Design and Implementation of Object-Oriented Views*

Cássio Souza dos Santos**

I.N.R.I.A., B.P. 105, 78153 Le Chesnay Cedex, France
tel: 1/39 63 56 79 fax: 1/39 63 53 30

Abstract. We discuss the design and implementation of the O_2Views object-oriented database view mechanism, which allows the redefinition of both the structure and the behavior of objects stored in a database. The data model extended with views is given and the functionalities of the prototype implementing it are presented. The paper focuses on the requirements for the implementation of an object-oriented view mechanism, ranging from the conception of a view definition language to optimization strategies for querying and updating through views.

1 Introduction

It has been agreed that, as for relational databases, views are an important tool for adapting data interface to the needs of different applications in object-oriented databases (OODBs). Despite the number of recent proposals for view mechanisms in OODBs [1, 5, 10, 13, 14, 11], there is very few account on prototype implementations and, to date, no commercial OODB system features views.

In this paper, we discuss the design and implementation of an object-oriented view mechanism in the light of an experience with the design and implementation of O_2Views [12], a view mechanism built on top of the object-oriented database system O_2 [3] in the framework of the Esprit Project GoodStep [7].

The main contributions of this paper are: (1) an account on a running prototype of an object-oriented view mechanism which implements a view definition language and associated graphic interface, (2) a presentation of the formal data model on which the implementation is based, (3) a discussion on how various technical problems related to the implementation of a view mechanism can be solved in the object-oriented context and, in particular, (4) how the concept of *surrogate object* is used to implement the semantics of object restructuring and method resolution in a view.

The paper is organized as follows. Section 2 describes the functionalities of the current O_2Views prototype. Section 3 presents the O_2 data model extended with views in a formal framework. Section 4 presents the view definition language and its implementation and Section 5 finally concludes and gives some directions for future work.

---

* Partially supported by Esprit Project GoodStep.

** CNPq grant number 200.803-92.1, Brazil.

Although it is based on the O_2 data model and on the O_2 system, the discussion in this paper can be generalized to other object-oriented database systems.

2 O_2Views

O_2Views is a tool allowing the definition and implementation of views in the OODB system O_2 [3]. Once defined, a view can be used for querying and updating the database through applications written on top of it. The tool has been implemented using the O_2C programming language, the O_2 native language, on top of the O_2 system.

In this section, we give an overview of the functionalities of O_2Views. A more complete description is given in [12]. We assume the reader is familiar with O_2 basic concepts (e.g. schema, class, method, inheritance).

A view in O_2Views is defined as a *virtual schema* derived from another (real or virtual) schema, called the *root schema* of the view. Views can therefore be composed and nested to any degree. A *virtual base* corresponds to the image of a *root base* (a base instantiated from the root schema) through the view. Figure 1 illustrates the relationship between virtual entities and their corresponding roots for a given virtual base derivation.

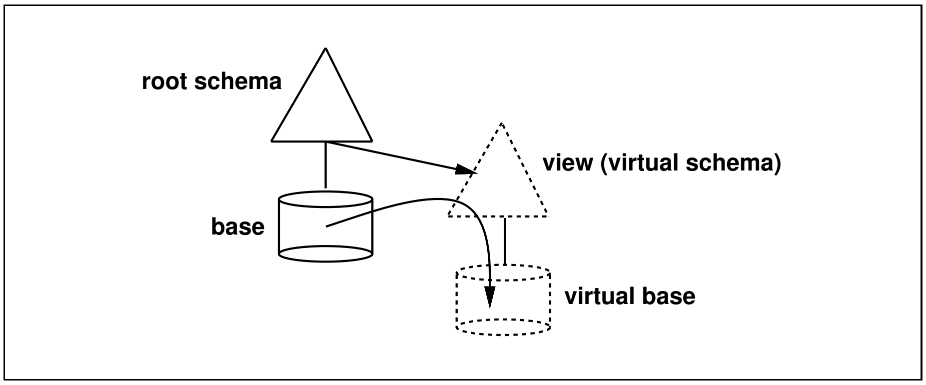

Fig. 1. Relationship among schemas and bases in a view derivation

For those familiar with the deductive database terminology, virtual schema definitions correspond to intensional predicates defining an intensional database (the virtual base) whereas real schemas and bases correspond to extensional predicates and bases respectively. The main primitives of the view definition language (hereafter the O_2Views language) are virtual classes, imaginary classes, virtual attributes and hidding.

A virtual class selects through a query objects existing in the root database and provides a named set containing these objects: the extension of the virtual class. Moreover, since it is a class, its interface can be modified through the view primitives for hiding or adding attributes (structure modification) and the (re)definition of methods (behavior modification). Instances of a virtual classes, called virtual objects, are thus objects existing in the root database with a possibly different interface in the view.

An imaginary class selects and restructures through a query data from the root database or the view and turns them into new objects. The oid of an imaginary object is a function of its *core attributes*. Instances of an imaginary class, called, imaginary objects, exist only in the virtual base during view activation.

A virtual attribute attaches (possibly restructured) data to an object in the view, through a query on the root database or the view. It augments the original interface of (virtual) objects. It also attaches to imaginary objects data not involved with their identity (i.e. attributes other than those declared as core attributes).The hidding of attributes restricts the interface of virtual objects in the view and also allows the hidding of core attributes of imaginary objects that are meant to be used only as internal keys.

Figure 2 gives an example of a virtual and an imaginary class. Class `Person` is defined in the root schema `person_schema`, whose presentation is ommited here.

```
virtual schema example from person_schema;

virtual class Adult from Person extension Adults
    virtual attributes
      age: integer has value
         self->get_age;
    hide attributes date_of_birth
    includes
      (select p from p in People where p->get_age ≥ 21)

imaginary class Family
    core attributes husband:Person,wife:Person
    extension Families
    virtual attributes
      children: set(Person) has value
            self->husband->children union self->wife->children;
    includes
      (select tuple(husband:h,wife:w)
       from h in People,w in People
       where h->is_married_to(w) and h->is_male)
```

Fig. 2. Example of a virtual and an imaginary class

As mentionned above, an instance of `Adult` correspond to an instance of `Person` (they share the object identifier) whereas instances of `Familiy` do not exist in the root database and can be seen only through the view.

Once defined, a virtual schema definition is compiled and stored into a view repository. O_2Views can thus generate the implementation of such view, which is standard O_2 C code for a schema with compiler generated classes, methods and functions implementing the view definition. This schema is called the *view* *implementation schema*. It can be used as any ordinary O_2 schema and different applications can be defined on top of it. This is illustrated in Figure 3, where gray circles represent modules implemented by O_2Views. Application programmers are not necessarily view designers and they possibly do not know they are working on a view instead of a real database.

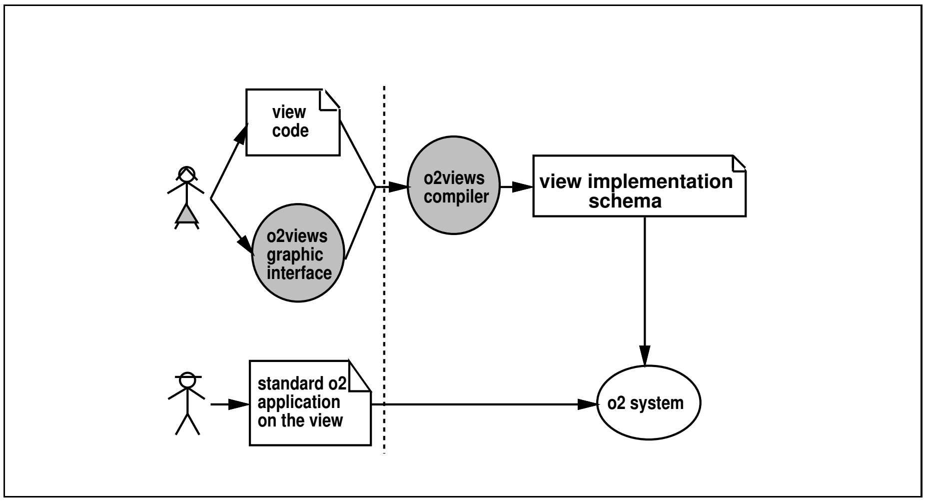

Fig. 3. Using O_2Views

A base instantiated from a view implementation schema corresponds to a virtual base. It does not contain data of its own but is an image of another base through the view.

3 The View Data Model

In this section, we extend the formal presentation of the O_2 data model given in Chapter 3 of [3] with the main concepts introduced in the previous section. We first present the basic model without views, which is a condensed version of the original presentation given in [3] and then proceed with the extended data model featuring virtual and imaginary classes. Some aspects such as attribute hiding and virtual attributes are not considered here. Modularization aspects such as importing, exporting and visibility of definitions among schemas are also ignored in this presentation, due to space limitations.

Core Data Model

We consider the set $\mathbf{D}$ of atomic values, the set $\mathbf{A}$ of attribute names, the set $\mathbf{I}$ of object identifiers and the set $\mathbf{C}$ of class names. The definition of value remains unchanged with respect to that given in [3], where tuple, list and set constructors are defined, as well as that of values over I, a subset of $\mathbf{I}$, which is denoted by $\mathbf{V}(I)$. In particular, $\mathbf{V} = \mathbf{V}(\mathbf{I})$. An object is a pair (i,v), where i is an identifier of $\mathbf{I}$ and v is a value of $\mathbf{V}$.

$\mathbf{T}(C)$, where C is a subset of $\mathbf{C}$, denotes the set of all types over C together with the special name **any** (a default type, the type of class `Object`). $\mathbf{T}$ denotes $\mathbf{T}(\mathbf{C})$. The definition of the subtyping relationship $\leq$ on $\mathbf{T}(C)$ is standard.

A *class hierarchy* is defined as a triple $(C,\sigma,\prec)$, where C is a finite set of class names, σ is a mapping from C to $\mathbf{T}(C)$ and $\prec$ is a partial order on C which defines the inheritance relationship. A class hierarchy is well formed if for all classes c and c' in C, if $c \prec c'$ then $\sigma(c) \leq \sigma(c')$.

A *disjoint oid assignment* π_{d} for each class name c in C is a function mapping c to a disjoint set of object identifiers. This models the fact that each object is assigned to a most specific class, where it is created. The mapping π inherited from π_{d} is such that $\pi(c) = \pi_{\mathrm{d}}(c) \cup \{\pi(c')|c' \in C, c' \prec c\}$. Thus, for pairs of classes (c,c') such that $c \prec c'$, we have $\pi(c) \subseteq \pi(c')$. We say that $\pi(c)$ is the *extension* of class c.

Given the usual definition of the domain function $dom(t)$ of a type t in $\mathbf{T}(C)$ (for details on its definition, see [3]), for all types t and t' in $\mathbf{T}(C)$ such that $t \leq t'$, we have $dom(t) \subseteq dom(t')$.

A database schema S is a 5-tuple $(C, \sigma, \prec, root, M)$ such that $(C,\sigma,\prec)$ is a well-formed class hierarchy and M is a well-formed set of method signatures. The fourth component, namely *root*, relates a virtual class to its root class (here we consider only virtual classes with a single root class). It is empty in a real schema (since there are no virtual classes) and is defined for allowing a uniform definition of view composition.

An instance of a database schema S, or a base for short, is defined as a triple (π,ν,μ), where

- π is an oid assignment for the schema and $I = \bigcup\{\pi(c)|c \in C\}$.
- ν is a mapping from pairs (object identifiers,class) to values, i.e. ν is a partial function from $I \times C$ to $\mathbf{V}(I)$. In a real base, (our concern so far), ν maps (o, c) for o in class c, to the value of o. For all $o \in \pi(c)$, $\nu(o, c) \in dom(\sigma(c))$.
- μ is a mapping from method signatures in M to finite input-output relations.

Resolution is defined using the partial order $\prec$ in the standard way, with method overloading and covariance of well formed method signatures as defined in [3]. More precisely, the semantics of $m(o)$ in a class c is defined by:

- if M contains a signature α of the form $m : c \times t_1 \times ... \times t_{\mathrm{n}} \rightarrow t$ it is $\mu(\alpha)(o)$;
- otherwise, we use the semantics of m in some superclass of c (conventional method lookup). If m is undefined in all superclass, m is undefined in c, and if m is multiply defined (multiple inheritance conflict), the schema is inconsistent.

Extended Data Model

In a view, i.e. in the virtual schema, we have to specify the virtual classes, the type of the values of objects in these classes, the nature of the objects in these classes, i.e., the existing classes where they come from, the signatures of the new methods. The extensions of virtual classes are specified using characteristic functions and the values of the new objects by special methods.

Given a schema $S = (C, \sigma, \prec, root, M)$, a view S' of S is a 5-tuple $(VC \cup IC, \sigma' \cup \sigma_{\text{core}}, \prec', root', M')$ where:

- VC is a set of virtual class names;
- IC is a set of imaginary class names;
- σ' is a mapping from $VC \cup IC$ to $\mathbf{T}(C \cup VC \cup IC)$;
- σ_{core} is a mapping from IC to $\mathbf{T}(C \cup VC \cup IC)$ such that for all c in IC, $\sigma_{\text{core}}(c) \leq \sigma'(c)$. It gives the core type of the imaginary class, which is a subset of its attributes that determines identity of its instaces.
- $root'$ relates each virtual class in VC to a class in $C \cup VC \cup IC$ (the root classes of a virtual class);
- $\prec'$ defines a partial order on VC where:
  - $\forall c_1, c_2 \in VC, c_1 \prec' c_2 \rightarrow root'(c_1) \prec^* root'(c_2)$ (this ensures inheritance in the virtual schema to be consistent with respect to inheritance in the root schema, i.e. there is no inheritance conflict);
- M' is a set of well formed signatures over $C \cup VC \cup IC$ containing, in addition to user defined methods in virtual and imaginary classes, the following *system-supplied functions*:
  - for each c' in VC:
    - $\aleph_{c'} : root'(c') \rightarrow bool$;
    - $val_{c'} : root'(c') \rightarrow dom(\sigma'(c'))$;
  - for each c' in IC:
    - $\aleph_{c'} : \sigma_{\text{core}}(c') \rightarrow bool$;
    - $encode_{c'} : \sigma_{\text{core}}(c') \rightarrow \mathbf{I}$;
    - $val_{c'} : dom(\sigma_{\text{core}}(c')) \rightarrow dom(\sigma'(c'))$.

$\aleph_{c'}$ and $val_{c'}$ are the characteristic and the value constructor functions associated to each virtual and imaginary class c'.

To specify an instance of a view, i.e. a virtual base, we only have to provide (i) an instance of the root schema and (ii) the semantics to the methods in M' (the code of the view) since there is no data stored in a view.

Let $B = (\pi, \nu, \mu)$ be an instance of some schema S and S' a virtual schema defined from S. The image of B through S', i.e. the virtual base B', is simply defined by a mapping μ' from M' to finite input-output relations of appropriate types. This definition may seem somewhat puzzling since the instance that we obtain is rather far from the original definition of an instance. However, let us now reexamine the components of a standard instance in the context of the view. We obtain an instance $B' = (\pi', \nu', \mu')$ where:

- for each c' in VC, $\pi'_{\text{d}}(c') = \{o \mid o \in \pi(root(c')) \wedge \aleph_{c'}(o))\}$;
- for each c' in IC, $\pi'_{\text{d}}(c') = \{encode_{c'}(v) \mid v \in dom(\sigma_{\text{core}}(c')) \wedge \aleph_{c'}(v)\}$;
  In addition, for each c' in IC, $\forall c \in C : \pi(c) \cap \pi'_{\text{d}}(c') = \emptyset$, i.e. instances of imaginary classes are created in the view;
- for each c' in $VC \cup IC$ and o in c', $\nu'(o, c') = val_{c'}(o)$;

- μ' is a mapping from method signatures in M' to finite input-output relations;

Resolution is defined using inheritance and function *root* for virtual classes. More precisely, the semantics of $m(o)$ in a class c is defined by:

- if $c \in VC \cup IC$:
  - if M' contains a signature α of the form $m : c \times t_1 \times ... \times t_n \rightarrow t$, it is $(\mu'(\alpha))(o)$;
- otherwise, if $c \in VC$:
  - we use the semantics of $m(o)$ if it is defined in some virtual superclass of c (conventional method lookup), and if it is multiply defined, the schema is inconsistent;
  - otherwise, we use the semantics of $m(o)$ in class $root(c)$ where it belongs (resolution in the root schema) and if it is undefined in $root(c)$, m is undefined in c.
- otherwise, m is undefined in c.

Remark: The definition above states that every method resolution on virtual objects that cannot be accomplished in the view is tried in the corresponding root context. This can be viewed as a form of *instance inheritance* (*delegation*) [8]. Finally, method (re)definitions in virtual (super)class(es) have precedence over those in the root hierarchy.

4 The O_2Views Language

The O_2Views language is an extension of O_2C, the native language of O_2. Its design was based on the following guidelines:

- Avoiding disfunction between O_2C and O_2Views: The O_2Views language is a conservative extension of O_2C. In particular, the typing system of O_2C was not changed.
- Use of OQL as a basis for specifying computations in the view: OQL queries are encapsulated into some of the view language primitives and brings declarativeness to view specifications;
- Neat separation between real and virtual words: Real schemas describe the structure and behavior of persistent data only and therefore are not allowed to see (import) definitions from virtual schemas View primitives can be used only in the definition of virtual schemas.

A complete presentation of the language is given in [12]. Here we comment on the commands illustrated in Figure 2, where a virtual schema `example` is defined from the root schema `person_schema`. Definitions in the root schema (classes, types and names) are all visible in the body of a virtual schema definition but are not part of the virtual schema interface itself, unless explicitly stated through an import clause.

The virtual class `Adult` is created from root class `Person`, declared in the `from` clause. The name through which the extension of the virtual class is accessed in the view is given in the `extension` clause. If no extension name is given, a default name is provided.

There are two ways of defining the population of a virtual class, through a query expression (clause `includes`) or through a boolean expression (clause `if`). The query expression (called the *defining query*) defines the population of the virtual class and must return a unique set of instances of the root class (or its subclasses, if any). The boolean expression is an O_2C boolean expression (like in an O_2C `if` statement) and gives the condition that an instance of the root class must satisfy in order to be an instance of the virtual class (the *characteristic function* of the virtual class). If neither the `includes` nor the `if` clause are used, a default boolean expression `true` is assumed.

Methods (behavior) and attributes (structure) are "inherited" through *delegation* from the root class. In the current implementation, each instance of a virtual class holds a pointer to an instance of its root class (which can be real or virtual). Methods and attributes which are not defined or redefined in the virtual class are thus automatically delegated to the root object. Such *instance inheritance* mechanism is automatically provided so as to allow view objects to share behavior with their corresponding root objects.

Virtual attributes are defined with a name, a type and a query expression. The query must be of the declared type and specify the value of the virtual attribute.

Imaginary classes are defined as illustrated in Figure 2. They don't have a root classe (the `from` clause in the virtual class definition) and they have an additional `core attributes` clause. Their defining query must return a unique set of tuple values whose type is that defined by the core attributes. To each tuple value returned by the query, a new object identifier is associated, so that the tuple values are "objectified". No two objects are created for the same tuple value.

View definitions can thus be composed to any extent (intra-view and inter-views).

The Implementation of O_2Views

A requirement of the current O_2Views was to run on top of O_2 without any change to the O_2 engine, i.e. the object and schema managers were left unchanged. Views were thus implemented by standard O_2 objects and using the standard O_2 semantics for typing, interface resolution, query evaluation and object management. Apart from limitations on the performance, this has also limited the functionality of the language *as it is implemented in the prototype.*

O_2 offers a very rich data model. The following O_2 features turned out to be very useful in the implementation of our view mechanism:

- powerful declarative query language with associated object-preserving algebra (allowing the retrieval but not the creation of objects)

– separation between class and type concepts
– automatic object identity management (oid creation and garbage collection)
– modularization mechanism for sharing definitions among schemas and data among bases

We decided to use the query language to specify the population of classes in the view, instead of relying on characteristic functions, as described in the data model in Section 3. The power of expression of OQL, which is enhanced by the fact that O_2C methods and functions can be invoked from a query expression, is satisfactory for the definition of virtual classes and allows class population to be specified in a declarative and functional style. Populations involving more complex computations (e.g. recursive queries and transitive closures) can be coded into external O_2C functions and called from an OQL expression. The characteristic functions ($\aleph_c$ for each virtual class c) can in turn be derived from the defining queries.

The separation between type and class allows these two concepts to be freely combined and this possibility is exploited in the current implementation, where the subtyping relationship is relaxed between virtual and corresponding root classes.

Some required features are not directly supported by O_2 . In particular, multiple class instantiation (objects belonging to many different classes), system maintained class extensions and union types had to be simulated in the current prototype, as it will be described in the sequel.

We will not be able to describe the whole implementation due to space limitations. Instead, we will focus on the implementation of virtual classes.

As discussed before, a virtual class allows the redefinition of both the structure and behavior of existing objects, so that they can be seen and behave differently when accessed through a view. A virtual class is defined from a *root class* (which in turn can be a real or virtual class) and its instances are actually instances of its root class whose interface (attributes and methods) change during view activation. Thus, for each instance of a virtual class, there is a corresponding instance of its root class and these two related instances share the object identity.

O_2 does not support multiple class instantiation, i.e. an object cannot be an instance of two classes. In a view, however, we want to be able to temporarily assign alternative properties to objects. In our implementation, this is achieved by *surrogate objects*. A surrogate object is an instance of a virtual class which points to its corresponding root object (the corresponding instance of the root class of whom it is an image through the view).

Virtual classes are implemented as immediate subclasses of an abstract class called `Virtual_Object`. This special class implements the functionalities common to all virtual objects, which are explained below.

A special relationship, orthogonal to the inheritance one, relates virtual classes to their corresponding root classes. The inheritance relationship is usually called the *is-a* relation (an instance of a class *is an* instance of its superclass(es), and the relationship between a virtual class and its corresponding root class can be seen as a *may-be* relation, in that an instance of the root class *may be* an instance of its corresponding virtual class(es), depending on whether it satisfies the predicate stated through the defining query of such virtual class(es), as specified in Section 3.

An instance of a virtual class points to an instance of its root class through the private attribute `root_object` defined in class `Virtual_Object` with type `Object`, the topmost class in the class hierarchy. This attribute holds a pointer to the root object and its type is redefined in each subclass of `Virtual_Object` to be the root class of the corresponding virtual class.

Let us consider the virtual class `Adult` in Figure 2. Class `Adult` is implemented as a subclass of `Virtual_Object` in the view implementation schema generated by the compiler, as shown in Figure 4.

```
class Adult inherits Virtual_Object type
tuple (private root_object:Person,
     read name:string,
     read age:integer)
methods
     public is_consistent:boolean,
     public update_attributes:boolean,
     public is_equivalent_to(obj:Object):boolean,
     public set_name(name:string),
     .... /* plus all public and read methods which are propagated
          from from Person and delegated to the root object */
end
```

Fig. 4. Implementation of Virtual Class `Adult`

By keeping virtual classes in a different hierarchy (or, more generally, in a separate schema), we stick to our philosophy of separating virtual and real words and avoid various technical problems. First, there is no type mismatch, i.e. there are no constraints on typing between a class and its virtual class(es). Another incompatibility is related to the class extension containement. Sometimes, the type of a virtual class may be a supertype of the type of the root class but its extension is a subset of the extension of the root class. In a well formed class hierarchy, subtyping and set containement between class extensions go together, since the extension of a class is defined as a subset of the extension of its superclass(es).

A virtual object is instantiated from its corresponding root object. An instance of `Adult` can thus be created by applying the system supplied function `As_Adult()` (system supplied functions are generated by the O_2Views compiler), which takes an instance of `Person`, the root class, as parameter. This function tests whether the instance of `Person` can become an instance of `Adult`, i.e. if it satisfies the predicate derived from the defining query (the characteristic function), and returns `nil` if the predicate is not satisfied for that object.

The functionalities provided by class `Virtual_Object` through its methods and associated auxiliary functions are summarized below.

The convertion function `As_`$\mathit{Virtual\_Class}$`(obj:`$\mathit{Root\_Class}$`)`: $\mathit{Virtual\_Class}$ allows the instantiation of a virtual class from an instance of its root class. The convertion function incorporates the method $val_{Virtual\_Class}$ of the data model, but the value of the virtual object is encapsulated into a new object. Equivalence (or virtual equality) between this new object and the root object can be tested using the equivalence method given below).

The characteristic function `Is_`$\mathit{Virtual\_Class}$`(obj:`$\mathit{Root\_Class}$`):boolean` tests the membership of an instance of the root class in the virtual class, i.e. if the root object satisfies the predicate stated in the defining query. Efficient test functions can be derived from the defining query which allow us to avoid query evaluation to a large extent.

The consistency check method `is_consistent:boolean` tests whether a virtual object has become inconsistent due to an update to the corresponding root object (side-effect) and update method `update_attributes` updates the values of virtual attributes. These can become obsolete due to side effects. For each public attribute *att* of type t "inherited" from the root class, a public method `set_`*att*`(att:`t`)` is provided which allows the update of an attribute in the virtual object and the propagation of such update to the root object.

Method `is_same_as(obj:Object):boolean` tests whether two given objects are different points of view on the same real object. In the current implementation, the identifier of a virtual object is not the same as that of its corresponding root object. In that sense, the virtual equivalence method is the equality operator between objects in the view.

By relating virtual and corresponding root classes through the *may-be* relationship, we avoid the creation of auxiliary intermediate classes which have to be generated to allow the integration of virtual classes into the inheritance hierarchy, as described in [9]. From a philosophical point of view, integrating virtual classes into a single inheritance hierarchy amounts to define a single taxonomy which encompasses the whole set of concepts (in the spirit of the KL-ONE classification algorithm), whereas relating such classes to the inheritance hierarchy through an orthogonal relation gives these classes a different conceptual status, making them not concepts themselves, but different points of views (possibly exceptional and therefore involving type incompatibility, as discussed in [6]) of existing concepts instead.

The implementation of imaginary classes is similar to that of virtual classes. In the current implementation, and, we insist, this is a consequence of simplifying assumptions only, a virtual object can be seen as special kind of imaginary object where its root object plays the role of a core attribute. In that sense, it is easy to generalize the implementation strategy of virtual classes to take into account a tuple of attributes instead of a single object as the key to access (and construct) an imaginary object in the view.

5 Conclusion

In this paper, we have described our experience in designing and implementing O_2Views, an object-oriented view mechanism on top of the O_2 system. Despite its limitations, the prototype offers a great level of abstraction by allowing the rapid redefinition of both the structure and the behavior of objects stored in a database through a view definition facility which is heavily based on a declarative query language. The prototype is being used by members of the R&D Esprit Project Goodstep[7] in the construction of software engineering environments.

Some other related issues are under investigation, such as the ability to query and update data from a relational base through an object-oriented view, the integration of schema update facilities with the view mechanism and the integration of active rules for update propagation.

Acknowledgments: I would like to thank Serge Abiteboul, Claude Delobel, Joelle Madec and Emmanuel Waller for their valuable comments and fruitful discussions on the material of this paper.

References

1. S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 238–247, 1991.
2. Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1993.
3. F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.
4. T. Barsalou and N. Siambela. Updating Relational Databases through Object-Based Views. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1991.
5. Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. of Intl. Conf. on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.
6. Alexander Borgida. Modeling Class Hierarchies with Contradictions. Technical report, Rutgers University, New Brunswick (NJ US), 1988.
7. GoodStep Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proc. of the Asia-Pacific Software Engineering Conference*, Tokyo, Japan, 1994.
8. Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. ACM Conf. OOPSLA*, Oregon, 1986.
9. E. Rundensteiner. A Class Integration Algorithm and Its Application for Supporting Consistent Object Views. Technical Report 92-50, Department of Information and Computer Science, University of Califoria, Irvine, May 1992.
10. E. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. of VLDB*, 1992.
11. C. Santos, C. Delobel, and S. Abiteboul. Virtual Schemas and Bases. In *International Conference on Extending Data Base Technology*, Cambridge, March 1994.
12. C. Santos and E. Waller. *The O_2 Views User's Manual*, January 1995.
13. M. H. Scholl, Christian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *DOOD*, 1991.
14. K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema Virtualization in Object-Oriented Databases. In *Proc. IEEE Data Engineering Conf.*, Los Angeles, 1988.