Object migration in temporal object-oriented databases

Object migration in temporal object-oriented databases

Angelo Montanari Elisa Peressi
Dipartimento di Matematica e Informatica, Università di Udine
Via Zanon, 6 - 33100 Udine, Italy
e-mail: [montana,peressi]@dimi.uniud.it

Barbara Pernici
Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci, 32 - 20133 Milano, Italy
e-mail: pernici@elet.polimi.it

Keywords: object-oriented databases, temporal databases, query languages, roles

Edited by:

Received: **Revised:** **Accepted:**

The paper presents T-ORM (Temporal Objects with Roles Model), an object-oriented data model based on the concepts of class and role. In order to represent the evolution of real-world entities, T-ORM allows objects to change state, roles and class in their lifetime. In particular, it handles structural and behavioral changes that occur in objects when they migrate from a given class to another. First, the paper introduces the basic features of the T-ORM data model, emphasizing those related to object migration. Then, it presents the query and manipulation languages associated with T-ORM, focusing on the treatment of the temporal aspects of object evolution.

1 Introduction

Since the '70s, relational databases have been successfully used in many application domains. In the last years, however, many advanced application areas have been identified for which the data model underlying relational databases is not the most appropriate one. A number of applications in the areas of CAD/CAM, office automation, knowledge representation, software engineering indeed require semantically richer data models. New constructs are needed to model structured entities, complex attribute domains, different types of relationships among entities, relationships among entity types. To support such features, object-oriented databases have been developed, and several commercial systems based on the object-oriented paradigm are now available. In addition, many of these applications must cope with problems involving temporal information about object evolution. For this reason, conventional snapshot databases, that maintain information about the current state of the world only, need to be replaced by temporal databases that record information about past, present and, possibly, future states. In order to model the evolution of real-world entities, object-oriented database must be able to handle both changes in object states and changes in object structures due to their migration to other classes [30]. As an example, they must be able to represent the fact that a person becomes an adult (class change), that a student becomes a professor (role change), and that a student moves from a given university to another one (state change) in a uniform framework.

During the last fifteen years, several time models have been proposed to manage temporal knowledge in database systems. Most of them extend the relational model with one or more time dimensions, e.g. [13, 28]. Temporal extensions of object-oriented models have been proposed in [6, 7, 8, 29, 31]. Most extensions are only concerned with the representation of state evolution, and neither support the notion of object role, nor allow the shift of an object from one class to another (they assign a class to an object once and for all).

In the paper we consider the problem of providing temporal object-oriented databases with the notion of *object migration*. The importance of such a notion has been pointed out by [20, 30]. In object-oriented databases objects belong to hierarchically structured classes, and remain statically linked to their original position in the hierarchy. On the contrary, in many application domains it is quite natural to allow objects to dynamically change the class(es) they belong to. For example, it seems fairly acceptable to allow an object belonging to the class *PERSON* to *migrate* to the subclass *ADULT*. Only few papers deal with object migration in object-oriented databases using either behavioral constructs to describe semantic information

[4], or dynamic integrity constraints [30], or considering a restricted notion of migration [10, 11]. The issue of object migration has not been addressed at all in the context of temporal object-oriented databases.

The paper describes T-ORM [19], a temporal extension of the object-oriented conceptual model ORM (Object with Roles Model [20]), that generalizes the temporal models proposed in [6, 8, 27] by adding the notion of object migration. It first analyzes in detail the notion of object evolution, and consider different types of evolution which can be of interest for database applications; then, it identifies the basic requirements that a temporal model of object evolution must satisfy; finally, it shows how object evolution may affect state, structure and behavior of the evolving object and of the objects related to it. All the features of the T-ORM model are illustrated in terms of query and manipulation languages. There is plenty of literature on temporal extensions to query languages [7, 21, 22, 24, 25, 28, 29, 31] and we do not deviate from them defining a query language which is based on the SQL syntax. Besides the usual primitives of structured query languages, it is provided with all temporal relations of Allen's interval logic and with some specific constructs that allow one to query the history of objects.

The organization of the paper is as follows. Section 2 first illustrates the ORM model and then describes the basic features of its temporal extension T-ORM. Section 3 introduces and discusses the notion of object migration, and shows how it is dealt with in T-ORM. Section 4 provides a detailed presentation of the T-ORM query language. Section 5 sketches out the basic features of T-ORM data manipulation language. The concluding remarks provide an assessment of the work.

2 The T-ORM data model

2.1 Classes and roles

One of the main problems in real-world modeling is the management of object behavior. Most of the efforts in this area have been limited by static schema definitions, supplying objects with methods which operate on object states. Recently, it has been suggested to incorporate rules within objects for expressing object behaviors. Besides the necessity of representing changes in state, another problem occurs. Many applications have the necessity of describing particular entities from different perspectives, dealing with multifaceted object states, that is, an object can play different *roles* and its behavior depends on the role it plays. The term role has been used in various contexts with different meanings. As regards our approach, similar concepts have been developed by Richardson and Schwarz [23], Su [30], Wieringa [32], Sciore [26], and Papazoglou [18].

The ORM model has been originally proposed as an object-oriented design framework for specifying information systems requirements. Such a model allows one to represent object behavior by means of the concept of *role*. A role is a state in which an object can be and if the object is in that state, we say that it *plays* that role. Traditional class-based object-oriented systems model the various states which an entity may assume using specialization hierarchies and representing real-world entities as instances of the most specific class they belong to. This approach has numerous drawbacks. Consider the following example appeared in [32].

> Assume that passenger is a subclass of person and consider a person who migrates to the passenger subclass of person, say by entering a bus. This bus may carry 4000 passengers in one week, but counted differently, it may carry 1000 persons in the same week. So counting persons differs from counting passengers.

The conclusion of this observation can be stated in terms of identifiers. If PASSENGER would be a subclass of PERSON, then each passenger identifier would also be a person identifier. Since this is not the case, persons and passengers apparently have different identifiers. We should have a different way to represent those instances. We must realize that a passenger is not *identical* to a person, but that it is a *state* of a person, or, more properly, it is a role of the class PERSON. So, when we count passengers, we really count how often persons have been playing the role of passenger. Moreover, using only the mechanism of class specialization, when we have an entity which can assume different roles independently (for example a person may be a student and an employee), we have to define a separate class which is a subclass of both EMPLOYEE and STUDENT classes. Subclasses of this type should be defined for every possible combination of roles.

In ORM, an object assumes a certain role via a mechanism of instantiation which is analogous to that used to populate classes. We talk about role instances in the same way in which we speak about class instances. Every time that a role is instantiated, we associate a unique identifier (Role Identifier o RID) with the instance which preserves instance identity across changes of its state (i.e., changes to attribute values). We assume that this identifier is unique across the database. All instances of roles evolve independently.

From another point of view, the reason why roles cannot be implemented as subclasses is that the classification mechanism does not allow multiple instantiation. As we said, a person could become a passenger more then once during a week. We cannot instantiate the same person as a passenger more than once and

also we cannot think of representing all different kinds of passengers as different subclasses. On the contrary, the role mechanism allows an object to play different roles at different times, to play more than one role at the same time, and to have more than one instance of the same role at the same time (for example, a person who is employed in two different firms). This capability is one of the features that distinguishes the ORM model from other object-oriented models with roles. As an example, in the model proposed in [18] an entity can play several roles simultaneously, but only a single occurrence of each role type is permitted per entity.

At a first glance, one could object that roles represent only particular states which an entity could assume during its lifetime and, as such, one could implement them as a multi-valued time-stamped attribute "state". In general, this is not possible because an object playing a role has a particular behavior specific of that role, which is specified in the role component of a class description through a set of rules and messages and that could not be represented with the traditional way of modeling classes.

A class in the ORM model is defined by a name $\mathtt{Cn}$ and a set of roles $\mathtt{R}_i$, each one representing a different behavior of this object:

$$\mathtt{class = (Cn, R_0, R_1, \ldots, R_n)}$$

Each role $\mathtt{R}_i$ is a 5-uple:

$$\mathtt{R}_i \mathtt{ = < Rn}_i\mathtt{, P}_i\mathtt{, S}_i\mathtt{, M}_i\mathtt{, Rr}_i \mathtt{>}$$

consisting of a role name $\mathtt{Rn}_i$, a set of properties $\mathtt{P}_i$ of that role (abstract description of object characteristics), a set of abstract states $\mathtt{S}_i$ that the object can be at while playing this role, a set of messages $\mathtt{M}_i$ that the object can receive and send in this role, and a set of rules $\mathtt{Rr}_i$.

Rules fall into two categories: state transition rules and integrity rules. State transition rules define which messages an object can receive/send in each state and the state changes these messages cause. Integrity rules specify constraints on object evolution. This is another aspect which characterizes roles: we can represent object evolution by means of rules and constraints on those rules [9].

Every class has a *base-role* R_0 that describes the initial characteristics of an instance and the global properties concerning its evolution. These properties are inherited by all the other roles; the messages of the base-role are used to add, delete, suspend and resume instances of roles; the possible states in the base-role are pre-defined (*active* and *suspended*); and the rules define transitions between roles and global constraints for the class. Each property has a property name and a domain. Domains may be simple, composite or complex. Simple domains are predefined domains (such as string, integer, real, boolean), classes, or roles; composite domains are classes and roles; complex domains are defined as aggregates, sets (unordered collections of objects) or sequences (ordered collections of homogeneous objects) of other domains (simple or complex).

Finally, a class can be a subclass of one or more classes (multiple inheritance) and inherits all roles specified in the parent class(es).

2.2 Adding time to objects

Adding the time dimension to object-oriented systems is required for modeling how the entities and the relationships the object denote may change over time [6]. Often an object is created at a given time and is relevant to a system for only a limited period of time. Furthermore, during their existence, objects may change the values of their attributes, the roles that they play, and even the classes they belong to. Temporal (object-oriented) databases may differ from each other both in the structure of the underlying time domain and in the way of associating time information to database entities.

The basic features of time domains have been precisely identified in the literature. Referring to the classification given in [1], we assume that the T-ORM time domain is bidimensional (both valid time and transaction time are supported) and linear in both dimensions, the valid time axis is unbound in both directions, whereas the transaction time axis is bound in both directions (it spans from database creation until the current instant), and both axes map to integers. Furthermore, the time point is taken as the primitive temporal entity (intervals are defined as a derived concept) and the usual metric on integers is defined to measure distances between time points.

With respect to the association of time with data, object attributes can be partitioned in *time-varying* and *constant* ones [17], depending on the fact that their value may change or not over time. The values of time-varying attributes are usually time-stamped at specific time points or intervals; therefore we do not know their value at a time where there is no a specific entry. Common assumptions about their value in such points fall into three categories: (i) *step-wise constant values*, (ii) *discrete values*, and (iii) *values changing according to a given function of time* (e.g. uniformly changing values) [17]. In cases (i) and (iii), the unknown values can be derived from the stored values using a suitable interpolation function. In case (ii), if there is no a specific entry stored at a given time, the attribute must be considered undefined. A further distinction is concerned with the choice of the data unit to time stamp. Two approaches have been proposed in the literature: attribute versioning [5], and object versioning [1]. In the first case, valid and transaction times are associated with each time-varying attribute; in the second case, they are associated with the whole object, and so to all attributes of that object. Attribute versioning presents several advantages,

including the following ones: (i) different properties may be associated with time at different granularities; (ii) some properties are inherently not time-varying, so recording time information for them is useless; (iii) time-varying properties of the same object may change asynchronously over time, so as we have to record all object values when a change occurs, we have to duplicate a lot of information (the values which did not change).

Besides associating time information to attributes, object-oriented temporal databases (OOTDBs) can temporally characterize the existence of objects, that is, they can specify when and how an object exists in the database. In most OOTDBs, the set of time intervals during which an object logically exists in the database is called its *lifespan* [6]. This object lifespan spans from the object creation (the point in time when the database first records any information about it) till its complete termination (i.e., logical deletion). As an object can be member of different classes, an object lifespan is the union of its lifespans in all classes in which it has participated. An object lifespan within a class coincides with the union of the lifespans of its properties as a member of that class. In historical object-oriented databases the notion of "reincarnation" is also supported, because a death of an object is not necessarily terminal [6]. For example, employees can be hired, fired, and subsequently rehired.

In the T-ORM model, time is associated with single attributes, class instances and role instances.

With respect to attributes, we assume that their values are step-wise constant. Therefore an object attribute identifies a sequence of values, each one associated with a different time interval, which has been called *time sequence* (TS) in the literature [27]. Due to the bidimensionality of time, time sequences are constituted by triples $< attribute\ value, valid-time\ interval, transaction-time\ interval >$. Each time interval is represented by a pair $[s, e)$, where s denotes the starting point of the interval, and e its ending point. The interval is closed at the left and open at the right. We assume that valid time intervals for a given attribute are totally ordered with respect to any given transaction point. Finally, if the attribute value has a complex structure, e.g. an aggregate, a set, or a list, we assume that valid and transaction times can be associated with both the whole structure and each of its components. As an example, suppose that the attribute *address* is defined as the aggregate composed of *street* and *town*. Time sequences for address represent changes of values of either *street* or *town*, or both.

With respect to classes and roles, we associate a time sequence with each class (role) instance to denote the time periods during which it is active. The lifespans of role instances and those of the corresponding objects are linked by specific constraints. An object after being suspended can neither send nor receive messages. Therefore, lifespans of role instances are always contained in the lifespan of the corresponding object. Formally, let o be an object instance of a class C, ρ(C) be a function that maps C to the set of roles its instances can play and $r(o, R)$ be a function that maps an object o to the set of its role instances of the role R. The following constraint must hold:

$$\forall R \in \rho(C)\ \forall r \in r(o, R)$$

$$(r.\text{LIFESPAN}) \subseteq (o.\text{LIFESPAN})$$

If

$$(r.\text{LIFESPAN}) = \{[s_1^r, e_1^r), ..., [s_n^r, e_n^r)\}$$

and

$$(o.\text{LIFESPAN}) = \{[s_1, e_1), ..., [s_m, e_m)\}$$

the given constraint states that

$$\forall i = 1, ..., n \exists j \in \{1, ..., m\} \text{ such that } [s_i^r, e_i^r) \subseteq [s_j, e_j)$$

All role instances are deleted when the corresponding object is deleted. When an object is suspended, all the roles it has instantiated are also suspended. Object suspension allows us to represent what has been called in [6] object killing and reincarnation.

Let us introduce now a simple schema that will be used in the rest of the paper as a source of exemplification (see Figure 1). We consider four classes, namely PROJECT, DOCUMENT, PERSON and ADULT, which is a subclass of PERSON. Objects belonging to the class PERSON can play two different roles (Employee and Student), each one characterized by its own properties. Projects are developed by persons playing the role of employee. Each project has associated a set of documents written by the employees who participate in the project.

In the following we present some extensions to the ORM model defined in [20] regarding the object-oriented data modeling aspects, and then we explore them in the temporal context. The main concept we examine is object migration. This important issue has still been little researched on. In fact, while existing OODBMSs may capture the notion that an adult is a person, through the mechanism of is-a hierarchies, most of them do not support the notion of a given entity being created as a person and then becoming an adult, that is an entity "migrating" along the class hierarchy it belongs to.

2.3 Composite objects

In object-oriented data models the value of an attribute can itself be an object. In this way, an object can *refer* to another object. In our model we adopt the categorization of references proposed in ORION [15]:

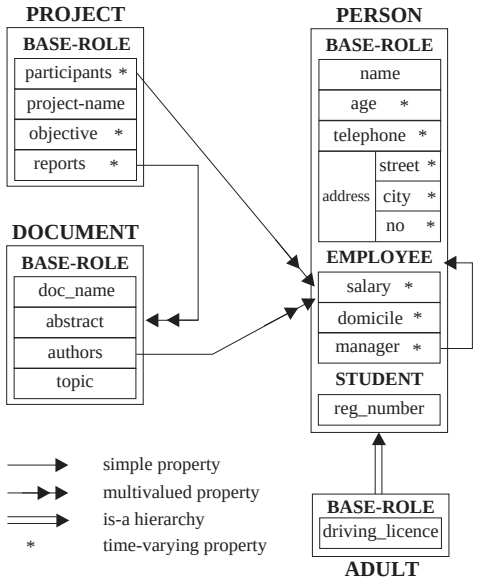

Figure 1: Example of ORM schema

- **weak references**: they are the standard references used in object-oriented systems, and are not provided with any special semantics;

- **composite references** (called also *part-of* relationships): they allow one to define composite objects, i.e., objects composed of other objects.

  A composite reference can be:

  – **exclusive** or **shared**

  In the first case, the referred object can be part of one and only one object; in the second case, it can be part of several composite objects. Two interpretations of exclusivity are possible, depending on its temporal characterization. According to a time-independent interpretation of exclusivity, an object can be part of only one object during its existence. According to a time-dependent interpretation, an object can be part of only one object at each time instant, but it can be part of different objects at different instants. In this second case, exclusivity can be expressed by the following constraint: if an object o is part of the composite objects o' and o'', then the period during which it is part of o' must have an empty intersection with the period during which it is part of o''.

  – **dependent** or **independent**

  In the first case, the referred object exists (if and) only if the composite object exists, while in the second case, the existence of the referred object does not depend on the existence of the composite object.

The classification of composite references as exclusive or shared, and as dependent or independent are orthogonal, and thus identify four different types composite references.

The main problems involved in the management of composite references concern the relationships between the creation and deletion of composite objects and the creation and deletion of their components. As an example, let o' be a composite object and o be a component of o'. We could state that the deletion of o' causes the deletion of o if one of the following two conditions hold: (i) o' has a dependent exclusive reference to o; (ii) o' has a dependent shared reference to o, but it is the only object currently involved in such a relation with o.

In [6], for instance, a rather restrictive notion of *part-of* relationship is adopted, based on the assumption that a composite object can exist only when its components exist. Such an assumption can be formalized as follows. Let us assume that some composite object o' is defined in terms of n other objects $o_1, .., o_n$ and that its lifespan consists of a set of m disjoint intervals, that is, $o'.LIFESPAN = \{[s_1, e_1), .., [s_m, e_m)\}$. Moreover, let i_m be the number of disjoint intervals belonging to the lifespan of the component object i, for each $i = 1, .., n$. According to the given assumption, the following constraint must always be satisfied:

$$\forall i \; o'.LIFESPAN \subseteq o_i.LIFESPAN$$

which is equivalent to:

$$\forall i, j (1 \leq i \leq n \; \wedge \; 1 \leq j \leq m \; \wedge [s_j, e_j] \in$$

$$o'.LIFESPAN \supset \exists k (1 \leq k \leq i_m \wedge$$

$$[s_k, e_k) \in o_i.LIFESPAN) \; \wedge \; [s_j, e_j) \subseteq [s_k, e_k)))$$

Such a solution has two major drawbacks: (i) a composite object cannot be created until all its components have been created; (ii) a composite object must be deleted when one of its components is deleted.

An alternative approach consists in making the existence of a composite object independent from the existence of its components by modeling the *part-of* relationship in terms of roles. This allows us to deal with composite objects which dynamically change their components, supporting the addition/dropping of components to/from a composite object.

3 Migration

In most object-oriented data models proposed in the literature an object is created as an instance of a class with some attribute values and operations associated with it, and remains an instance of that class till its deletion from the database. This restriction strongly limits the expressiveness of those models. In ORM, it has been partially removed by adding the concept of role, that allows one to deal with the case of an object that plays the same role more than once by the

mechanism of multiple role instantiation, preserving the single object identity. As an example, an object of the class PERSON can simultaneously play the roles of Student and Employee (instantiation) and, later, can lose the role of Employee (suspension). The notion of role, however, does not suffice to model the case of an object that migrates from one class to another maintaining its identity (its oid). This means that a member of the class PERSON cannot migrate to the class ADULT maintaining its oid. In the literature, these aspects of object modeling are classified under the general term of *instance evolution.*

3.1 Instance evolution

Instance evolution may assume different forms. In particular, it is possible :

- to let the object migrate to a different class (the object becomes an instance of the new class);
- to specialize the object, that is, it migrates to a subclass (the object becomes an instance of the subclass, but remains a member of the original class);
- to generalize the object, that is, it migrates to a superclass (the object becomes an instance of the superclass and it is no more a member of the original class);
- to dynamically add new classes to an object, so that it can be an instance of more than one class at the same time;
- to dynamically delete classes from an object;
- to specialize or generalize at instance level adding/redefining/deleting attributes and methods for single objects.

These evolutions are controlled by specific semantic constraints in order to restrict the set of classes where an object can migrate to. For example, referring to the schema of Figure 1, a PERSON can become an ADULT, but he/she cannot become a PROJECT. In [33] those constraints are treated as special integrity constraints, which allow one to specify, for each class, its essentiality or its exclusivity. A class C is *essential* if object migration is constrained on the inheritance hierarchy rooted at C. An object could be member of more than one essential class if the model allows multiple inheritance. A class C is *exclusive* with respect to a class C′ if its instances cannot migrate to C′.

In T-ORM we only support two forms of object migration: object generalization and object specialization. In such a way, object migration is allowed only along a unique class hierarchy. This is not an unacceptable restriction if the data model allows the definition of a common root for all class hierarchies. In that case, using an appropriate combination of generalization and specialization operations, we may allow an object to migrate everywhere. In general, however, object migration does not make sense when it occurs between different hierarchies, because it can involve a complete change of the nature and the structure of an object. For example, it does not make sense to allow a person to become a vehicle. One simple way to avoid the problem of unrestricted migrations is to define different class hierarchies (e.g. one rooted on the class PERSON and one rooted on the class VEHICLE) and maintain them separated. The usefulness of having a common root is advocated in [15]. Accordingly, in the ORION system the class hierarchy forms a direct, rooted, acyclic graph (a DAG), having the system-defined class OBJECT as root. That constitutes one of the schema invariants defined by the ORION model in order to maintain schema consistency after schema updating. For example, when we add a new class to the schema hierarchy without specifying its superclass(es), the new class is added as a subclass of the root class OBJECT. It is worth noting that, even in the presence of a common root, one can still avoid object migration between different hierarchies preventing the migration of objects to pass through the root.

In T-ORM, we asssume to have a number of disjoint class hierarchies, that is, T-ORM classes form a disconnected forest and not a tree.

3.2 Constraints on object migration

The inheritance mechanism requires to impose semantic constraints on object migration operators. Let us assume that there is an object o which is an instance of class C_i and the object migrates to class C_j. Consider the four cases illustrated in Figure 2.

case 1) specialization with single inheritance: non-inherited properties defined for class C_j are added to the object; their values are either provided by the user or considered to be null; the object starts its life cycle as a member of class C_j;

case 2) generalization with single inheritance: all properties that are specific for C_i, i.e., not inherited from C_j, are dropped from the object; the lifespan of object o as an instance of class C_i is terminated;

case 3) generalization with multiple inheritance: all properties which are not defined for C_j are dropped; these properties include all properties which are specific for C_i, and all specific or inherited C_k properties not defined for C_j through inheritance links; the lifespan of o as an instance of class C_i is terminated; the lifespans of o as a

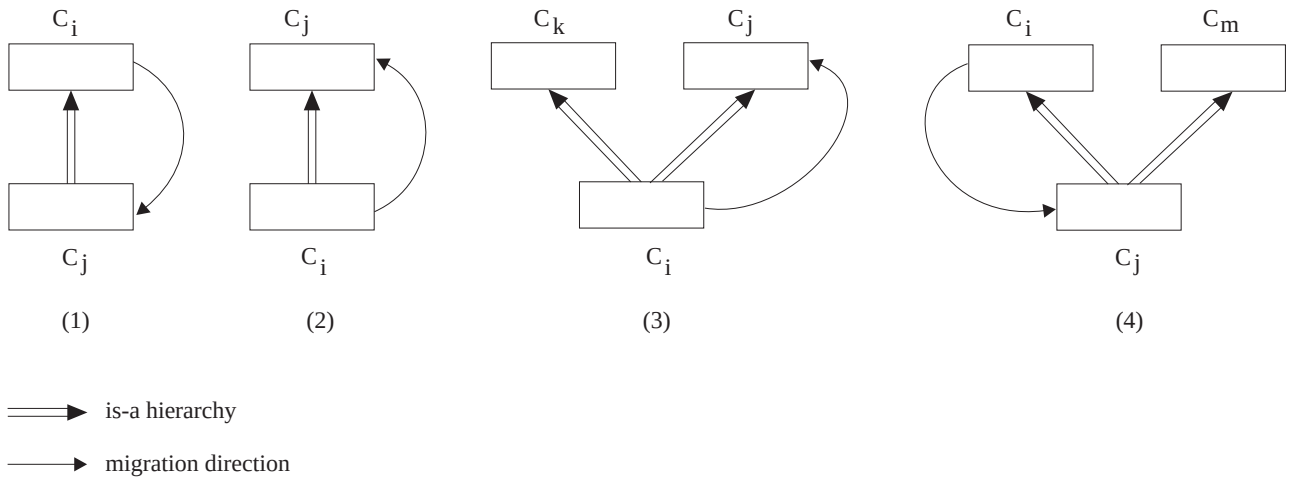

Figure 2: Different cases of object migration

member of class C_k and its ancestors are terminated appropriately, depending on possible inheritance links between C_k and its ancestors and C_j: lifespans in classes belonging also to C_j ancestors are not terminated;

case 4) specialization with multiple inheritance: all properties of class C_j that are inherited from a superclass C_m of C_j, where C_m is not a superclass of C_i, and all properties specific for C_j are added to the object; their values are either provided by the user or considered to be null. The lifespan of o in classes C_i and all its ancestors, excluded C_i and its ancestors, which are already active, are started.

Consistency of data referring to composite objects has also to be examined in view of object migration. In fact most object-oriented DBMS establish that if an attribute has a class C as domain, its values may be all objects belonging to C or to any subclass of C. If an object o instance of a class C is used as value of an attribute A (with domain C) of an object o′, the migration of o to a superclass of C violates the domain integrity constraint of A. In fact object o′, after the migration of o, will have, as a value of A, an object which is neither instance nor member of A′s domain. We remember that an object is said to be an *instance* of a class C, if C is the most specialized class which the object belongs to. An object is said to be a *member* of a class C if it is an instance of C or of a subclass of C. A solution proposed in [33] allows temporary inconsistency and provides a notification mechanism to determine which objects are inconsistent. In these cases, we adopt the same constraints defined above for the deletion of objects, so that inconsistent references must be dropped.

3.3 Storing information about object life cycle

During its lifetime an object can change roles and migrate along the class hierarchy.

As mentioned in Sect. 2, different kinds of temporal information can be associated to objects:

- The object has associated a lifespan for each instantiated role and for each class of which it is (has been) a member, as discussed in section 2.2. We denote with *oid.LIFESPAN(classname)* the lifespan of *oid* as a member of class *classname* i.e., the set of intervals in which *oid* is instance of the class *classname*. Similarly, we indicate with *oid.LIFESPAN(rolename)* the history of instantiations of role *rolename* for a given object indicated by *oid*.

- The *class-lifespan* stores the history of object migration. It is a time sequence representing the various classes the object is (or was) instance of. The value components are sets of the class types the object belongs to, during the associated valid and transaction time intervals. We indicate with *oid.CLASSLIFESPAN* the time sequence representing migration history for object *oid*.

- The *role-lifespan* is a time sequence which represents the union of the lifespans of the single role instances the object has played during its history. The value components in the time sequence are the sets of role identifiers of the active instances of roles in the associated valid and transaction time intervals. We indicate with *oid.ROLELIFESPAN* the role-lifespan of object *oid*.

The object migration mechanism leads us to impose some temporal constraints on the object lifespan. In particular, if class C_j is an ancestor of class C_i, and *oid*

is the identifier of an object which has been member both of C_i and C_j, the following temporal constraint must hold, according to the consistency constraints on migration indicated in the previous section:

$$\text{oid.LIFESPAN}(C_i) \subseteq \text{oid.LIFESPAN}(C_j)$$

Example

Consider the following example of evolution of an object through a series of role instantiations and class migrations (the example is based on the T-ORM schema illustrated in Figure 3 and the history of the object is schematically represented in Figure 4):

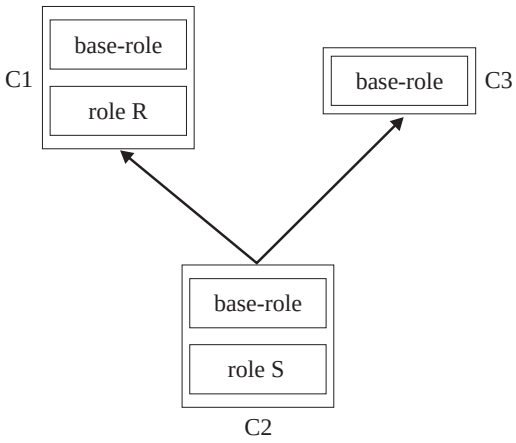

Figure 3: Example of T-ORM schema

- at time t1, the object o1 is created as an instance of the class C_1
- at t2, role R of class C_1 is instantiated the first time as role instance r1
- at t3, role R is instantiated the second time as role instance r2
- at t4, o1 migrates to class C_2
- at t5, role r1 is suspended
- at t6, role r1 is resumed and role S of class C_2 is instantiated as role instance r3
- at t7, role r2 is suspended
- at t8, the object o1 is suspended
- at t9, the object o1 is resumed
- at t10, role r1 is suspended again
- at t11, role r2 is resumed
- at t12, role r1 is resumed

Given the class hierarchy shown in Figure 3, when the object *o1*, instance of class C1, is migrated to class C_2 at time $t4$, its life cycle as an instance of class C_1 continues; in addition, besides starting being an instance of class C_2, it starts also as an instance of class C_3. When the object is suspended at time t8, all active roles are also suspended; roles which were active at the object suspension time are also resumed when the object is resumed.

For the given example, the object lifespan, the role-lifespan, and some of the roles and classes lifespans graphically represented in Figure 4 are shown (for sake of simplicity, only valid times are indicated):

o1.CLASSLIFESPAN = <{C1},[t1,t4)>, <{C1,C2,C3},[t4,t8)>, <{C1,C2,C3},[t9,+∞)>

o1.ROLELIFESPAN = <{r1}, [t2,t3)>, <{r1,r2}, [t3,t5)>, <{r2}, [t5,t6)>, <{r1,r2,r3},[t6,t7)>, <{r1,r3},[t7,t8)>, <{r3}, [t10,t11)>, <{r2,r3}, [t11,t12)>, <{r1,r2,r3, [t12,+∞)>

o1.LIFESPAN(C2) = <[t4,t8),[t9,+∞)>

o1.LIFESPAN(R) = <<{r1},[t2,t3)>, <{r1,r2},[t3,t5)>, ...

r1.LIFESPAN = <[t2,t5), [t6,t8), [t9,t10), [t12,+∞) > <r1,[t6,t8)>, ...

4 Querying T-ORM databases

The complete definition of a data model requires the definition of the corresponding query and data manipulation languages. The goal of querying a temporal database is the retrieval of stored information, taking into account the modifications performed on it. Since bitemporal databases model two temporal dimensions, we can distinguish two basic types of queries: (i) queries that retrieve the sequence of historical values of time-varying information (along the valid time axis); (ii) queries that retrieve data as of a past database state (along the transaction time axis).

In this paper, we focus mainly on queries of the first type that allow us to:

- select an attribute value valid at a given instant, e.g. *find John's salary on 04/15/1986*;
- select an attribute value valid at a time instant associated with another attribute value of the same object, e.g. *find John's salary when Mary was his manager*;
- select an attribute value valid at a time instant associated with another attribute value of another object, e.g. *find John's salary when Mary's salary was $4000*;

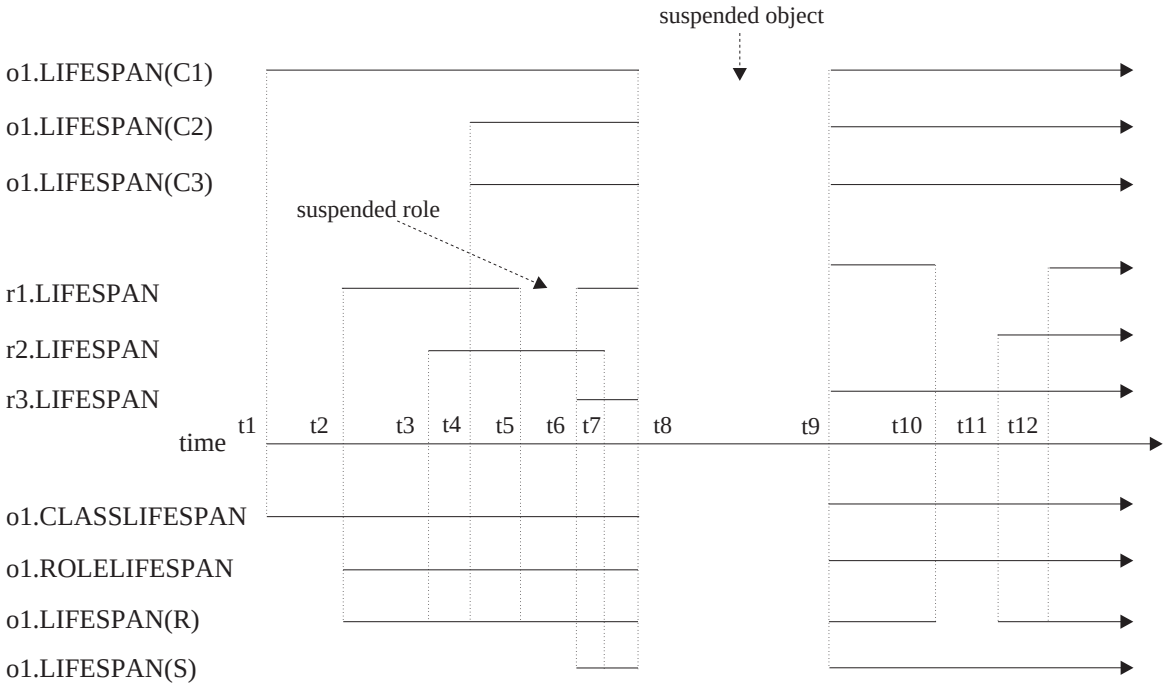

Figure 4: Lifespan dimensions

– select objects stored in the database during a given time interval, e.g. *find all employees in year 1992*;

– select time intervals starting from attribute values, e.g. *find the time period during which Mary was John's manager*.

General aspects of temporal object-oriented data models require special retrieving properties in order to deal with the concepts of class hierarchy, object identifier, complex domain, complex relationship, valid time, transaction time, time intervals, part-of relationships. Some authors attempted to provide a new query language which is compatible with a relational query language (e.g. SQL in the case of IRIS [12]). Other systems, such as ORION, support a new query language which is based on the nested-relational model. Moreover, there are other features which our query language must take into account introduced by the ORM model, such as the concept of role. Due to those new concepts, we define a language which has suitable operators for additional attribute domains (e.g. time), for all kinds of entity compositions and relationships and which allows selecting a portion of object histories.

Query languages for temporal object-oriented databases, like query languages for conventional databases, are divided in two categories: declarative languages and procedural languages. Declaratives languages allow one to describe a query specifying its target and the conditions it must satisfy, without saying how to obtain the result. Procedural languages, instead, use operators to specify a procedure which tells the system how to obtain the result starting from data. The extensions to existing query languages proposed in literature are based both on declarative languages (such as TQuel [29], extension of Quel, and TOOSQL [25], extension of OSQL) and on procedural languages (e.g. the relational algebra [13]). In an object-oriented perspective it is important to abstract from implementation details, so we think that declaratives languages are the best choice. Thus, the language we define is an extension of the query language of the ORION system [15], and is based on SQL syntax.

In the following we focus our presentation on those aspects which are related to object migration and time In the examples, we refer to the schema of Figure 1.

4.1 Basic query structure

A query has the following structure:

RETRIEVE < `target clause` >
FROM < `specification clause` >
WHERE < `qualification clause` >
AS OF < `as-of clause` >

The *target clause* specifies what parts of the selected information must be retrieved, which could be a set of instances (specifying only an instance variable), a time sequence, a set of values, or a sequence of time intervals (points).

The *specification clause* specifies instance variables used in the query, linking them with the correspondent set of object (role) instances.

The *qualification clause* specifies conditions on time sequences to select particular information. In

bitemporal databases we have three dimensions: the data dimension, the valid-time dimension and the transaction-time dimension. The language we are going to define has operators suitable for manipulating all dimensions. In order to maintain the language as simple as possible, we chose to have only one clause (qualification clause) to specify constraints both on the value and the valid-time dimension, whereas other extensions of SQL (such as TSQL [31]) introduce additional clauses.

The *as-of clause* specifies constraints on the transaction-time dimension. It is used to determine the values of object properties as they were recorded sometime in the past and successively revised. In this way we could retrieve information about previous states of the database.

One important element of an object-oriented query language is the facility to express equality between two objects, comparing either their value (value equality) or their oids (object or identity equality). Therefore our query language needs to support both types of equality, which are denoted as == and =, respectively.

Because of the nested definitions of objects arising from the class-composition hierarchy, the T-ORM query language must easily allow the specification of predicates on a nested sequence of attributes. In this respect we adopt the well-known *dot notation* to express paths along the class-composition hierarchy (obtaining what we call *path-expressions*).

4.2 Queries on time-varying properties

Time-varying attributes are the main distinguishing characteristics of temporal databases. Each attribute is modeled with a time sequence which represents all its history (see 2.1). A query language must allow the selection of a portion of that history through the specification of conditions either on time, or on attribute values, or both. We can directly select the first two components of < value, valid-time, transaction-time > triplets in time sequences, with the following notations:

```
e.salary.value
e.salary.vtime
```

A path expression of the kind `e.salary` retrieves the time sequence associated with an instance for the specified attribute (salary). We must provide our query language with operators which allow one to select portions of that history. To do that we can use in the where clause predicates with relational operators involving time. Such operators are those of Allen's interval logic [2] (that is PRECEDES, MEETS, OVERLAPS, STARTS, ENDS, INCLUDES, their inverse and EQUAL), those between time points (i.e. <, = and >) and those between time points and intervals (i.e. BEFORE, BEGINS, ENDS, IN, AFTER and their inverse). For example the following query retrieves all values assumed by the property salary of the instance of the class EMPLOYEE whose name is John, which were valid before 04/10/1987. We assume that the property *name* of class PERSON is not time-varying, so it is not modeled as a time sequence, but it assumes only one value.

EX1: *"Find John's salary before that of 04/10/1987"*
RETRIEVE s.value
FROM (e,Employee)
WHERE e.name == "John"
AND (04/10/1987 AFTER s.vtime)

A path expression which refers to a set of values (such as e.salary) can be quantified using either the existential (EXISTS) or the universal quantifier (FORALL) having the usual meaning. Quantification cannot be made on the variables of the target clause, which are free. We assume that all operators, when applied to a set, distribute on its elements (in the previous example the operator AFTER distributes on the elements identified by s.vtime). Particular elements of a sequence can be selected with the following operators:

- `FIRST(s,e.salary)` $\Longrightarrow$ retrieves the first element in the sequence and assigns it to the variable s
- `CURRENT(s,e.salary)` $\Longrightarrow$ retrieves the current element in the sequence
- `LAST(s,e.salary)` $\Longrightarrow$ retrieves the sequence whose element is the last element in the given sequence
- `<n>-TH(s,e.salary)` $\Longrightarrow$ retrieves the n-th element in the sequence

EX2: *"Find John's current salary"*
RETRIEVE c.value
FROM (j,Employee)
WHERE j.name == "John"
AND CURRENT(c,j.salary)

Our model is based on time intervals, however we could also select the endpoints of intervals using the functions *BEGIN* and *END* which could be applied to a unique interval or to a sequence of intervals, so they return a single time point or a sequence of time points.

Summarizing, we have defined selection operators on time sequences, which act at different levels of detail, as shown in Figure 5.

4.3 Queries on the history of an object

The other important aspect related to time in object-oriented databases concerns the history of an object

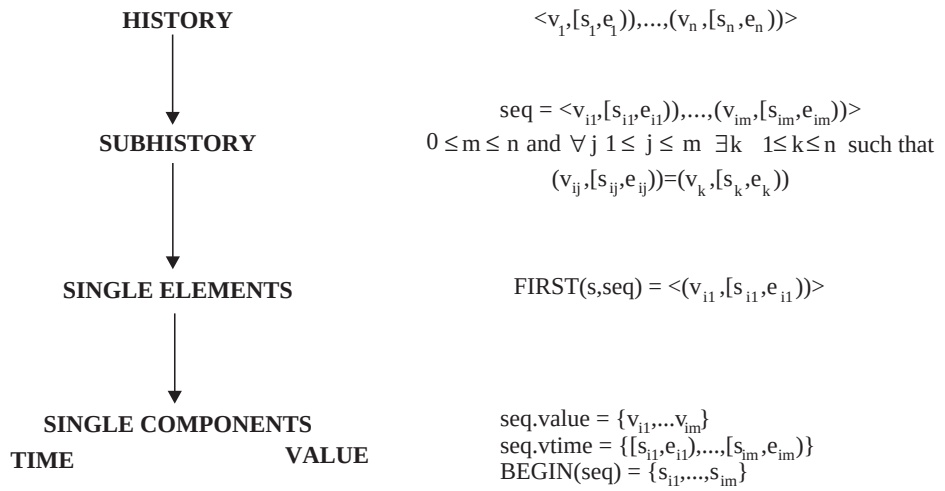

Figure 5: Operators on time sequences

as a whole, in addition to considering the history of single attributes as in the previous section.

In our model, we distinguish between *local histories* and *global object histories*. The local history regards single classes and roles and refers to the variations suffered by the set of their instances. The global object history refers to the previous information viewed from the side of the object, that is it contains the history of its variations as member of various classes and instance of various roles.

4.3.1 Local history

Due to object migration the set of objects belonging to a class can change over time, and therefore some representational primitives are needed to denote the set of instances of a certain class at a specific time point. They allow a user to query the database to know the time interval(s) during which a given object was an instance of a certain class, to know a particular attribute value or the roles played by the object during such a period of time, or whatever else. The T-ORM query language allows one to refer to instances of a class (role) in three different ways, depending on the considered fragment of the class history:

1. the set of all past and current class (role) instances. According to the previously introduced notation, such a set can be identified as follows :

   `(<object-variable>,<class/role name>)`

   EX3: *"Find all employees (now and in the past)"*
   RETRIEVE e
   FROM (e,Employee)

2. the set of instances belonging to a given class during a specific time interval. They are retrieved by means of appropriate conditions on the valid time dimension in the where clause:

   EX4: *"Find John's salary in 1975 when he was an employee and his manager was Mary"*
   RETRIEVE s.value
   FROM (e,Employee)
   WHERE e.name == "John"
   AND EXISTS(s,e.salary):
   ([01/01/1975,12/31/1975]) INCLUDES s.vtime
   AND EXISTS(m,e.manager):
   (m.name == "Mary") AND
   ((m.vtime INCLUDES s.vtime) OR
   (m.vtime OVERLAPS s.vtime) OR
   (m.vtime STARTS s.vtime) OR
   (m.vtime ENDS s.vtime) OR
   (m.vtime EQUAL s.vtime))

3. the current set of class (or role) instances

   `(<object-variable>,` CURRENT`(<class/role name>))`

   EX5: *"Find all employees (now)"*
   RETRIEVE e
   FROM (e,CURRENT(Employee))

If we denote the set of instances of a class (or role) `C` at time `t` with the function **o**(`C`)(`t`), then the following constraint must hold:

$$\texttt{o} \in \mathbf{o}(\texttt{C})(\texttt{t}) \iff \texttt{t} \in \texttt{o}.\mathbf{LIFESPAN}(\texttt{C})$$

and we have:

`(o,`**CURRENT** `C)` returns **o**`(C)(t)` with `t = NOW`
`(o,C)` returns $\cup_{t\in[-\infty,NOW]}$ **o**`(C)(t)`

4.3.2 Migration history

In this paragraph we show how to apply the operators defined for attribute time sequences also to the lifespans time sequences. Remember our representation of object lifespans discussed in paragraph 3.3. We are interested in answering questions of the following type:

EX6: *At which time did Mary become an employee? (role)*
EX7: *At which time did Mary become an adult? (subclass)*
EX8: *During which time period was Mary an employee?*
EX9: *During which time period was Mary an adult?*
EX10: *When did Mary change class?*
EX11: *Which roles did Mary play at 9/6/1994?*
EX12: *Which roles did Mary play during 1994?*

The answer to those questions can be easily found by appropriate queries on the various dimensions of the object lifespan with the use of temporal functions like BEGIN and END. For instance, in query EX10, we select the starting points of valid time intervals from the object class-lifespan, which indicate when a class migration or resuming occurred:

```
EX10: RETRIEVE BEGIN(p.CLASSLIFESPAN.vtime)
      FROM (p,PERSON)
      WHERE p.name == "Mary"
```

In query EX11, we select the role identifiers from the role-lifespan time sequence:

```
EX11: RETRIEVE s.value
      FROM (p,PERSON)
      WHERE p.name == "Mary"
      AND EXISTS(s,p.ROLELIFESPAN):
                (9/6/1994 IN s.vtime)
```

Query EX12 shows an example of using predicates on object history inside the qualification clause:

```
EX12: "Find the salary of the employees who became employees
      before becoming adults"
      RETRIEVE c.value
      FROM (e,Employee)
      WHERE EXISTS
                (s,e.LIFESPAN(Employee)):
                (s.vtime PRECEDES
                FIRST(e.LIFESPAN(ADULT)).vtime)
              AND CURRENT(c,e.salary)
```

5 The Data Manipulation Language

In the DML the operations to create, delete and modify instances have to be extended to involve valid-time specifications. Moreover new operators have to be provided to manipulate the particular features of the extended model such that states, roles and object migration.

5.1 Instance creation

An object is created as an instance of a class. That object could become instance of other subclasses later through the mechanism of object migration. Values for all attributes of that class must be provided by the user, otherwise a null value is assigned by the system. This operation returns the oid of the created object, which can be assigned to an object variable.

If an attribute value is an instance or a set of instances of a class or a role, we can specify those instances directly via their oids (or rids), or indirectly specifying a query. The VALID clause may be omitted. In this case the created object is valid since the time of insertion; an interval whose left end is constituted by transaction time and whose right end is constituted by the value $+\infty$ is inserted in the object lifespan. If only the FROM part is specified, the interval [t1, $+\infty$) is inserted in the object lifespan. Finally if both FROM and TO parts are specified, the interval [t1, t2) is inserted. A time-varying attribute value may be associated with its valid-time. If a validity interval is specified, it must be contained in the validity interval of the whole object. If valid-time specification is omitted, then the attribute value is considered to be valid since time *t1* of the valid clause, or since transaction time if *t1* is not specified, up to time *t2* of the VALID clause, or $+\infty$. If only the FROM part is specified, the attribute value is valid to *t2* or $+\infty$.

In the following example, a PROJECT object is created.

```
EX13: CREATE-OBJECT PROJECT
      WITH (project-name : "P11ts6765",
            participants : {John,Mary} UNION
            RETRIEVE e
            FROM (e,EMPLOYEE)
            WHERE e.manager.name == "Smith",
            reports : {})
```

Roles instantiation is similar to class instantiation, but the user must provide the identifier of the object to be instantiated.

In the following example, the Employee role is instantiated for object *John* with the listed attribute values, and starting from Jan. 1, 1993.

```
EX14: John-empl :
      INSTANTIATE-ROLE (John,EMPLOYEE)
      WITH (salary : 1600000,
            address : "via G. Cesare 57 - ROMA"
            manager : %Smith)
      VALID FROM 1/1/1993
```

5.2 Properties updating

Properties updating is an important issue in temporal databases, because we have the possibility of modify-

ing present, past and future data without losing the old one. Updating is not done directly on stored data, but it is performed by insertion of new components in the object history or, more precisely, in their time sequences. Therefore updating an attribute value requires the selection of one or more time sequence components. The selection is based either on the value component, or on the valid-time component, or both. Finally updating existing values requires the "invalidation" of the old ones, and that is done by acting on the transaction-time component.

We could define two different primitives to update and insert information and impose that when we try to update a value during a time period where there is no correspondent time sequence component, the request will be ignored. But in that case, the user should have a precise knowledge of how data are distributed in time. Instead, we chose to have a unique primitive to update and insert information.

The instance to modify is selected via its identifier or retrieving it with the specification of a query on its property values. The temporal qualification of properties can be omitted. In this case the interval [NOW, $+\infty$) is assumed. If the **TO** part is not specified, then $+\infty$ is assumed.

Let us suppose that P1 is a time-varying property whose value is a time sequence like the following:

$$<(v_1,[VT_{i1},VT_{f1}),[TT_{i1},TT_{f1})),$$
$$(v_2,[VT_{i2},VT_{f2}),[TT_{i2},TT_{f2})),\ldots>$$

First of all we must retrieve all time sequence components whose valid-time interval overlaps $[T_{i1},T_{f1})$ and whose transaction-time interval rightend point is $+\infty$ (i.e. the corresponding value is currently valid). These components must be modified as follows according to five cases.

Let $(v_1,[VT_{i1},VT_{f1}),[TT_{i1},TT_{f1}))$ be such a component, we may have the cases illustrated in Figure 6.

We modify the old value for the portion of interval in the object lifespan which overlaps $[T_{i1},T_{f1})$, for the rest of the interval we insert a new component in the time sequence.

Let us follow in detail case a. The specified valid-time interval partially overlaps a valid-time interval in the time sequence. For the portion of time which overlaps with [VT_{i1},VT_{f1}) the attribute value must be modified, for the part which does not overlap with [VT_{i1},VT_{f1}) a new value is inserted. We must put:

TT_{f1} = NOW (the time sequence component is no more valid)

and insert three new components in the time sequence:

$$(v_1,[VT_{i1},T_{i1}),[NOW,\ +\infty)),$$
$$(val_1,[T_{i1},VT_{f1}),[NOW,\ +\infty)),$$
$$(val_1,[VT_{f1},T_{f1}),[NOW,\ +\infty))$$

As we can note from the figure, after updating we could have two or three contiguous intervals with the same value associated with them. Even if we could collapse the two intervals into one in order to have time sequence components with associated different values, we chose to maintain those intervals separated, because they represent portions of object life which have different histories behind.

The constructs defined for properties updating may be further enriched allowing the specification of operators which calculate the new attribute values starting from the old ones.

5.3 Object migration

In our model, objects can migrate only along the class hierarchy which they belong to, so we consider every class as essential. An object can migrate from a class C to a class C′ which is either a superclass or a subclass of C with the primitives **MIGRATE [UP/DOWN]**.

EX15: MIGRATE DOWN (John,PERSON)
TO ADULT
WITH (driving-license : UD56865G9)
VALID FROM 1/1/1993

In this case a new lifespan for the object identified by John as an instance of ADULT starts at time 1/1/1993 and its lifespan as a member of the class PERSONS goes on.

In the case of migration of an object to a superclass, we must check if that causes the violation of domain integrity constraints for some attributes (see par. 3.3) and delete inconsistent references.

We remember also that in order to maintain the identity constraint of objects, object migration does not change object identifiers. Finally in our model we do not consider a hierarchy between roles, such as introduced in [32].

6 Conclusions

Object-oriented data models have several promising features that make them suitable for being extended with new capabilities. In this paper, we studied a temporal extension of an existing object-oriented conceptual model (the ORM model), focusing our attention on object evolution. The basic features of the proposed approach to object migration do not depend on the particular model we chose, and, in principle, can be extended to any other object-oriented data model. The ORM model was chosen for its particular suitability in representing dynamic aspects of object life. We discussed some alternatives for associating temporal information to attributes, to class membership, and to role instantiations. A query and manipulation language have been defined and discussed, focusing on the constructs provided to manage temporal information.

Some remaining open issues concern the definition of a formal semantics for the T-ORM definition, query

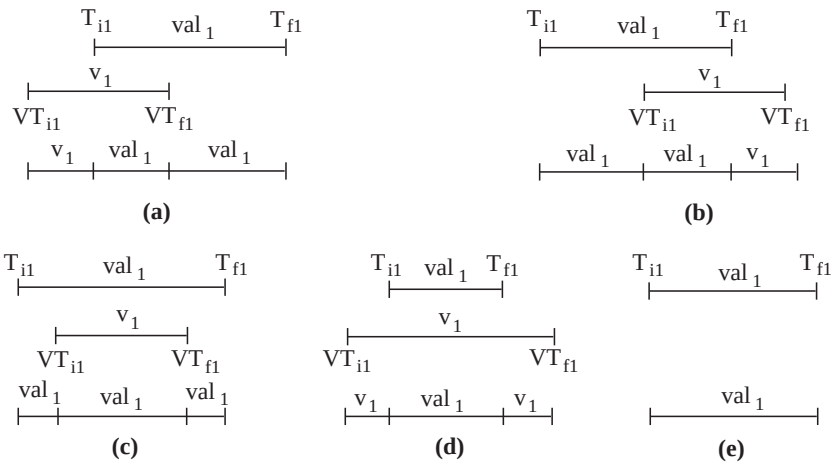

Figure 6: Cases of properties updating

and manipulation languages, and the generalization of the notion of object evolution to deal with changing schemas. Further work is also needed to model temporal aspects in complex objects, such as variations of object composition in time.

Acknowledgments

This work has been partially supported by the P.A.O.L.A. Consortium (Asem Resolutions, INSIEL, and University of Udine) within the project "Sistemi Multimediali per la Gestione del Patrimonio" and by the Italian Consiglio Nazionale delle Ricerche. The authors would like to thank Nina Edelweiss for her useful suggestions. A preliminary version of this paper appeared in [19].

References

[1] Ahn, I., and R. Snodgrass; A taxonomy of time in databases. SIGMOD Record, Vol. 14, 1985, 236-246.

[2] Allen J. F.; Maintaining knowledge about temporal intervals. Comm. ACM, Vol. 26, No. 11, 1983, 832-843.

[3] Bolour, A, and L.J. Dekeyser; Abstractions in temporal information, Information Systems. Vol. 8, No. 1, 1983, 41-49.

[4] Brodie, M.L.; On modeling behavioral semantics of databases, Proc. Int. Conf. on VLDB, 1981, 32-42.

[5] Clifford, J., and A.U. Tansel; On an algebra for historical relational databases: Two views. ACM SIGMOD 1985, 247-265.

[6] Clifford, J., and A. Croker; Objects in time. IEEE Data Eng., Vol. 11, No. 4, 1988, 11-18.

[7] Dayal, U., and G.T.J. Wuu; Extending existing DBMSs to manage temporal data: an object-oriented approach. In [29].

[8] Edelweiss, N., J.P.M. de Oliveira, and B. Pernici; An object-oriented temporal model. Proc. CAISE 93, Paris, Springer Verlag, June 1993.

[9] Edelweiss, N., J.P.M. de Oliveira, E. Peressi, A. Montanari, and B. Pernici; T-ORM: Temporal aspects in objects and roles. Proc. ORM-1, International Conference on Object-Role Modelling, Townsville, Australia, July 1994, 18-27.

[10] El-Sharkawi, M.E., and Y. Kambayashi; Object migration mechanisms to support updates in object-oriented databases. Proc. PARBASE 1990, 378-387.

[11] El-Sharkawi M.E.; Answering queries in temporal object-oriented databases. Proc. Int. Symposium on Database Systems for Advanced Applications, Tokyo, Japan, April 1991, 21-30.

[12] Fishman, D.H. et al.; Overview of the IRIS DBMS, Chapter 10 in [16], 219-250.

[13] Gadia, S.K.; A homogeneous relational model and query languages for temporal databases. ACM TODS, Vol. 13, No. 4, December 1988, 418-448.

[14] Hartmann, T, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch; Revised version of the modeling language TROLL (TROLL Version 2.0). Tech. Rep. no. 94-03, University of Braunschweig, April 1994.

[15] Kim, W., et al.; Features of the ORION object-oriented database system. Chapter 11 in [16], 251-282.

[16] Kim, W., and F.H. Lochovsky (eds.), *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley, New York, 1989.

[17] Montanari, A., and B. Pernici; Temporal Reasoning. Chapter 21 in [31], 534-562.

[18] Papazoglou, M.P.; Roles: a methodology for representing multifaceted objects. Proc. DEXA 1991, Springer Verlag, 7-12.

[19] Peressi, E., A. Montanari, and B. Pernici; T-ORM: evolving objects and roles, Proc. 4th International Conference on Dynamic Modeling and Information Systems, A. Verbraeck, H.G. Sol, and P.W.G. Bots (eds.), Tech. University Delft, Noordwijkerhout, NL, September 1994, 101-119.

[20] Pernici, B.; Objects with roles. Proc. IEEE/ACM Conference on Office Inf. Syst., Cambridge, MA, 1990, 205-215.

[21] Pissinou, N., K. Makki, and Y. Yesha; Research perspective on time in object databases. In [29].

[22] Pissinou N., Snodgrass R., Elmasri R., Mumick I., Oszu M.T., Pernici B., Segev A., Theodoulidis B.; Towards an infrastructure for temporal databases - A workshop report, SIGMOD Record, March 1994, 35-52

[23] Richardson, J., and P. Schwartz; Aspects: Extending objects to support multiple, independent roles. Proc. of the ACM SIGMOD Int. Conf. on MOD, Denver, Colorado, May 1991, 298-307.

[24] Rose, E., and A. Segev; A temporal object-oriented algebra and data model. Tech. Rep. LBL-32013, The University of California, Information and Computing Sciences Division, June 1992.

[25] Rose, E., and A. Segev; TOOSQL - A temporal object-oriented query language. Tech. Rep. LBL-33855, The University of California, Information and Computing Sciences Division, March 1993.

[26] Sciore, E.; Object specialization. ACM Trans. on Information Systems, Vol. 7, No. 2, April 1989, 103-122.

[27] Segev, A., and A. Shoshani; Logical modeling of temporal data. Proc. of the ACM SIGMOD Conference, San Francisco, CA, May 1987, 454-466.

[28] Snodgrass, R.; The Temporal Query Language TQuel. ACM TODS, Vol. 12, No. 2, June 1987, 247-298.

[29] Snodgrass, R. (ed.); Proceedings of the International Workshop on an Infrastructure for Temporal Databases. Arlington, Texas, June 1993.

[30] Su, J.; Dynamic constraints and object migration. Proc. of the 16th Int. Conf. on VLDB, September 1991, 233-242.

[31] Tansel, A.U., J. Clifford, S.K. Gadia, S. Jajodia, A. Segev, and R.T. Snodgrass (eds.); *Temporal Databases: Theory, Design, and Implementation.* The Benjamin/Cummings, 1993.

[32] Wieringa, R., and W. de Jonge; The identification of objects and roles - Object identifiers revisited. Technical report IR-267, Vrije University, Amsterdam, December 1991.

[33] Zdonik, S.; Object-oriented type evolution. In Bancilhon, F., and P. Buneman (eds.), *Advances in Database Programming Languages*, Addison-Wesley, 1990, 277-288.