Object Views and Updates

S. Amer–Yahia and P. Brèche* and C. Souza dos Santos†
INRIA Rocquencourt
Domaine de Voluceau. BP. 105 78 153. Le Chesnay Cedex - France

tel.: (33) 1 39 63 52 75 – fax: (33) 1 39 63 56 74
e-mail: {Sihem.Amer–Yahia,Philippe.Breche,Cassio.Souza}@inria.fr

July 24, 1996

Abstract

We present a mechanism for updating views in Object DataBase Systems (ODBS). Starting from an existing view data model, chronologically in line with [1, 2, 14], we extend it to provide data update ability.
Updates to data views are performed in such a way that consistency with root data is preserved. The main contribution of this paper is to detail such a mechanism together with its corresponding data model.
This is, to the best of our knowledge, the first implementation of such a mechanism in the ODBS framework (as opposed to the RDBS framework). Since this view mechanism is ODMG 93 compliant - views are specified with OQL queries extended by operations on attributes (attribute hiding, renaming and derivation) - it may be implemented on top of various ODBSs. We discuss issues on implementation on top of the O_2 ODBS and consider related work.
Various applications of updatable views conclude the article.

`Keywords:` change management, object views, data update, object database systems.

*On leave from Farbereich 20, Datenbanken und Informationssysteme (DBIS), J.W. Goethe Universität, Frankfurt am Main, Germany.
†On leave from Departamento de Informática, Universidade Federal de Pernambuco, Brazil. Partially supported by CNPq grant number 200.803-92.1.

1 Introduction

Object views can be defined for different purposes such as data customization, data integration, inter-operability between heterogeneous databases and data–schema evolution. Thus, as in the relational model, object views are mainly used to treat different aspects of change management. In the object model, changes can be applied to class structures and methods. Consequently, existing data may be affected and, controlling the impact of changes to persistent objects becomes an important issue [8, 17, 16].

The concept of view, dealing with change management issues, has been introduced by a number of authors working on the object model [1, 8, 22].

Moreover, to promote logical independence, data seen through a view must be manipulated as ordinary data. In particular, updates must be allowed on such data. View updates have been extensively studied in the relational framework and must be revisited in the object context. First steps in that direction have been proposed in [22], providing update primitives for views defined through a particular object algebra. Nevertheless, very little has been said about object view updates and, to the best of our knowledge, no commercial object database has implemented a view mechanism to date.

The main contribution of this paper is to clarify the data update facilities required for object views and to describe how they can be provided.

We show that most view updates cannot be performed without context dependent information, which in turn cannot be expressed with current view definition languages. An existing View Definition Language (VDL) is extended for that purpose. We detail its implementation in O_2Views, a prototype view mechanism implemented on top of O_2.

The rest of the paper is organized as follows. Section 2 introduces a *new* O_2Views model based on the O_2 data model. Section 3 deals with view data updates. Related works are considered in Section 4 and Section 5 concludes and points to future work.

Since this view mechanism is ODMG 93 [5] compliant, it may be implemented on top of other ODBSs.

2 The O_2Views mechanism

In our approach, a view is seen as the customization of a given database schema, called the *root schema* of the view, and associated instance base, the *root base* of the view. A view is specified for particular groups of final users according to specific application requirements. A view is expressed by means of a view definition language, hereafter referred to as *VDL*. This language allows the definition of *virtual classes* derived from classes of the root schema. Virtual classes are populated with objects selected through queries on the root class extensions. Proposed *VDLs* [8, 22, 23] differ slightly according to the underlying view data model and to the particular aspects addressed such as data integration and schema evolution. Our *VDL* is an extension of O_2C, the native language of O_2[15]. It implements the view data model O_2Views, as defined in [2, 14] and is based on the *OQL* query language[5] which is the object–oriented SQL-like query language of the ODMG-93 standard [5].

2.1 Extended O_2 data model

In the following, we introduce the main features of the O_2 object-oriented data model and we proceed with the definition of *object referents* which extend the basic data model. We present an example of a database schema together with an associated base. The O_2Views data model is further presented with a motivating example of a view definition. We assume the reader to be familiar with object-oriented related concepts. Readers interested in obtaining more details are referred to [6].

Data might be *values* (instances of types) or *objects* (instances of classes). Classes *encapsulate* data defined by an underlying class type (tuple type, the C–like atomic types and the constructed types `list`, `set` and `bag`) together with methods which operate on the data. A class might inherit from different super-classes (*multiple inheritance*), and the inheritance mechanism is defined on a *covariant* basis (sub-typing relationship between class and super-class). Persistence is achieved through *naming* as opposed to class extensions. An object is persistent if it is reachable from a name, called the root of persistence. The model is considered to have *extensions* explicitly implemented by names. Persistent objects are accessed through given names.

The ideas above are formalized in the style of [3]. We do not address methods here. We introduce a new concept, that of *object referent*, which extends the O_2 data model. In the original O_2 data model, an *object* corresponds to a pair (o, v) where o is the object identifier (OID) and v its value. In addition, it is commonly accepted that the object identified by o belongs to a single class, that in which it was created, and that it has a unique value.

Definition 2.1.1 [Referent of object]
We define a *referent* as a pair (o, c) where o is an OID and c is a class name. An object now becomes a pair (r, v), where r is a referent of the object and v its value. $\Box$

Referents play the role of OIDs in the original model. Notably, a referent is a value pointing to a peculiar instance of an object *as it is seen* and as it behaves in a given class. This representation allows us to address an object which has a value v in a particular class c. The object is still uniquely identified by its OID, but the same object can belong to many different classes and have different values and behaviors. This is a form of *multiple instantiation*, as opposed to the concept of single instantiation, according to which an object is an instance of one and only one single class, that in which it is created. As will become clear later, multiple instantiation will be used in the definition of multiple points of view over an object: a given object can be instance of the class where it was first created in the root schema, and also an instance of many different unrelated classes in the same view or in different views.

Given the previous definitions, let $R = O \times C$ be the finite set of referents and $\mathcal{V}(R)$ the finite set of values, built up from R, applying the constructors of values [1]. The definition of an O_2 schema remains unchanged.

Definition 2.1.2 [Schema]
Let C be the finite set of class names in a schema, σ be the mapping from C to types over C which associates to each class of the schema its type, and $\prec$ be the sub-classing relationship which is a strict partial ordering among C. This relationship is based on the common sub-typing relationship $\leq$.
A schema $S = (C, \sigma, \prec)$ is *well-formed* iff

$\forall$(c, c') $\in C \times C$, c $\prec$ c' $\Rightarrow \sigma$(c) $\leq \sigma$(c'). $\Box$

[1] It is noticed that $R \subset \mathcal{V}(R)$.

Definition 2.1.3 [Base]
An *instance* B of a schema S, or *base* for short, is a pair (π,ν) where:

- $\pi : C \rightarrow 2^R$

  is a *referent assignment* which associates to each class name in C a finite set of referents such that:

  1. $(o, c') \in \pi(c) \Rightarrow c' = c$
     Two populations associated to two different classes are disjoint.
  2. $(o, c) \in \pi(c) \ \wedge \ (o, c') \in \pi(c') \ \Rightarrow \ \neg(c' \prec c) \ \wedge \ \neg(c \prec c')$
     A given oid occurs at most once along an inheritance path but it can appear in different branches of the hierarchy.

  In addition, the set $\pi^*(c) = \pi(c) \bigcup \{\pi(c') \mid c' \prec c\}$ denotes the *extension* of a class c.
  $R \ = \bigcup\{\pi(c) | c \in C\}$ is the set of referents stored in the base.

- $\nu : R \ \rightarrow \mathcal{V}(R)$

  is a partial function mapping referents in R to their value in $\mathcal{V}(R)$, and giving a value of type $\sigma(c)$ for each referent (o, c).

□

Example 2.1 [The client schema and base]

Figure 1 shows the definition of a database schema using the O_2C syntax. The schema models information about clients of a given telemarketing company. Each client is an instance of the class `Person`. In Figure 2, an associated base of this schema is populated with instances of `Person`. These instances are collected under the root of persistence `Persons` which appears as a set of *referents*. Methods are not included in this specification since we are only concerned with data structure.
□

2.2 Virtual schemas and bases

A *virtual schema*, derived from a given root schema, corresponds to the definition of a view. A *virtual base* is an instance of a virtual schema and maps an instance of the root schema to its image through the view. A *virtual class* is based on a *root class* and allows the customization of its structure and behavior. It is populated with *virtual objects* corresponding to objects filtered out of the root class extension.

A virtual schema is obtained by the extension of a schema S to S', followed by a projection of S' into S''. The virtual base B'' is derived from a root base B instance of S through schema S''. We say that S is the root schema of S''. Figure 3 illustrates these successive operations (triangles and squares represent schemas and bases respectively).

More generally, a view can be seen as a composition of schemas $S_1, \ldots, S_n$ where each schema S_{i+1} is obtained by a transformation on its root schema S_i. This transformation consists in mapping the classes between the two schemas. It is expressed by algebraic operations on objects and on values of the attributes of objects.

```
schema client_schema;
base client_base;

class Person
   public type tuple (
        family_name: string,
        first_name: string,
        age: integer,
        address: Address,
        income: real,
        spending: real,
        spouse: Person,
        children: set(Person))
end;

class City
   public type tuple (
        t_name: string,
        country: string)
end;

type Address tuple (
          street: string,
          city: City,
          zip_code: string);

name Persons = set(Person);
name Cities = set(City);
export schema class Person;
export schema name Persons;
```

Figure 1: An O_2 schema

Base *client_base*		
Class	Name	Objects and Values
	Person	(($oidp_1$, Person), { Platini, Michel , 40, (Av. Clemenceau, ($oidc_1$, City), 57 000), 50 000, 20 000, nil, () }) (($oidp_2$, Person), { Dupond, Marcel, 45, (Rue de_Rennes, ($oidc_2$, City), 75014), 12000 ,7550, ($oidp_3$, Person), (($oidp_4$, Person))}) (($oidp_3$, Person), { Dupond, Raymonde, 42, (Rue de_Rennes, ($oidc_2$, City), 75014), 8500 , 5600, ($oidp_2$, Person), (($oidp_4$, Person))}) (($oidp_4$, Person), { Durand, Juliette , 19, (Bd St Germain, ($oidc_2$, City), 75006), 2000 , 550, nil , () })
Persistence	Name	Extension
	Persons	(($oidp_1$, Person), ($oidp_2$, Person), ($oidp_3$, Person), ($oidp_4$, Person))
	Cities	(($oidc_1$, City), ($oidc_2$, City))

Figure 2: An O_2 database

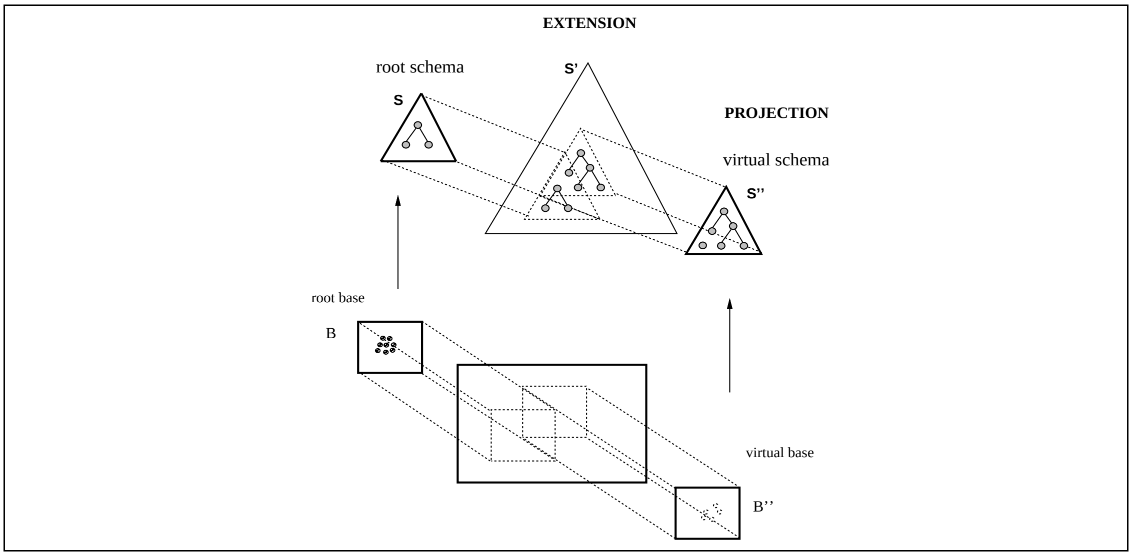

Figure 3: Extension and projection of schema and base.

Definition 2.2.1 [Virtual schema]
Let $S' = (C', \prec', \sigma')$ and $S = (C, \prec, \sigma)$ be two schemas.
S' is an *extension* of S (conversely, S is a *projection* of S' on C), denoted $S \sqsubseteq S'$ iff $(C \subseteq C') \;\wedge\; (\prec'_{|C} = \prec) \;\wedge\; (\sigma'_{|C} = \sigma)$.
The projection S'' of an extension S' of S is a *virtual schema* iff it exists a total injective function $\alpha : C' \to C$ mapping each virtual class to its root class and such that:

1. $\alpha_{|C} = id$
2. $\forall c'_1, c'_2 \in C', c'_1 \prec' c'_2 \Rightarrow \alpha(c'_1) \prec \alpha(c'_2)$
3. $\forall c' \in C'$ it exists a *function* $\aleph_{c'}$ with signature $\alpha(c') \to boolean$.

   Such a function computes the population of a virtual class.
4. $\forall c' \in C'$ there exists a *function* $\mathcal{V}_{c'}$ with signature $\alpha(c') \to \sigma'(c')$. This function computes the values of objects of a virtual class.

□

Definition 2.2.2 [Virtual Base]
Let $B = (\pi, \nu)$ and $B' = (\pi', \nu')$ be two bases.
B' is an extension of B (conversely, B is a projection of B') iff $(S \sqsubseteq S') \;\wedge\; (\pi'_{|C} = \pi) \;\wedge\; (\nu'_{|C} = \nu)$.
The projection $B'' = (\pi'', \nu'')$ of an extension B' of B is a *virtual base* derived from B through S'' iff:

- a virtual class c'' is populated by those instances of its root class, $\alpha(c'')$, which satisfy $\aleph_{c''}$:

  $\pi''(c'') = \{(o, c'') \mid (o, \alpha(c'')) \in \pi^*(\alpha(c'')) \;\wedge\; \aleph_{c''}((o, \alpha(c''))) \;\wedge$
  $\neg\exists\ (c' \prec c'' : \aleph_{c'}((o, \alpha(c'))))\}$

- the value of an object o, instance of a virtual class c'', is computed by $\mathcal{V}_{c''}$ from the value of o instance of the root class $\alpha(c'')$:

$$\nu''(o, c'') = \mathcal{V}_{c''}(o, \alpha(c''))$$

The extension of the virtual class contains only those instances which are not already present in one of its virtual sub-classes. The root object (o, $\alpha(c'')$) is denoted $\beta_{c''}(o)$.
□

By definition, a virtual base is *complete*, i.e. all objects of its root base qualified by a given $\aleph_{c''}$ are in $\pi^*(c'')$ in the virtual base, and *consistent* if all objects in $\pi^*(c'')$ in the virtual base satisfy $\aleph_{c''}$. These two properties together define the *invariant* of the view. In the next section we show how updates to the virtual base can preserve this invariant.

Example 2.2 [An example of an O_2 Views definition]
The telemarketing company decides to promote financial products for a particular category of clients, based on their age. Such clients are selected through an OQL query to compose the population of a virtual class in the view. In addition, since products are to be proposed to these clients, based on specific information (e.g. the amount of taxes they pay, the number of children they have, etc), such information is made available as virtual attributes in the view (e.g. `tax`, `nb_children`). Finally, part of the information visible in the original base must be hidden in the view (e.g. `age` and `income`) and some renaming must also take place (`address` becomes `home`). The O_2Views definition given in Figure 4 takes into account these requirements. OQL queries are used to specify the population of the class as well as the values of virtual attributes.
□

```
virtual schema telemarketing_view from client_schema;

import schema client_schema class Person;
import base client_base name Persons;

virtual class Adult from Person
   virtual attributes
         balance: real = self->base_object->income - self->base_object->spending;
         tax: real = self->base_object->income * 0.2;
         nb_children: integer = count(self->base_object->children) ;
   hide attributes
         first_name,age,income,spending,spouse,children
   rename attributes
         address as home
    with extension Adults =
         select distinct p
         from p in Persons
         where (p.age >= 20);
end;
```

Figure 4: A virtual schema definition

The virtual base `telemarketing_base`, derived from the base given in Figure 2 is shown in Figure 5.

Base *telemarketing_base*		
Virtual Class	Name	Virtual object and its value
	Adult	(($oidp_1$, Adult), { Platini, 30 000, 10 000, (Av. Clemenceau, ($oidc_1$, City), 57 000), 0 }) (($oidp_2$, Adult), {Dupond, 4 450, 1 510, (Rue de_Rennes, ($oidc_2$, City), 75014), 1 } (($oidp_3$, Adult), {Dupond, 2 900, 1 700, (Rue de_Rennes, ($oidc_2$, City), 75014), 1 })
Persistence	Name	Virtual extension
	Adults	(($oidp_1$, Adult), ($oidp_2$, Adult), ($oidp_3$, Adult))

Figure 5: A virtual base

2.3 The O_2Views prototype

The O_2Views prototype implements the model described above on top of the O_2 ODBS. It is composed of a view compiler that generates, for a given view definition, a standard O_2 schema, the view implementation. This schema is composed of a set of O_2 classes together with auxiliary functions and methods implementing the semantics of the view.

The table in Figure 6 shows how some of the main concepts of the view data model are implemented in the prototype. For the virtual class `Adult` defined in Figure 4, for instance, the compiler generates a class `Adult` in the view implementation schema. Auxiliary functions such as `As_Adult` and `Is_Adult` are derived in the schema to allow instantiation of virtual objects. The virtual base can be completely derived (i.e. materialized) or constructed on demand by the application.

3 Updates in O_2Views

In this section, we consider the problem of updating data in a view. We show how data updates specified on the virtual base must be translated into updates on the corresponding root base. The translation must respect completeness and consistency of the virtual base, as specified in Definition 2.2.2. We define basic data update operations on the virtual base, namely *creation, deletion* and *modification* of virtual objects. Creation of virtual

VIEW MODEL	PROTOTYPE	COMMENTS
virtual class	`virtual class`	defines a virtual class
referent $(o, Adult)$	`As_Adult(o)`	conversion function implementing multiple instantiation
$\aleph_{Adult}((o, \alpha(Adult))$	`Is_Adult(o)`	characteristic function filtering objects in a virtual class extension
$\mathcal{V}_{Adult}((o, \alpha(Adult))$	`virtual attributes` `hide attributes` `rename attributes`	the value of a virtual object is built from (renamed) attributes of the corresponding root object plus new derived virtual attributes
$\pi(Adult)$	`Adults`	extension of virtual class
$\beta_{Adult}(o)$	`o->base_object`	root object of an object in the view
extension and projection	`export` and `import`	a view is defined as a schema based on definitions and data imported from other schema(s) and base(s)

Figure 6: Implementation of the view model in O_2.

objects generates the corresponding root objects in the root class whereas deletion of virtual objects removes the corresponding root objects from the extension of the root class. Finally, modification of virtual objects corresponds to changes to values of root objects attributes.

3.1 The update process

Let us consider the class `Adult`, as defined in Section 2.2, with the virtual attribute `balance` derived from the attributes `income` and `spending` of the root class `Person`. Assume the user wants to modify the value of the attribute `balance` for an instance of `Adult` in the virtual base. Values of the attributes `income` and `spending` in the corresponding root object in the class `Person` should be consequently modified. The value of `balance` computed from the new incoming and spending (after propagation) should match the value of `balance` specified in the update. Figure 7 illustrates this situation. Update operations on the objects of views should be committed only if the invariant holds after propagation.

More formally, let us consider B, a database, $V(B)$, a view of this base and u a view update. $u(V(B))$ is the new state of the virtual base after the update u has been performed on $V(B)$.

A *translation* t of an update u takes a base B to a state $B'' = t(u)(B)$. The diagram in figure 8 illustrates how the concept of translation is used for view update propagation. If completeness and consistency, i.e. the invariant of the virtual base, are preserved, the diagram commutes. In this context, updating the view $V(B)$ amounts to choosing, for each update u, one [1] translation t which takes the root base to a state mapping onto the updated view, i.e.:

$$V(t(u)(B)) =_u u(V(B))$$

[1]There might be many!

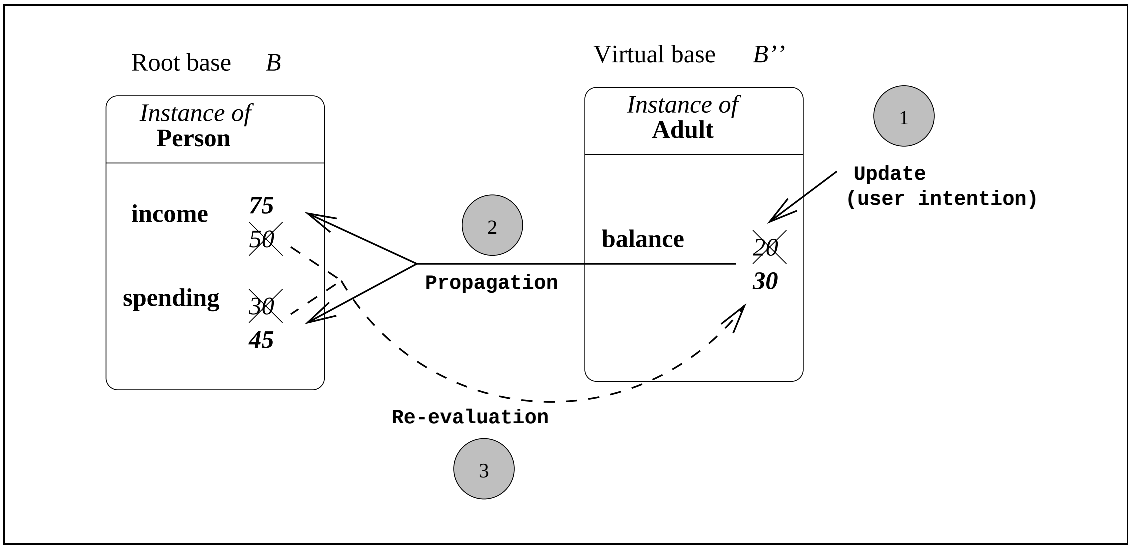

Figure 7: *Example of modification*

Remark: After an update propagation, the view must be re-evaluated, i.e. refreshed, so that side-effect updates can be taken into account. Therefore, the operator "$=_u$" corresponds to an inclusion rather than to an equality relation.

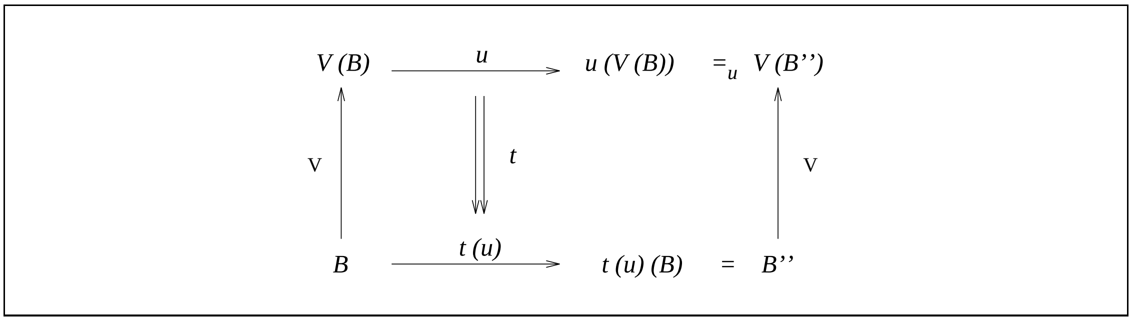

Figure 8: *Translation diagram*

3.2 Update operations on views

We define a *safe update* as an update which can be propagated in a way that preserves the invariant of the view. Definitions of a view and its corresponding root schema are not sufficient to ensure safe view updates. We propose an extension of the *VDL* to solve ambiguities in the translation of updates. This extension allows a translation $t(u)$ to be associated to each view update u. The invariant of the view must hold *after* all propagations specified by $t(u)$ have taken place. In that sense, each update and associated translation is regarded as an *atomic operation*.

3.2.1 Creation

Every virtual object derives from a root object. According to Definition 2.2.2, a virtual object cannot exist without its root object ($o \in \pi'(c') \Rightarrow \aleph_{c'}(o)$).

The creation of an object in a virtual class entails the creation of its root object in the extension of the root class. Moreover, the virtual object must belong to the virtual class c' where it is created, i.e. it must satisfy $\aleph_{c'}$ or, in other words, be in the result of the query defining the extension of the virtual class. When the root object is created, values of virtual attributes must be propagated to the corresponding root attributes in the root object. This propagation defines the initial state of the root object, which, for a virtual class c', must satisfy $\aleph_{c'}$.

The problem of creation is strongly related to the kind of persistence supported by the ODBS. In O_2Views, persistence is achieved by roots of persistence defined in the virtual schema. A root of persistence is an entry point in the database. An object must be reachable from at least one root of persistence to become persistent. As a consequence, newly created virtual objects are added to the extension of their corresponding virtual class. For instance, a newly created instance of virtual class `Adult` must be added to the named extension `Adults`.

To allow the creation of new objects in a virtual class, the view designer must associate a creation operation to the virtual class. We have extended the *VDL* with a creation command, whose syntax is given below:

`on create in class` *class_name* `with` *(parameters)* `do` *code* `enddo ;`

A list of parameters can be associated to the creation operation. These parameters can be assigned either to attributes of the virtual class or to those of its root class. As an example, we consider the following creation operation defined for the class `Adult`:

```
on create in class Adult with (name:string, home:Address)
do { Persons += set(self->root_object);}
enddo;
```

In the action part of the command (clause `do`), the view designer usually specifies to which root(s) of persistence the root object must be attached. Propagation of value parameters follows default rules. The parameter `name`, for instance, is an attribute of the root class `Person`. When the root object is created, this parameter is automatically assigned to the corresponding attribute in the root object. The parameter `home`, in turn, is a virtual attribute. Its value is first automatically assigned to the corresponding attribute in the newly created virtual object and then propagated to the attribute `address` in the root object. Note that the root of persistence `Persons`, to which the created root object `self->root_object` is added, is, as expected, the one used in the query defining the virtual class `Adult`.

3.2.2 Deletion

The deletion of a virtual object amounts to removing it from the extension of the corresponding virtual class. In order to preserve the view invariant, this must be followed by the removal of the corresponding root object from the root(s) of persistence on which the virtual class extension is based, since, by definition, $\aleph_{c'}(o) \Rightarrow o \in \pi'(c')$.

We propose the following syntax for the virtual object deletion command:

`on delete in class` *class_name* `do` *code* `enddo ;`

As an example, we consider the following delete operation defined for class `Adult`:

```
on delete in class Adult
do { Persons -= set(self->root_object);}
enddo ;
```

If an object of class `Adult` is removed from the extension `Adults`, its root object is also removed from the extension `Persons`.

3.2.3 Modification

We distinguish four different types of attributes in a virtual class: *hidden, derived, imported* and *renamed* attributes.

- A hidden attribute is an attribute of the root class that is hidden in the view through the `hidden attributes` clause. Since it is not visible, it cannot be directly updated. Nevertheless, it can be used in the definition of other attributes and thus be indirectly updated as a result of an update propagation.
- A derived attribute is defined from other attributes through the `virtual attributes` clause. Its modification must be propagated to the corresponding root attributes.
- An imported attribute is an attribute of the root class that is neither hidden, nor renamed or redefined as a virtual attribute in the virtual class. Its modification is automatically propagated to the homonymous attribute in the root class.
- A renamed attribute is an attribute of the root class that is renamed in the view through the `rename attributes` clause. Its modification is similar to the modification of an imported attribute up to renaming.

The modification of imported, renamed and hidden attributes is straightforward. In the following, we consider the modification of derived attributes in more detail.

A virtual attribute can be defined through the `virtual attributes` clause of the virtual class creation command, or through a separate command whose syntax is given below:

> `virtual attribute` *attribute_name* `":"` *type_expression*
> `in class` *class_name* `"="` *query_expression*`;`

where:

- *type-expression* corresponds to the type of the virtual attribute;
- *query-expression* corresponds to the OQL query defining the virtual attribute and returns a result of type *type-expression*. A query returns an object or a value. Returned objects are those existing in the root base. Returned values are atomic values or complex values constructed using type constructors (e.g. `set`,`tuple`) on existing values and objects.

As described in [6], OQL is a functional language. An OQL query is a function taking zero or many arguments. The query computing the value of a virtual attribute can be written as follows:

$$av = f(ar_1, ..., ar_n)$$

where each ar_i is a root attribute and f is the OQL query defining av. Let us first consider the case where f takes only one argument, i.e. $av = f(ar)$. A modification u applied to the virtual attribute av gives a new value av', where $av' = u(av)$. This value must be propagated to the root base. The propagation can be implemented in two different ways. A translation t of modification u can be defined such that $t(u)(ar) = ar'$ and $av' = f(ar')$, where ar' is the value of the root attribute ar after modification. The reverse function f^{-1} of the function f, if there exists, can also be used to perform the propagation. In this case, the new value ar' of the root attribute ar is given by: $ar' = f^{-1}(av')$. This is illustrated in Figure 9.

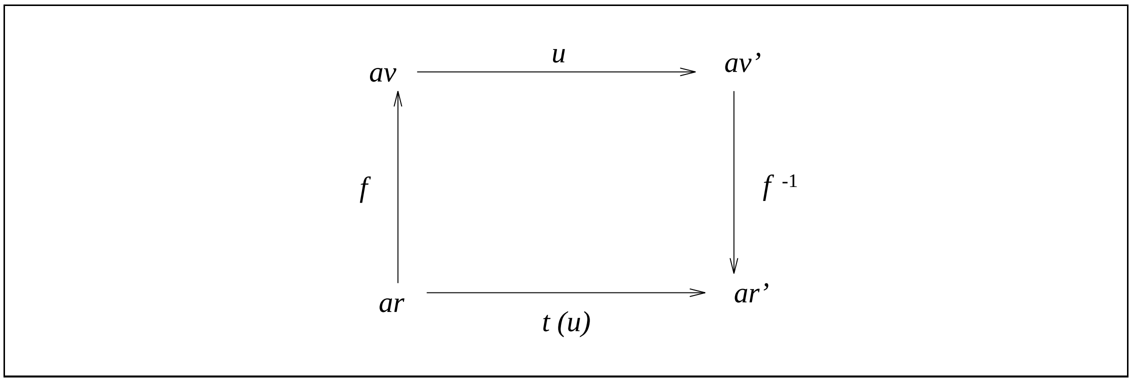

Figure 9: *Modification of a virtual attribute*

To modify a virtual attribute, we consider the following different update policies:

1. To forbid the modification of a virtual attribute when the corresponding reverse function f^{-1} does not exist.

2. To derive the reverse function automatically when it exists and to use this function to propagate the update.

3. To ask the user for providing a translation t associated to the virtual attribute av and to use t to propagate the modification.

In the following, we consider some examples of virtual attribute modifications, for various types of attributes.

Modification of an atomic attribute

Example 3.1 Consider the attribute `tax` in the virtual class `Adult`. This attribute is defined using the attribute `income` of the root class `Person`. Assume a user wants to modify the attribute `tax`. The reverse function can be straightforwardly derived and propagation to the root attribute `income` can thus be performed automatically. □

Example 3.2 Assume now that a virtual attribute `age` is defined in the virtual class `Adult` using the attribute `date_of_birth` defined in the root class `Person`. The modification of the virtual attribute `age` must be forbidden. Given the attribute `age`, it is not possible to compute, i.e. guess, the corresponding `date_of_birth`. □

Example 3.3 Consider the case of the attribute `balance` defined in the class `Adult`. The function f defining this attribute takes two arguments and is not reversible. Nevertheless, the view designer can give a translation of the modification together with the definition of the virtual attribute. One possible translation is:

```
income =  2.5 * balance;
spending= 1.5 * balance;
```

The modification of an atomic attribute depends on the function f defining it. If f is an aggregation function, the modification of the virtual attribute is not allowed. In the other cases, if f is reversible, we can use f^{-1} for the propagation in the root base, otherwise, we must associate a translation $t(u)$ to the virtual attribute. □

Modification of an object attribute

When an attribute is a reference to another object, its modification amounts to either replacing the pointed object by another one or modifying the value of the pointed object. In the following, we consider each case in turn. For each case, we discuss which modification is authorized. These examples show that the modification rights defined on a virtual object should not be associated to its class but to the paths used to access it.

As a first example, let us consider attribute `home`, defined in class `Adult`, which points to an object of class `City`. Each object of this class can be accessed through two different paths. Either from the root of persistence `Cities` or from the entry point `Adults` if at least one object of class `Adult` points to it (see Figure 10). Consider *Marcel* and *Raymonde*, two adults living in the same city *Paris* and an adult *Michel*, living in the city *Nancy*. Assume that *Marcel* moves from *Paris* to *Nancy*. The attribute `home` of *Marcel* must point now to the city *Nancy*. According to the semantics of the application, this can be achieved either by modifying the name of the `City` instance *Paris* pointed by the `Adult` instance *Marcel* or by replacing the pointer to *Paris* by a pointer to *Nancy*.

As another example, consider the virtual attribute `drives` to be added to the class `Adult`. This attribute points to an object of class `Car` representing the car driven by a given adult (for instance, in a car rental application). The modification of the attribute `drives` may consist of either replacing the referenced car by another one (in which case the pointed adult is supposed to be driving another car) or modifying a nested attribute, e.g. the mileage, of the pointed car. Different policies, which are application dependent, can be adopted.

Modification of a collection attribute

Modifying a collection consists in either deleting objects from the collection, adding new objects to the collection or modifying objects in the collection. The last case is equivalent to modifying of an object. The function f defining the value of the virtual attribute can be

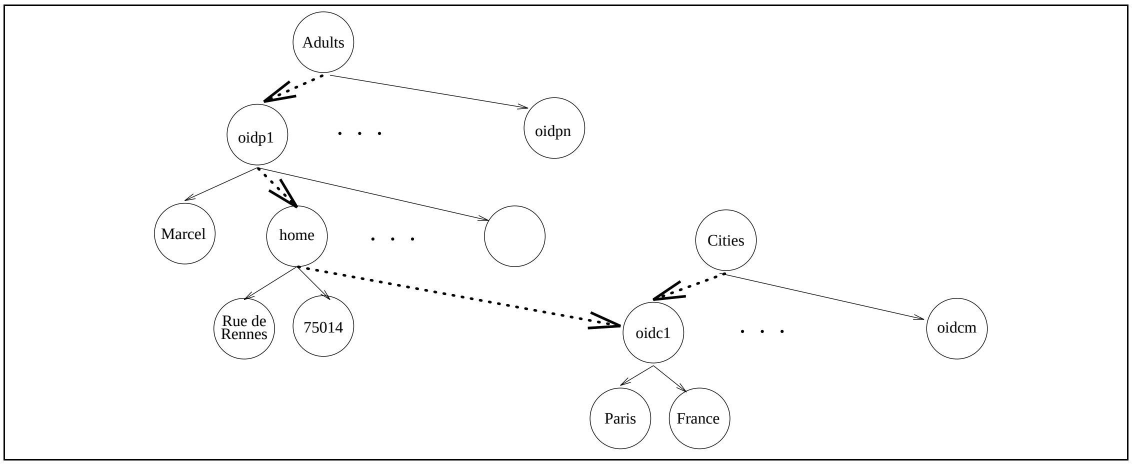

Figure 10: *Example of an object attribute*

of various types. When f is an intersection of two root collections, for instance, there is no ambiguity and all operations performed on the virtual attribute must be propagated to both root collections. If f is a union of two collections, the modification and the deletion of an object from the virtual collection can be propagated to all root collections. However, there is ambiguity in the propagation of an insertion.

Update of collections has been addressed in [22], where a set algebra is used to define virtual class extensions. Ambiguities in [22] are solved by ad hoc rules. We believe that different update policies should be able to be applied according to the specificities of f and the context of the application. In order to solve ambiguities in the propagation of the modification of virtual attributes, translations should be explicitly given by the view designer.

We propose the following syntax for the virtual object modification command:

`on update` *att_name* `in class` *class_name* `do` *translation* `enddo;`

As an example, let us consider the modification of attribute `balance` defined in the class `Adult`. As discussed above, the following modification operation can be associated to this virtual attribute:

```
on update balance in class Adult do
   { self->root_object->income = self->balance * 2.5 ;
     self->root_object->spending = self->balance * 1.5 ; }
enddo ;
```

This translation is used by the system every time an attribute `balance` of an instance of `Adult` is modified. A method called `set_`*att*`(att:`*t*) is generated by the view compiler for each virtual attribute *att* for which an `on update` clause has been defined. This method is used to update the corresponding virtual attribute and it triggers the propagation of the modification to the root base. In addition, the system tests whether the propagation preserves the invariant of the view. If so, the update is validated, otherwise it is aborted.

Finally, the virtual base must be refreshed to take into account update and propagation side-effects. A naive approach consists of re-evaluating the whole virtual base. This would be too expensive. Instead, the system must consider dependencies between view definitions to minimize the cost of view refreshment. For instance, the system can keep information about the set of all root attributes used in the query defining the virtual class extension, in the style of [25]. These and other optimization techniques, which are related to materialized virtual data maintenance, are currently being studied and should be implemented in the near future.

The update of virtual attributes to which an `on update` clause has not been defined or for which the system cannot automatically derive the corresponding inverse function is not authorized.

4 Related work

Regarding change management, the concept of view has been considered in different contexts such as programming languages, knowledge representation systems and databases.

Extensions to object-oriented languages have been proposed to allow different interpretations (or views) to be applied to objects[24, 26]. Persistence, and in particular data updates, are not considered in such approaches.

Knowledge representation systems are more concerned with inference capabilities allowing the classification of concepts (or points of view) in a global taxonomy, as well as the inference of the membership of objects to such concepts [9]. Theoretical results related to complexity and computability of the classification activity have been obtained from knowledge representation investigations, but, again, persistence and updates are not a main issue. Usually, updates can be performed on a very restricted basis: new information can be added but deletions and modifications are either forbidden or subject to several restrictions.

In the relational database framework, much work on update propagation from view relations to base relations [7, 12, 20] and maintenance of materialized views [18, 19, 11] has been done to date. Nevertheless, commercial relational databases systems apply the most simplistic and pragmatic approaches. Updates can be performed only on views defined by simple query expressions, e.g. queries involving only one base relation. This ensures the propagation of view updates but it strongly limits the applicability of relational views.

These limitations can be too restrictive in the object context, where object views are defined to promote data integration and inter-operability. Very little work has been done on object view update to date. In [8], emphasis is placed on the use of views for simulating schema evolution. In this approach, the way identifiers are generated for view objects can be controlled by the user through some special clauses (`identity from` and `OID`) associated to the view creation command. It is argued that this influences the way updates are propagated from views to the underlying root objects, but no more is said about that subject. In [22], a set algebra is defined that is used to specify the extension of view classes. An update operator `update` allows the application of a given update method `m` to a set of objects passed as parameters. Other update operators (`insert` et `delete`) allow the modification of view class extents in a restricted way. In [21], the update semantics for an object model including views is defined. Techniques of automatic classification known from knowledge representation systems are integrated to allow the verification of necessary and sufficient conditions defining constraints on object membership to classes in the view.

Many of the ideas introduced in these domains have influenced our approach, in particular, important notions such as multiple instantiation in knowledge representation systems and applicability of update operations for relational views.

5 Conclusion

In this paper we presented an extended view data model which allows view data updates. Since this view mechanism is ODMG 93 compliant, it may be implemented on top of other ODBSs. We discussed issues on implementation with the O_2 ODBS.

Due to the limited space, we did not address the notions of *imaginary classes*, classes developed to model the occurrence of new objects in a view, and of methods solving, a specific delegation mechanism.

The view mechanism described in this work has proven to be an important tool in different kinds of applications. An O_2 interface to a relational database system has been developed using O_2Views, as described in [13]. O_2Views has been also used to customize O_2 data presentation through the World Wide Web; a prototype being presented at CEBIT'96. In such applications, view applicability would be strongly hindered by limited update facilities. Finally, we consider simulation of schema change through views [10], which strongly relies on the ability to update data in views.

As for implementation, a prototype has been developed [4] which integrates the extensions dealing with data operation updates proposed in section 3.

Acknowledgments: We would like to thank Claude Delobel for his valuable comments and fruitful discussions on the material of this paper.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD'91 Conference Proceedings, Int. Conf. on Management of Data*, volume 20(2), pages 238–247, Denver, Colorado, May 1991. ACM Press.

[2] S. Abiteboul, C. Delobel, and C. Souza dos Santos. Virtual Schemas and Bases. In *Proceedings of the 4th International Conference on Extending Database Technology*, Cambridge, United Kingdom, March 1994.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison–Wesley, 1995.

[4] S. Amer-Yahia. Mise à Jour des Données dans les Vues Objet. Master's thesis, DEA 127, Informatique: Systèmes Intelligents, Université Paris–Dauphine, September 1995.

[5] T. Atwood and al. *The Object Database Standard: ODMG - 94*. Morgan Kaufmann, San Francisco, 1994.

[6] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System — The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

[7] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Trans. on Database System*, 6(4):557–575, 1981.

[8] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Intl. Conference on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.

[9] R. J. Brachman and J. G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9:171–216, 1985.

[10] P. Brèche, F. Ferrandina, and K. Kuklok. Simulation of Schema Change using Views. In N. Revell and A.Min. Tjoa, editors, *Proc. of the 6th Int. Conference on Database and Expert Systems Applications*, number 978 in LNCS, London,United Kingdom, September 1995. Springer Verlag.

[11] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[12] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Trans. on Database Systems*, 8(3):381–416, 1982.

[13] C. Delobel, C. Souza dos Santos, and D. Tallot. Object Views of Relations. In *Proceedings of the 2nd International Conference on Applications of Databases - ADB '95*, San José, California, December 1995.

[14] C. Souza dos Santos. Design and Implementation of Object–Oriented Views. In N. Revell and A.Min. Tjoa, editors, *Proc. of the 6th International Conference on Database and Expert Systems Applications*, number 978 in LNCS, London, England, September 1995. Springer Verlag.

[15] C. Souza dos Santos and E. Waller. *The O_2 Views User's Manual, version 2.0.* INRIA, France, January 1995.

[16] F. Ferrandina, G. Ferran, J. Madec, T. Meyer, and R. Zicari. Database Evolution in the O2 Database System. In U. Dayal, P.M.D. Gray, and S. Nishio, editors, *Proceedings of the 21st Intl. Conference on Very Large Databases*, pages 170–181, Zurich, Switzerland, September 1995. Morgan Kaufman.

[17] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of the 20th Int. Conference on Very Large Databases*, pages 261–272, Santiago, Chili, September 1994. Morgan Kaufmann.

[18] A. Gupta, D. Katiyar, and I.S. Mumick. Counting Solutions to the View Maintenance Problem. In K. Ramamohanarao, J. Harland, and G. Dong, editors, *Proc. of the JICSLP Workshop on Deductive Databases*, 1992.

[19] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinitions. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 211–222, 1995.

[20] A. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. In *Proceedings of the 4th ACM SIGACT–SIGMOD Symposium on Principles of Database Systems, March 25-27, Portland, Oregon*, pages 154–163, Portland, Mar 1985. ACMPress.

[21] C. Laasch and M. H. Scholl. Generic Update Operations Keeping Object-Oriented Databases Consistent. In *Proc. 2nd GI-Workshop on Information Systems and Artificial Intelligence*. Springer, Feb 1992.

[22] C. Laasch, M.H. Scholl, and M. Tresch. Updatable views in object–oriented databases. In *Proc. of the 2nd Int. Conference on Deductive and Object–Oriented Databases*, number 566 in LNCS, Munich, Germany, December 1991. Springer Verlag.

[23] Y-G. Ra and E.A. Rundensteiner. A Transparent Schema Evolution System Based on Object–Oriented View Technology. In *IEEE Int. Conf. on Data Engineering, Tapei, Taiwan*, Vancouver, Canada, March 1995.

[24] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 298–307, 1991.

[25] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.

[26] K. Tsuda, K. Yamamoto, M. Hirakawa, M. Tanaka, and T. Ichikawa. MORE: An Object-Oriented Data Model with Facility for Changing Object Structures. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):444–460, Dec. 1991.