathcal{TIME}$ to the union of the sets of legal values for type $\mathtt{t}$, for each instant $t'$ in $\mathcal{TIME}$, such that if $r(t')$ is defined then

[13] We recall that $\mathcal{AOT} = \mathcal{CI}$.

[14] In both temporal and immutable attributes the function is partial.

$r(t') \in [\![\, \mathtt{t} \,]\!]_{t'}$. Thus, the value of a temporal relationship where $r\_type = \mathtt{t}$ is of type $temporal(\mathtt{t})$, where $temporal(\mathtt{t})$ is a $T$_ODMG legal type. The relationships `teaches` and `is_taught_by`, introduced in Section 2, are examples of temporal relationships. Usually a professor teaches different courses in different periods of his career and it could be useful to keep track of the courses a professor teaches over time. Static relationships are relationships with a static type. For what concerns this kind of relationships, we are interested only in the current value and not in recording the past values of the relationship. An example of static relationship is the relationship `live_in` between the classes `Person` and `City`. Usually one is interested in knowing the city where a person *actually* lives and not in the cities where the person has lived before. Immutable relationships are a particular type of temporal relationships. The value of an immutable relationship $r$, with $r\_type = \mathtt{t}$, is a constant function from $\mathcal{TIME}$ to type $\mathtt{t}$, that is, the type of an immutable relationship is $temporal(\mathtt{t})$, but the value is constant for each time instant. An example of an immutable relationship could be the relationship `son_of`, relating a person with his parents.

As we have seen in Section 2 supertype information, extent naming and specification of keys are characteristics of classes. Class characteristics can be defined as follows.

**Definition 9** (*Class Characteristics*). Let $C$ be a class and let $c$ be its class identifier, the characteristics of $C$ are represented as a 3-tuple $(super, extent, keys)$, where:

*super* is the set of class names of the direct superclasses of $c$;
*extent* is a pair $(e\_name, e\_set)$, where:
  *e_name* is the extent name;
  *e_set* keeps track of the instances in the extent of $C$ over time; it is a value of type $temporal(\mathtt{Set}{<}c{>})$;
*keys* is a set of pairs $(k\_nature, prop)$ where:
  $k\_nature \in \{\mathtt{absolute,\ relative}\}$;
  $prop \in \mathcal{AN} \cup \mathcal{RN}$.

□

As in the ODMG data model these characteristics are not mandatory for a class, any of the components of the 3-tuple may be empty.

The class characteristics affected by the introduction of temporal features are *extent* and *keys*. Since we do not consider schema modifications, the set of superclasses of a given class $C$ is invariant over time.

Usually, the extent, that is, the set of all the instances of a class, is associated with each class. In a temporal context the extent of a class can vary over time. Then, a temporal value $extent.e\_set$ is associated with each class $C$ representing the objects instances of $C$ over time: $extent.e\_set$ is a value of type $temporal(\mathtt{Set}{<}c{>})$, where $c$ is the identifier of class $C$.

In the remainder of the discussion we use function: $pe\_set : \mathcal{CI} \times \mathcal{TIME} \rightarrow \mathcal{COT}$, which takes as argument a class identifier $c$ and an instant $t$ and returns the set of oids of objects for which $c$ is the most specific type along the inheritance hierarchy at time $t$, that is, the set of proper instances of $c$. If $c$ is the identifier of the class, the type of the value returned by $pe\_set$ is `Set` <c>.

Let $C$ be the class identified by $c$. Then, $pe\_set(c,t) \subseteq C.extent.e\_set(t)$, $\forall t \in C.lifespan$, since all objects instances of a class at a given instant are also proper instances of the class at the same instant. Function $\pi$ (cfr. Table 1), is such that, for each class name $c$ and for each $t \in C.lifespan$, $\pi(c,t) = C.extent.e\_set(t)$.

For what concerns *keys*, each property part of a key can be of two kinds: absolute or relative. In order to clarify the meaning of these kinds of keys we have to distinguish between static, immutable and temporal properties.

- If the property is static and the key is `relative`, no two instances may have the same value for the property within the same extent.
- If the property is temporal or immutable and the key is `relative`, no two instances may have the same value for the property within the same extent for an overlapping period of time.
- If the property is static, temporal or immutable and the key is `absolute`, no two instances may have the same value for the property even if they belong to different extents and in different time periods.

**Example 2** Consider a class `Person`. Suppose that objects of this class have as attributes a `ssn` and a `name` which are immutable during the object lifetime, an `address` whose variations over time are not relevant for the application at hand. Two temporal relationships `spouse` and `children` and two operations `marriage` and `move` belong to the interface, too. The corresponding $T_$ODMG class interface is:

$c = c\_type = \texttt{Person}$
$lifespan = [start, now]$
$attr = \{(\texttt{ssn}, \texttt{immutable}, \texttt{String}), (\texttt{name}, \texttt{immutable}, \texttt{String}),$
$\quad (\texttt{address}, \texttt{static}, \texttt{String})\}$
$rel = \{(\texttt{spouse}, \texttt{temporal}, \texttt{Person}, (\texttt{spouse}, \texttt{Person})),$
$\quad (\texttt{children}, \texttt{temporal}, \texttt{Set<Person>}, (\texttt{parents}, \texttt{Person})),$
$\quad (\texttt{parents}, \texttt{temporal}, \texttt{Set<Person>}, (\texttt{children}, \texttt{Person}))\}$
$meth = \{(\texttt{marriage}, \texttt{p} : \texttt{Person} \rightarrow \texttt{Boolean}), (\texttt{move}, \texttt{newaddress} : \texttt{String} \rightarrow)\}$

Suppose moreover that `Person` is a direct subclass of `Object`. Let $\texttt{i}_1, \ldots, \texttt{i}_4 \in \mathcal{OI}$ denote instances of type `Person`. Suppose that two instances of the class cannot have the same `ssn` whatever extent is considered, or be married to the same person in an overlapping period of time. Then, the corresponding class characteristics are:

$super = \texttt{Object}$
$extent = (\texttt{person}, \{\langle [start, now], \{\texttt{i}_1, \ldots, \texttt{i}_4\}\rangle\})$
$keys = \{\langle \texttt{absolute}, \texttt{ssn}\rangle, \langle \texttt{relative}, \texttt{spouse}\rangle\}$

$\diamond$

According to the previous definitions the class specification is defined as follows.

**Definition 10** (*Class*). A class $C$ is a 3-tuple $(int, char, impl)$ where:

$int$ is the class interface, as specified by Definition 8;
$char$ are the class characteristics, as specified by Definition 9;
$impl$ is the class implementation.

□

Note that, due to space limitations, in the paper we do not give any description of class implementation. An informal definition of class implementation in ODMG can be found in [4]. To simplify the notation, given a class $C$ we denote each component of its interface with the following dot notation: $C.c$ denotes the class identifier,[15] $C.lifespan$ denotes the class lifespan, and so on. The same simplified notation is used for class characteristics. Note that we can adopt this simplified notation because each component of a class interface and of a class characteristics has a distinct name.

We now discuss relationships between a class and its property type. The identifier $c$, of a class $C$, denotes its corresponding object type. Such object type is the type of the identifiers of the objects instances of $C$. Suppose that class $C$ has as $attr$ component the set: $\{attr_1, \ldots, attr_n\}$, where $attr_i = (a_i, nat_i, \mathtt{t}_i)$, $i \in [1, n]$ and as $rel$ component the set: $\{rel_1, \ldots, rel_{\bar{n}}\}$, where $rel_i = (r_i, nat_i, \mathtt{d}_i, inv_i)$, $i \in [1, \bar{n}]$. The following property types can be associated with $C$.

- *Structural property type.* It represents the type of the attributes and relationships of instances of $C$. It is defined by the pair $(t\_attr, t\_rel)$ where:
  $t\_attr = \mathtt{Struct}\{\mathtt{t}'_1\ a_1; \ldots; \mathtt{t}'_n\ a_n\ \}$ where $\forall i \in [1, n]$: $\mathtt{t}'_i = \mathtt{t}_i$, if $nat_i$=$\mathtt{static}$; $\mathtt{t}'_i = temporal(\mathtt{t}_i)$, if $nat_i \in \{\mathtt{immutable}, \mathtt{temporal}\}$;
  $t\_rel = \mathtt{Struct}\{\mathtt{d}'_1\ r_1; \ldots; \mathtt{d}'_{\bar{n}}\ r_{\bar{n}}\ \}$, where $\forall i \in [1, \bar{n}]$: $\mathtt{d}'_i = \mathtt{d}_i$, if $nat_i$=$\mathtt{static}$; $\mathtt{d}'_i = temporal(\mathtt{d}_i)$, if $nat_i \in \{\mathtt{immutable}, \mathtt{temporal}\}$.
- *Historical property type.* It represents the type of the temporal properties of instances of $C$. It is defined by the pair $(t\_attr, t\_rel)$ where:
  $t\_attr = \mathtt{Struct}\{\mathtt{t}'_k\ a_k; \ldots; \mathtt{t}'_m\ a_m\ \}$ where $1 \leq k \leq m \leq n$ and $\{a_k, \ldots, a_m\} = \{a_i \mid a_i \in \{a_1, \ldots, a_n\}\ \wedge\ nat_i \in \{\mathtt{immutable}, \mathtt{temporal}\}\}$;
  $t\_rel = \mathtt{Struct}\{\mathtt{d}'_{\bar{k}}\ r_{\bar{k}}; \ldots; \mathtt{d}'_{\bar{m}}\ r_{\bar{m}}\ \}$, where $1 \leq \bar{k} \leq \bar{m} \leq \bar{n}$ and $\{r_{\bar{k}}, \ldots, r_{\bar{m}}\} = \{r_i \mid r_i \in \{r_1, \ldots, r_{\bar{n}}\}\ \wedge\ nat_i \in \{\mathtt{immutable}, \mathtt{temporal}\}\}$.
- *Static property type.* It represents the type of the static attributes and relationships of instances of $C$. It is defined by the pair $(t\_attr, t\_rel)$ where:
  $t\_attr = \mathtt{Struct}\{\mathtt{t}'_j\ a_j; \ldots; \mathtt{t}'_l\ a_l\ \}$, where $1 \leq j \leq l \leq n$ and $\{a_j, \ldots, a_l\} = \{a_i \mid a_i \in \{a_1, \ldots, a_n\} \wedge\ nat_i = \mathtt{static}\}$;
  $t\_rel = \mathtt{Struct}\{\mathtt{d}'_{\bar{j}}\ r_{\bar{j}}; \ldots; \mathtt{d}'_{\bar{l}}\ r_{\bar{l}}\ \}$, where $1 \leq \bar{j} \leq \bar{l} \leq \bar{n}$ and $\{r_{\bar{j}}, \ldots, r_{\bar{l}}\} = \{r_i \mid r_i \in \{r_1, \ldots, r_{\bar{n}}\} \wedge\ nat_i = \mathtt{static}\}$.

[15] The correct notation would be $C.int.c$.

The notions of *structural*, *historical* and *static* property types of a class will be used in the next section to check object consistency. We define three functions: *type*, *h_type*, *s_type*: $\mathcal{CI} \rightarrow \mathcal{T} \times \mathcal{T}$, taking as argument a class identifier $c$, and returning the structural, the historical and the static property type of the class identified by $c$, respectively.[16]

## 5 Objects

$T$_ODMG handles in a uniform way both historical and static objects. An object is historical if it contains at least one property with a temporal domain, it is static otherwise. Each object has a lifespan, representing the time interval during which the object exists. As for class interfaces, we assume the lifespan of an object to be contiguous. Objects can be instances of different classes during their lifetime, but we can assume that, for each instant in their lifespan, there exists at least an interface extent to which they belong to.[17] For example, an employee can be fired and rehired, but he remains instance of the class *person*, superclass of the class *employee*, till the end of its lifetime. Moreover, for each historical object the history of the most specific class to which it belongs to during its lifespan is recorded. On the contrary, for each static object, only the class identifier of the most specific class to which it currently belongs to is maintained.

**Definition 11** (*Object*). An object $o$ is a 5-tuple $(i, lifespan, v, r, class\text{-}history)$, where:

$i \in \mathcal{OI}$ is the oid of $o$;

$lifespan \in (\mathcal{TIME} \times \mathcal{TIME})$ is the lifespan of $o$;

$v \in \mathcal{V}$ is a value, containing the values of each attribute of $o$. It is a struct value $\langle v_1^a\ a_1, \ldots, v_n^a\ a_n \rangle$, where $a_1, \ldots, a_n \in \mathcal{AN}$ are the names of the attributes of $o$ and $v_1^a, \ldots, v_n^a \in \mathcal{V}$ are their corresponding values;

$r \in \mathcal{V}$ is a value, containing the values of each relationship of $o$. It is a struct value $\langle v_1^r\ r_1, \ldots, v_n^r\ r_n \rangle$, where $r_1, \ldots, r_n \in \mathcal{RN}$ are the names of the relationships of $o$ and $v_1^r, \ldots, v_n^r \in \mathcal{V}$ are their corresponding values;

$class\text{-}history$ stores information about the most specific class to which $o$ belongs to over time. It is a set $\{\langle I_1, c_1 \rangle, \ldots, \langle I_n, c_n \rangle\}$, where $I_1, \ldots, I_n$ are time intervals, $c_1, \ldots c_n$ are class identifiers, such that $c_i$ is the class identifier of the most specific class to which $o$ belongs to in $I_i$, $i \in [1, n]$. □

If $o$ is static, *class-history* records only to the most specific class to which $o$ currently belongs.

[16] Note that function *h_type* returns a null value when its argument is the identifier of a class whose instances are static, whereas function *s_type* returns a null value when its argument is a class whose instances only have temporal properties.

[17] This class is the most general class (in the inheritance hierarchy) the object has ever belonged to.

**Example 3** Suppose that $\mathtt{i}_1, \ldots, \mathtt{i}_7 \in \mathcal{OI}$ and $\mathtt{Person} \in \mathcal{CI}$. The following is an example of $T$_Chimera object:

$i = \mathtt{i}_1$
$lifespan = I$ where $I = \langle t_1, t_2 \rangle$ and:
$t_1 = \langle\langle \mathtt{1965, 3, 21} \rangle, \langle \mathtt{00, 00, 00} \rangle\rangle$ $t_2 = now$
$v = \langle$ $\{\langle I,$ `JS65I23`$\rangle\}$ `ssn`,
$\{\langle I,$ `JohnSmith`$\rangle\}$ `name`,
`"Fifth Avenue 275 NY"` `address` $\rangle$
$r = \langle$ $\{\langle\langle t_3, t_2 \rangle,$ $\mathtt{i}_2\rangle\}$ `spouse`, $\{\langle\langle t_4, t_5\rangle,$ $\{\mathtt{i}_3\}\rangle, \langle\langle t_6, t_2\rangle,$ $\{\mathtt{i}_3, \mathtt{i}_4\}\rangle\}$ `children`$\rangle$,
where:
$t_3 = \langle\langle \mathtt{1990, 5, 14} \rangle, \langle \mathtt{00, 00, 00} \rangle\rangle$
$t_4 = \langle\langle \mathtt{1993, 12, 31} \rangle, \langle \mathtt{00, 00, 00} \rangle\rangle$
$t_5 = \langle\langle \mathtt{1995, 1, 17} \rangle, \langle \mathtt{23, 59, 59} \rangle\rangle$
$t_6 = \langle\langle \mathtt{1995, 1, 18} \rangle, \langle \mathtt{00, 00, 00} \rangle\rangle$
$class\text{-}history = \{\langle I, \mathtt{Person}\rangle\}$

$\diamond$

In a temporal context, several temporal constraints must be satisfied by object lifespans. To formalize these constraints we define function $o\_lifespan$: $\mathcal{OI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$, that given an oid $i$ returns the lifespan of the object identified by $i$. Obviously, information about the historical extent of a class must be consistent with the class histories of the objects in the database, as stated by the following invariant.

**Invariant 1** $\forall i' \in \mathcal{OI}, \forall c' \in \mathcal{CI}, \forall t \in \mathcal{TIME}$, let $o$ be the object such that $o.i = i'$, $C$ be the class such that $C.c = c'$, then

1. $i' \in C.e\_set(t) \Rightarrow t \in o\_lifespan(i')$;
2. $(\forall t \in I, i' \in pe\_set(c', t)) \Leftrightarrow \langle I, c' \rangle \in o.class\text{-}history.$ $\triangle$

Moreover, the lifespan of an object can be partitioned into a set of intervals, depending on the object most specific class. Indeed, an object can be member of different classes during its lifetime. Therefore, we introduce function $c\_lifespan$: $\mathcal{OI} \times \mathcal{CI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$, that given an oid $i$ and a class identifier $c$, returns the interval representing the set of time instants in which $i$ was a member of the class identified by $c$.[18] The temporal constraints stated by the following invariant must be satisfied.

**Invariant 2** $\forall i' \in \mathcal{OI}, \forall c' \in \mathcal{CI}, \forall t \in \mathcal{TIME}$, then

1. $o\_lifespan(i') = \bigcup_{c \in \mathcal{CI}} c\_lifespan(i', c)$;
2. $t \in c\_lifespan(i, c') \Leftrightarrow i \in C.e\_set(t)$, where $c' = C.c$.[19] $\triangle$

[18] Note that $c\_lifespan(i, c) = \bigcup_{\langle I_i, c_i \rangle \in o.class-history,\ c_i \text{ subclass of } c} I_i$. Functions $o\_lifespan$ and $c\_lifespan$ are similar to those defined in [18].

[19] This also implies that $t \in c\_lifespan(i', c') \Leftrightarrow i' \in [\![ c' ]\!]_t$.

### 5.1 Consistency notions

Because of object migrations, the most specific class to which an object belongs to can vary over time. Moreover, an object can be an instance of the same class in different, not consecutive, time instants. As an example, consider the case of an employee that is promoted to manager, (*manager* being a subclass of *employee* with some extra attributes, like *subordinates* and *official_car*). The other case is the transfer of the manager back to normal employee status. The migration of an object from a class to another can cause the addition or the deletion of some attributes from the object. For instance, the promotion of an employee to the manager status has the effect of adding the attributes *subordinates* and *official_car* to the corresponding object, while the transfer of the manager back to the employee status causes dropping the attributes *subordinates* and *official_car* from the corresponding object. If the attributes *subordinates* and *official_car* are static, they are simply deleted from the object. No track of their existence is recorded in the object when it migrates to the class *employee*. If they are temporal, the values they had when the object migrated to the class *manager* are maintained in the object, even if they are not part of the object anymore.

We require that each object must be a consistent instance of all the classes to which it belongs to. In a context where objects can have both static and temporal attributes, the notion of consistency assumes a slightly different semantics with respect to its classical definition. Verifying the consistency of an object in a temporal context requires two steps. First, the set of attributes characterizing the object for each instant $t$ of its lifespan must be determined. Then, the correctness of their values must be checked. Note that, if we consider an instant $t$ less than the current time, we are able to identify only the temporal attributes characterizing the object at time $t$, since for static attributes we record only their current values. Thus, for instants lesser than the current time, it only makes sense to check the correctness of the values of the temporal and immutable attributes of the objects. Therefore, we start by introducing the following definition.

**Definition 12** (*Meaningful Temporal Properties* ). Let $p$ be a temporal[20] property of an object $o$. Property $p$ is said to be meaningful for $o$ at time $t$, if $p$ is defined at time $t$. □

We distinguish two kinds of consistency:

- *Historical consistency.* The values[21] of the temporal properties of the object at a given instant are legal values for the temporal properties of the class.
- *Static consistency.* The values of the static attributes of the object are legal values for the static attributes of the class.

[20] From now on with temporal property we mean a property which is temporal or immutable.

[21] Given an object $o$, with value of a relationship we mean the object or the set of objects in relationship with $o$.

Consider, for instance, an object $o = (i, lifespan, v, r, class\text{-}history)$, such that $v = \langle v_1^a\ a_1, \ldots, v_n^a\ a_n \rangle$ and $r = \langle v_1^r\ r_1, \ldots, v_n^r\ r_n \rangle$. Therefore, given an instant $t \in o.lifespan$, the following values can be defined:

- *Historical value.* It is a pair $(v\_attr, v\_rel)$, where $v\_attr$ represents the values of the temporal attributes meaningful for the object at time $t$ and $v\_rel$ represents the values of the temporal relationships meaningful for the object at time $t$. Let $\{a_k, \ldots, a_m\}$, $1 \leq k \leq m \leq n$, be the subset of $\{a_1, \ldots, a_n\}$ consisting of all the names of the temporal attributes meaningful for $o$ at time $t$ and let $\{r_j, \ldots, r_l\}$, $1 \leq j \leq l \leq n$, be the subset of $\{r_1, \ldots, r_n\}$ consisting of all the names of the temporal relationships meaningful for $o$ at time $t$. The historical value of $o$ at time $t$ is defined as $(v\_attr, v\_rel)$ where:
  - $v\_attr = \langle v_k^a(t)\ a_k, \ldots, v_m^a(t)\ a_m \rangle$, where $v_i(t)$ denotes the value of $a_i$ at time $t$, $i \in [k, m]$;
  - $v\_rel = \langle v_j^r(t)\ r_j, \ldots, v_l^r(t)\ r_l \rangle$, where $v_i^r(t)$ denotes the value of $r_i$ at time $t$, $i \in [j, l]$.
- *Static value.* It is a pair representing the values of the static properties of the object. Its definition is analogous to that of the historical value, considering static properties instead of temporal ones.

Thus, we define two functions: *h_state*: $\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V} \times \mathcal{V}$, receiving an object identifier and an instant $t$ as input, and returning the historical value of the object at time $t$; *s_state*: $\mathcal{OI} \rightarrow \mathcal{V} \times \mathcal{V}$, receiving an object identifier as input, and returning the static value of the object. Note that when an object consists only of temporal properties, *h_state* returns a snapshot of the value of the object properties for a specified time instant.

We are now ready to formally introduce the notions of *historical* and *static* consistency, by making use of functions *h_type* and *s_type* (cfr. Table 1). In the following, $\Pi_i(\langle e_1, \ldots, e_n \rangle)$ denotes the $i$-th component of the tuple $\langle e_1, \ldots, e_n \rangle$.

**Definition 13** (*Historical Consistency*). An object $o = (i, lifespan, v, r, class\text{-}history)$ is an historically consistent instance of a class $c'$ at time $t$ iff the following conditions hold:

- $\Pi_1(h\_state(o.i, t))$ is a legal value for the attribute specification $\Pi_1(h\_type(c'))$;
- $\Pi_2(h\_state(o.i, t))$ is a legal value for the relationship specification $\Pi_2(h\_type(c'))$.

□

**Definition 14** (*Static Consistency*). An object $o = (i, lifespan, v, r, class\text{-}history)$ is a statically consistent instance of a class $c'$, if the following conditions hold:

- $\Pi_1(s\_state(o.i, t))$ is a legal value for the attribute specification $\Pi_1(s\_type(c'))$;
- $\Pi_2(s\_state(o.i, t))$ is a legal value for the relationship specification $\Pi_2(s\_type(c'))$.

□

The consistency of an object is checked only with respect to its most specific class, since if an object is consistent with respect to its most specific class, it is also consistent with respect to all its superclasses.

**Definition 15** (*Object Consistency*). An object $o = (i, lifespan, v, r, class\text{-}history)$ is consistent iff the following conditions hold:

- For each pair $\langle I, c' \rangle$ in $o.class\text{-}history$, interval $I$ is contained in the lifespan of the class identified by $c'$, that is, $I \subseteq C.lifespan$, where $C$ is the class such that $C.c = c'$.
- For each instant $t$ of the object lifespan, a class $c$ to which the object belongs to exists and, vice versa, each instant $t$ such that $t \in I$ where $\langle I, c \rangle \in o.class\text{-}history$ belongs to the lifespan of the object: $\bigcup_{\langle I,c \rangle \in o.class-history} I = o.lifespan$.[22]
- If a temporal property $p$ is meaningful at time $t$ for object $o$ then $t \in o.lifespan$.
- For each pair $\langle I, c' \rangle$ in $o.class\text{-}history$, $o$ is an historically consistent instance of $c'$, for each instant $t \in I$.
- Let $\langle I, c \rangle$ be the (unique) element of $o.class\text{-}history$, such that $now \in I$. Object $o$ must be a statically consistent instance of class $c$. □

The above definition states that each object, for each instant $t$ of its lifespan, must contain a value for each temporal property of the class to which it belongs to at time $t$, and this value must be of the correct type and that each instant $t$ in which a property is defined belongs to the lifespan of the considered object. Moreover, at the current time also the consistency with respect to the static properties must be checked. This notion of consistency allows to uniformly handle both static and historical objects. In the case of static objects, Definition 15 reduces to the traditional notion of consistency.

## 6 Related work

Table 2 compares some temporal object-oriented data models proposed in the literature. Some considered approaches are compared under a different perspective by Snodgrass [12]. In [12] the emphasis is on temporal object-oriented query languages, while we consider only data model features. Moreover, in [12] only the temporal features are compared, disregarding the object-oriented ones, whereas we consider both.

Concerning the temporal aspects, most models support a linear discrete time structure,[23] whereas only few of them model a user-defined hierarchy of time types. Two time dimensions are of interest in temporal databases: *valid* time (the time a fact was true in reality) and *transaction* time (the time the fact was stored in the database). Most models consider only the valid time. Some approaches associate a timestamp with the whole object state; others associate a timestamp with each object attribute often regarding the value of a temporal attribute as a function from a temporal domain to the set of legal values for the

[22] Note that since we do not consider objects with multiple most specific classes we have that $\bigcap_{\langle I,c \rangle \in o.class-history} I = \emptyset$.

[23] We consider time structure and time dimension as discussed in [12].

attribute. This is also the approach taken in our model. Since the ODMG object model supports both attributes and relationships, both these kinds of object properties are extended with time in our model.

Another important characteristic is whether temporal, immutable and non temporal attributes are supported. A *temporal* (or historical) attribute is an attribute whose value may change over time, and whose values at different times are recorded in the database. An *immutable* attribute is an attribute whose value cannot be modified during the object lifetime,[24] whereas a *non temporal* (or static) attribute is an attribute whose value can change over time, but whose past values are not meaningful, and are thus not stored in the database. The support for static, immutable and temporal attributes has been firstly proposed in the $T$_Chimera data model [3, 2]. The main difference between $T$_Chimera and $T$_ODMG are, besides the underlying object data model extended with time, that $T$_ODMG also supports relationships, which are not supported in $T$_Chimera, and that $T$_ODMG allows the specification of (relative and absolute) keys.

Finally, some models keep track of the dynamic links between an object and its most specific class. Indeed, an important dynamic aspect of object-oriented databases is that an object can dynamically change type, by specializing or generalizing its current one.

| | [7] | [8] | [9] | [10] | [11] | [15] | [18] | [14] | Ours |
|---|---|---|---|---|---|---|---|---|---|
| o-o data model | Oodaplex | generic | Tigukat | MAD | generic | OSAM* | Oodaplex | OM | ODMG |
| time structure | linear discrete | linear discrete | user-def. | linear discrete | linear discrete | linear discrete | user-def. | linear discrete | linear discrete |
| time dimension | valid | valid | valid + trans. | valid | valid + trans. | valid | arbitrary[1] | valid | valid |
| values & objects | objects | objects | objects | objects | objects | objects | objects | both | objects |
| relationships | NO | NO | NO | NO | NO | NO | NO | YES | YES |
| what is timestamped | attr. | attr. | arbitrary | objects | attr. | objects | arbitrary | objects relat. | attr. relat. |
| temp. attr. values | funct.[2] | funct.[2] | sets of pairs | atomic valued[3] | lists of tuples | atomic valued[3] | atomic valued[3] | funct.[2] | funct.[2] |
| kinds of attributes | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm. | temp. + imm + non-temp. |
| histories of obj. types | NO | YES | YES | NO | NO | NO[4] | YES | YES | YES |

Legenda:

[1] One time dimension is considered, it can be transaction or valid time.
[2] With *funct.* we denote functions from a temporal domain.
[3] Time is associated with the entire object state.
[4] The information can be derived from the histories of object instances.

**Table 2.** Comparison among temporal object-oriented data models

[24] Immutable attributes can be regarded as a particular case of temporal ones, since their value is a constant function from a temporal domain.

## 7 Conclusions and future work

In this paper we have presented $T$_ODMG, a temporal extension of the ODMG object standard data model. $T$_ODMG supports temporal, static and immutable object properties. We have introduced the notion of temporal type and defined the set of legal values for each type. We have discussed the notion of object consistency and integrity and we have revised the notion of key in a temporal framework. A prototype implementation of the proposed model has been implemented on top of the Ode OODBMS [1], in which the histories of temporal object properties are organized as monotonic $B^+$-trees.

We plan to extend this work along several directions. First, we are investigating a temporal extension of OQL, the ODMG query language. A second relevant extension is the support for multiple time granularities and calendars. Indeed, in the current work we have prefered to keep our model of time as simple as possible and to provide a clear formalism for such model. On the top of this formalism we plan to investigate the use of multiple granularities and calendars and the problems related to the data accesses in such framework.

A third direction deals with mechanisms supporting a selective recording of past values of properties, according to the truth value of conditions associated with properties. Moreover, we plan to extend this work with time-dependent implementations. The informal idea of time-dependent implementations is that each method has a set of implementations, each referring to a different time interval. A formal treatment of temporal methods in the context of the $T$_Chimera model can be found in [2].

Finally, we plan to extend our work to comply with the recently published ODMG 2.0 [5] standard.

## References


1. E. Bertino, M. Bevilacqua, E. Ferrari, and G. Guerrini. Approaches to Handling Temporal Data in Object-Oriented Databases. Technical Report 192-97, Dipartimento di Scienze dell'Informazione, Università di Milano, Ottobre 1997.
2. E. Bertino, E. Ferrari, and G. Guerrini. T_Chimera: A Temporal Object-Oriented Data Model. *Theory and Practice of Object Systems*, 3(2):103–125, 1997.
3. E. Bertino, E. Ferrari, and G. Guerrini. A Formal Temporal Object-Oriented Data Model. In P. Apers, editor, *Proc. Fifth Int'l Conf. on Extending Database Technology*, number 1057 in Lecture Notes in Computer Science, pages 342–356, Avignon (France), March 1996.
4. R. Cattel. *The Object Database Standard: ODMG-93*. Morgan-Kaufmann, 1996.
5. R. Cattel. *The Object Database Standard: ODMG 2.0*. Morgan-Kaufmann, 1997.
6. P. Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
7. T. Cheng and S. Gadia. An Object-Oriented Model for Temporal Databases. In *Proc. of the Int'l Workshop on an Infrastructure for Temporal Databases*, 1993.
8. J. Clifford and A. Croker. Objects in Time. In *Proc. Fourth IEEE Int'l Conf. on Data Engineering*, pages 11–18, 1988.

9. I. Goralwalla and M. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. Twelfth Int'l Conf. on the Entity-Relationship Approach*, volume 823 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, Berlin, 1993.
10. W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 266–275. ACM Press, 1992.
11. E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. Tenth Int'l Conf. on the Entity-Relationship Approach*, pages 205–229, 1991.
12. R. T. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 386–408. Addison-Wesley/ACM Press, 1995.
13. R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.
14. A. Steiner and M.C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Proc. of the Fifth International Conference on Database Systems for Advanced Applications*, pages 381–390, 1997.
15. S. Su and H. Chen. A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 431–441, 1991.
16. A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Database Systems and Applications Series. Benjamin/Cummings, 1993.
17. Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. Available at: http://www.isse.gmu.edu/~csis/tdb/bib97/bib97.html, 1997.
18. G. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 230–247. Benjamin/Cummings, 1993.

This article was processed using the LaTeX macro package with LLNCS style