Design and Implementation of an Object-Oriented View Mechanism*

Cássio Souza dos Santos†
(Cassio.Souza@inria.fr)

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 first given and then 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 a view such as view materialization and consistency maintenance.

1 Introduction

Views are an important tool for extending the modeling capabilities of a database system as they provide more flexibility for defining virtual structures adapted to the needs of a particular application. In addition, in the object-oriented paradigm, views can be used not only for adapting the structure of data to different kinds of applications, but also for redefining the behavior of objects in different contexts. Despite the number of recent proposals of view mechanisms for object-oriented databases [AB91, Ber91, HZ90, RB92, SLT91, SS89, TYI88, SDA94], there is very few account on actual implementations [SLT91, KR93].

The implementation of views in relational databases has raised a number of interesting issues concerning querying, data integrity, updates and performance [Ker86, Tem86]. With object-oriented views, old problems, such as view materialization and view updates, must be reconsidered, given the fundamental differences between the relational and the object-oriented data models. In addition, some new problems arise, such as virtual class (the analogue of a virtual relation in the relational model) integration to the class hierarchy and method resolution for objects in such classes.

In this paper, we discuss the design and implementation of an object-oriented view mechanism in the light of an experience with the implementation of O_2Views [SW93], an advanced prototype of a view mechanism built on top of the object-oriented database system O_2 [BDK92]. The prototype is first briefly presented (a more complete presentation is given in [SW93], which is part of the publicly available O_2Views package) and several issues related to its implementation are discussed. Emphasis is given on the requirements for the implementation of the view mechanism, ranging from the conception of the view definition language to optimization strategies for querying and

*Partially supported by Esprit Project GoodStep.

†On leave from Departamento de Informática, Universidade Federal de Pernambuco, Brazil. Partially supported by CNPq grant number 200.803-92.1.

updating through a view such as view materialization and consistency maintenance. A more complete description of the mechanism can be found in [SDA94]. Here we focus on the implementation aspects.

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 gives an overview of O_2Views. Section 3 presents the O_2 data model extended with views in a formal framework. Section 4 presents the view definition language implemented by the O_2Views prototype, which is an extension of the O_2C data definition language and partially implements the model presented in Section 3. Section 5 describes the implementation of O_2Views and Section 6 discusses some implementation issues and how the implementation can be enhanced so as to overcome its main limitations. Section 7 finally concludes and gives some directions for future work.

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 object-oriented database system [BDK92]. Once defined, a view can be used for querying and updating the database. 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. We assume the reader is familiar with O_2 related basic concepts (e.g. schema, class, method, inheritance). Nevertheless, let us recall some basic concepts of O_2 before proceeding. The first part of section 3 summarizes the formal presentation of the O_2 data model given in [BDK92].

An O_2 schema is composed of classes, types, functions and names.

Classes are related through inheritance and class `Object` is the top most class in the class inheritance graph (multiple inheritance can also be defined). A class has a type which defines the structure of its instances and a set of associated methods defining its instances behavior. Methods and types are inherited from superclass(es) as usual.

A type can be a class, an atomic type (integer, real, character, string, boolean), and any type obtained by recursively applying the list, set and tuple type constructors. Instances of a type other than a class type are values and are distinguished from objects (instances of a class) in that they do not have an object identifier (they cannot be shared) and do not respond to methods. Every object has an associated value of the type of its class and this value can be encapsulated.

Names define data entry points (persistence roots) and can be of any type (an object or a value). Named sets are usually defined to store instances of a given class and implement class extensions. Persistence is defined by reachability from a named entry point.

Functions are usually used to implement auxilary routines on values and are, as names, global to the schema. Class and name definitions can be shared among schemas through import/export facilities. Named objects and values can also be shared among bases in the same way.

2.1 Virtual Schemas and Bases

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.

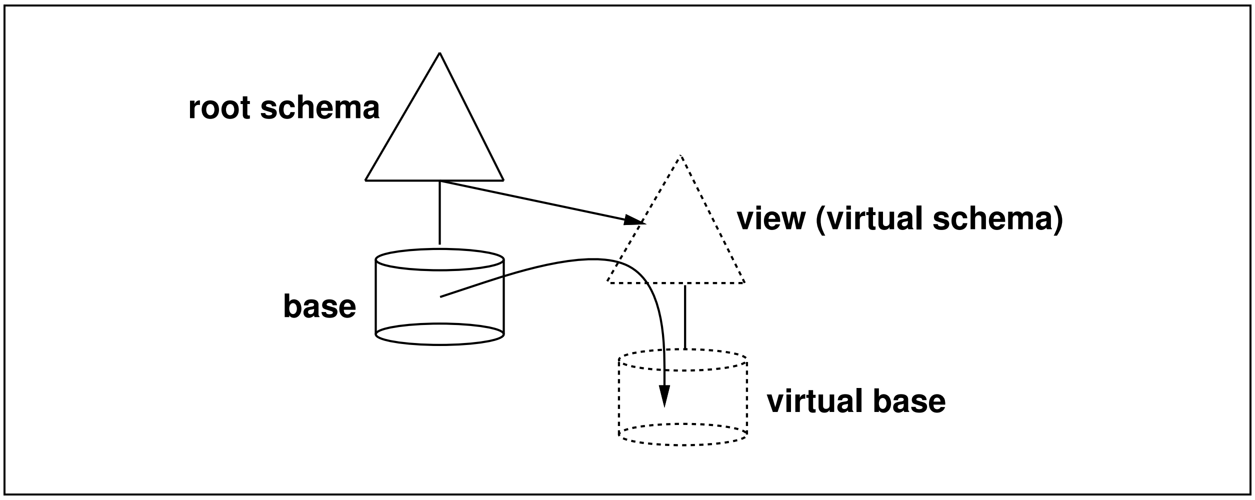

Figure 1: Relationship among schemas and bases

A view can be seen as a filtering schema through which the data of a root database schema are seen. The point is to get part of the structure and the data of the root database, possibly restructured. Roughly speaking, the primitives and their role are the following. *Virtual* and *imaginary classes* design which are the objects in the view, selected from the root database or created in the view. *Virtual attributes* augment the interface of objects, and *hidden attributes* restrict it. The user can access data through the extensions of classes.

We recall some differences between classes and named sets relevant in the present context. Designing a class corresponds to the following. (1) designing the structure of objects in that class, (2) providing them with object identifiers (oids), (3) attaching methods specific to this class, and (4) inheriting type and methods from the hierarchy. A named set (1) does not create a new type or class, (2) contains values or objects, (3) has no methods attached to it, and (4) is not connected to the class hierarchy. In addition, (5) its population is arbitrarily chosen by the programmer, for instance through a query, but it does not necessarily contain all the objects of a given class.

The main primitives of the language are:

Virtual Classes A virtual class (1) selects through a query objects existing in the root database, (2) is connected to the root hierarchy, and (3) provides a named set containing these objects: the extension of the virtual class. The aim is to provide the features of both usual classes and named sets described above. Moreover, since it is a class, (4) its interface can be modified by the view primitives for hiding an attribute or adding a virtual one.

Imaginary Classes An imaginary class (1) selects and restructures through a query data from the root database or the view, (2) turns them into objects, (3) is not connected to the root hierarchy, and (4) provides an extension. As for virtual classes, the aim is to provide the features of both classes and named sets, as well as being combined with other view primitives. The oid of an imaginary object is defined by its *core attributes*.

Virtual Attributes 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.

Attribute hiding It restricts the original interface of root objects or data. It hides the core attributes of an imaginary object not to be visible for an end-user.

Other features of the view mechanism such as generic views and generic virtual classes, which are discussed in [SDA94], are not considered in this paper.

Virtual classes allow the redefinition of both the structure and the behavior of existing objects whereas imaginary classes allow the definition of new objects which exist only in the view and whose values are built from existing data.

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 irrelevant for the understanding of the example and is therefore ommited here.

```
virtual schema example from person_schema;

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

imaginary class Family(husband:Person,wife:Person)
      extension Families
      virtual attribute 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)
```

Figure 2: Example of a virtual and an imaginary class

Virtual class `Adult` selects instances of class `Person` stored in the named set `People` whose age is above 20. Virtual attribute `age` is defined by using the method `age` "inherited" from the root class `Person` and attribute `date_of_birth` (also "inherited" from `Person`) is hidden. Inheritance of properties between a virtual class and its root class will be explained in Sections 3 and 5.

Imaginary class `Family` is built from two instances of class `Person`, which are joined according to a particular criterion (being married). In additon, virtual attribute `children` is derived by aggregation of nested attribute `children` of core attributes `husband` and `wife`.

An instances 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.

In the current prototype, a virtual schema is defined using an extension of the O_2C language, the O_2Views language, or through a user friendly graphic interface. The view definition language provides a set of primitives for defining virtual schemas and is presented in Section 4. A snapshot of the O_2Views graphic interface is illustrated in Figure 3, where we have four views stored in the view repository (upper right window), the view editor open on view `special_promotion` (lower left window) and the virtual class editor open on virtual class `Single` (lower right window).

Once defined using the O_2Views language or the graphic interface, a virtual schema definition is compiled and stored into a view repository. O_2Views can thus generate the implementation of such view, which is a standard O_2 schema with some system supplied classes, methods and functions implementing the view definition. This schema is called the *view implementation schema*. It can

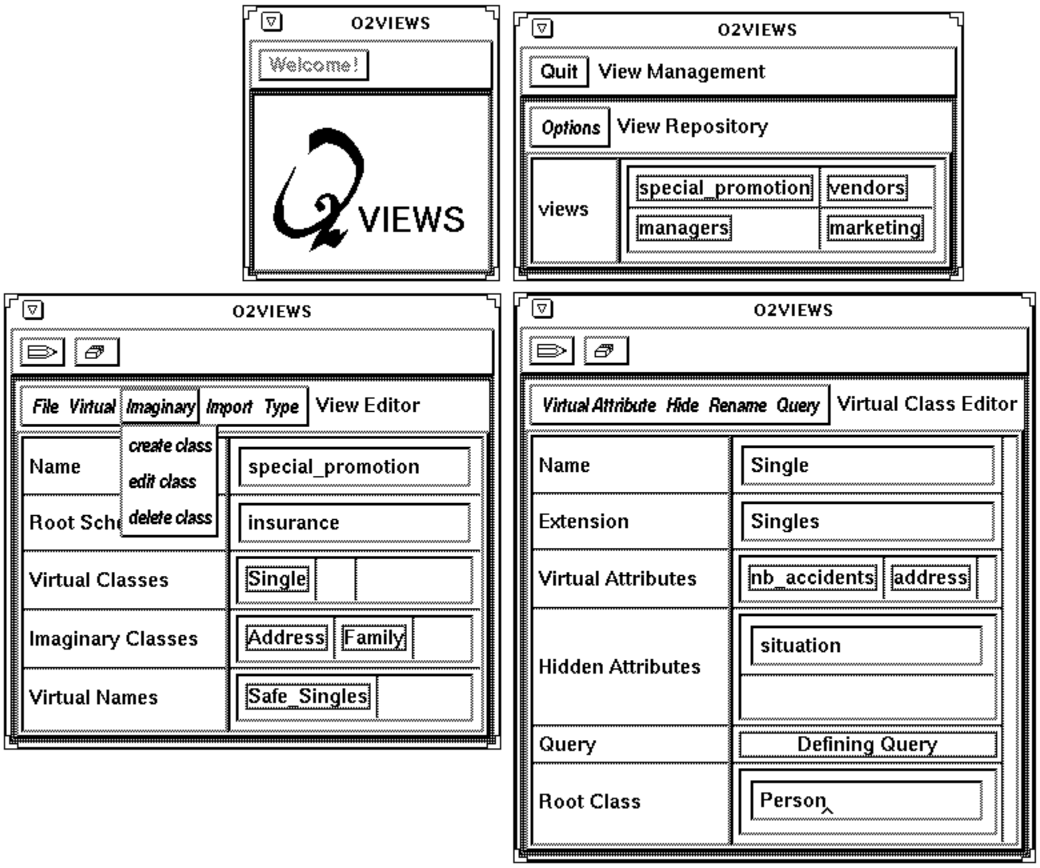

Figure 3: The graphic interface of O_2Views

be used as any ordinary O_2 schema and different applications can be defined on top of it. This is illustrated in Figure 4, where gray circles represent modules implemented by O_2Views.

A base instantiated from a view implementation schema corresponds to a virtual base in that it does not contain data of its own but is an image of other (virtual or real) base(s) through the view. Different bases instantiated from the same virtual schema are isomorphic up to the oids of virtual and imaginary objects, as it will become clear in the sequel.

3 Specification of the O_2 Data Model Extended with Views

In this section, we extend the formal presentation of the O_2 data model given in Chapter 3 of [BDK92] with the main concepts introduced in the previous section. We first present the core model without views, which is a condensed version of the original presentation given in [BDK92] and then introduce virtual schemas and virtual and imaginary classes. Some aspects such as attribute hiding and virtual attributes are omitted here for simplicity. Modularization aspects such as importing, exporting and visibility of definitions among schemas are also ignored in this presentation.

The prototype presented in the following sections implements the data model presented in this section and the functionalities of the implementation will be contrasted to the data model definitions whenever possible.

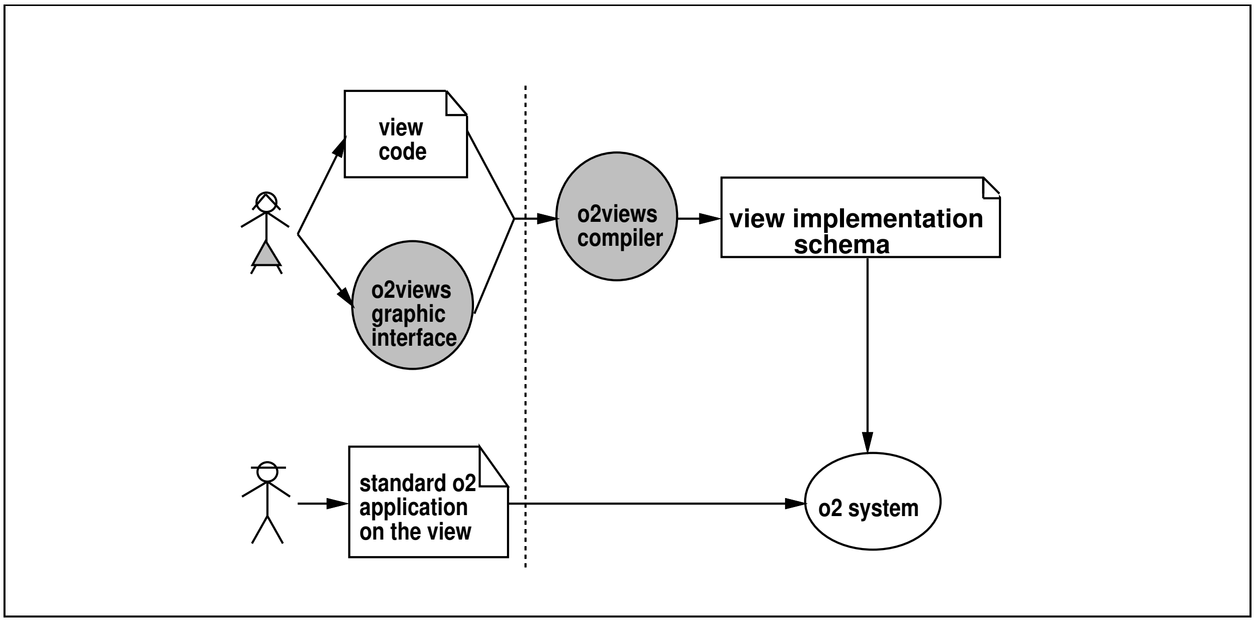

Figure 4: Using O_2Views

Core Data Model

Basics We have the set **D** of atomic values (i.e. the union of the atomic type domains integer, real, boolean, string and character and the special value **nil** which stands for the undefined value), the set **A** of attribute names, the set **I** of object identifiers and the set **C** of class names.

The definition of value remains unchanged with respect to that given in [BDK92], where tuple, list and set constructors are defined, as well as that of values over I, a subset of **I**, which is denoted by $\mathbf{V}(I)$. In particular, $\mathbf{V} = \mathbf{V}(\mathbf{I})$.

The definition of an object also remains unchanged, i.e. an object is a pair (i,v), where i is an identifier of **I** and v is a value of **V**.

$\mathbf{T}(C)$, where C is a subset of **C**, denotes the set of all types over C together with the special name **any** (a default type, the type of class `Object`). **T** denotes $\mathbf{T}(\mathbf{C})$. The definition of the subtyping relationship $\leq$ on $\mathbf{T}(C)$ remains unchanged (see [BDK92]), as well as that of types over C.

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 defined as follows:

$$\pi(c) = \pi_{\mathrm{d}}(c) \cup \{\pi(c') | c' \in C, c' \prec c\}.$$

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 [BDK92]), for all types t and t' in $\mathbf{T}(C)$ such that $t \leq t'$, we have $dom(t) \subseteq dom(t')$.

Schema A database schema S is a 6-tuple

$$(C, \sigma, \prec, maybe, M, G),$$

such that (C,σ,$\prec$) is a well-formed class hierarchy, M is a well-formed set of method signatures, G is a set of global names[1] with a type associated to each name. The fourth component, namely *maybe*, relates a virtual class to its root class(es). It is empty in a real schema (since there are no virtual classes) and is defined for allowing a uniform definition of view composition.

Base An instance of a database schema S, or a base for short, is defined as a 4-tuple (π,ν,μ,γ), 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.
- γ is a function associating each name of G, of type t, to a value of $dom(t)$.

Resolution 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 [BDK92]. More precisely, the semantics of $m(o)$ in a class c is defined by:

- if M contains a signature α of the form[2] $m : c \times t_1 \times ... \times t_{\rm n} \rightarrow t$ 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 (function *maybe* below), the signatures of the new methods. The populations of the virtual classes are specified using characteristic functions ($\aleph_c$ below) and the values of the new objects by methods (functions $val_{c'}$).

Virtual Schema Given a schema $S = (C, \sigma, \prec, maybe, M, G)$, a view S' of S is 6-tuple

$$(C \cup VC \cup IC, \sigma \cup \sigma' \cup \sigma_{\rm core}, \prec, maybe \cup maybe', M \cup M', G)$$

where:

1. VC is a set of virtual class names;
2. IC is a set of imaginary class names;
3. σ' is a mapping from $VC \cup IC$ to $\mathbf{T}(C \cup VC \cup IC)$;

[1] Global names are data entry points (class extensions are considered special cases of global names) are used to define the persistence semantics, so that every object or value with a global name is persistent and every object or value that is part of a persistent object or value is also persistent.

[2] In this notation, m is the name of the method and c,t_1, ...,$t_{\rm n}$ are types over C. The first type c is the class to which the method is attached.

4. σ_{core} is a mapping such that for all c in IC: $\sigma_{\text{core}}(c) \leq \sigma'(c)$ (σ_{core} gives the core type of the imaginary class, which defines the part of its value that determines the identity of its instance);

5. $maybe'$ relates each virtual class in VC to some nonempty subset of $C \cup VC \cup IC$ (the root classes of a virtual class);

6. M' is a set of well formed signatures over $C \cup VC \cup IC$ containing:

   - for each c' in VC:
     - for each c in $maybe'(c')$, $\aleph_{c'} : c \to bool$;
     - $val_{c'} : c' \to \sigma'(c')$;
   - for each c' in IC:
     - $\aleph_{c'} : \sigma_{\text{core}}(c') \to bool$;
     - $encode_{c'} : \sigma_{\text{core}}(c') \to \mathbf{I}$;
     - $val_{c'} : \sigma_{\text{core}}(c') \to \sigma'(c')$;

   $\aleph_{c'}$ is the characteristic function associated to each virtual and imaginary class. For a virtual class, the associated characteristic function takes an object of the root class as parameter and answers whether this object belongs or not to it. For an imaginary class, this function takes the core value, i.e. the tuple composed of the core attributes of an imaginary object[3] which determine its identity.

   $val_{c'}$ is the value constructor function associated to each virtual and imaginary class. For a virtual class, it takes an object of the root class as parameter and gives the value of the associated virtual object, whose construction takes into account information concerning hiding of attributes and virtual attribute evaluation. For an imaginary class, this function takes the core value in order to construct the corresponding value of the imaginary object.

and such that:

$$(\star) \text{ for each } c' \text{ in } VC,\ \sigma'(c') \geq \bigwedge\{\sigma(c) \mid c \in maybe'(c')\}.$$

The constraint $(\star)$ relates to the inheritance of structure. An object in class c' is from some class in $maybe'(c')$. Thus, it has at least the structure common to all these classes. Observe that it may have a richer structure. Note also that its value is defined by a method.

Virtual Base 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 $B = (\pi, \nu, \mu, \gamma)$ be an instance of some schema S and S' a virtual schema (view) 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, we now reexamine the components of a standard instance in the context of the view. We obtain an instance $B' = (\pi', \nu', \mu', \gamma)$:

1. π':

   - for each c in C, $\pi'(c) = \pi(c)$;

[3] We assume for simplicity that imaginary objects are tuple valued, as in most practical cases.

- for each c' in VC, $\pi'_{\mathrm{d}}(c') = \{o \mid \exists c \in maybe(c') : (o \in \pi'(c) \wedge \aleph_{c'}(o))\}$;
- for each c' in IC, $\pi'_{\mathrm{d}}(c') = \{encode_{c'}(v) \mid \exists v \in dom(\sigma_{\mathrm{core}}(c')) : (\aleph_{c'}(v))\}$;
  In addition, for each c' in IC, $\forall c \in C : c \neq c' \Rightarrow \pi'_{\mathrm{d}}(c) \cap \pi'_{\mathrm{d}}(c') = \emptyset$[4];

2. ν':
   - for each c in C and o in c, $\nu'(o, c) = \nu(o, c)$;
   - for each c' in $VC \cup IC$ and o in c', $\nu'(o, c') = val_{c'}(o)$;

3. μ' is already defined over M', and it is extended to M by: for each signature α in M, $\mu'(\alpha)$ $= \mu(\alpha)$;

Resolution We revisit resolution in the context of virtual classes.

Resolution is defined using the partial order $\prec$ and the relation *maybe*. 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 $(\mu'(\alpha))(o)$ (for c in $C \cup VC \cup IC$);
- otherwise, if c is a subclass, we use the semantics of $m(o)$ in some superclass of c. If it is undefined in all superclass, m is undefined in c, and if it is multiply defined, the schema is inconsistent;
- otherwise, if c is a virtual class, we use the semantics of $m(o)$ in some class in $maybe(c)$ where it belongs. If it is undefined in all classes in $maybe(c)$, m is undefined in c, and if it is multiply defined, the schema is inconsistent.

Remark: In this presentation, we separated inheritance with subtyping (a partial order) from inheritance from virtuality (a relation). It is possible to achieve a more uniform approach but we will not pursue this issue here.

4 The O_2Views Language

The O_2Views language is a conservative extension of O_2C, the native language of O_2. In what follows, we present the syntax of the main commands of the language and informally describe their semantics. A complete presentation of the language is given in [SW93].

O_2Views commands extend the set of commands provided by the O_2C language for schema development, but they can be used only in the definition of virtual schemas. This is in accordance with the philosophy of our mechanism, whereby there is a clean separation between real and virtual schemas. Real schemas describe the structure and behavior of persistent data only and therefore are not allowed to see (import) definitions from virtual schemas, whereas virtual schemas define alternative structures and behaviors for stored data and are thus allowed to see (import) definitions from other (real or virtual) schemas and data from other (real or virtual) bases.

Virtual Schemas

- "`virtual schema`" *schema_name* "`from`" *schema_name* "`;`"

This command allows the creation of a virtual schema and must be the first command in a view definition. The schema name declared after the `from` clause is the name of the *root schema* of the virtual schema being defined and must be an existing real or virtual schema.

[4] This models the fact that an imaginary object exists only in the virtual base and that a new object identifier is associated to it, i.e. it does not share its identifier with any existing object in the original base. Note, however, that it may share its identifier with a virtual object derived from it.

Virtual Classes

- "`virtual class`" *class_name* "`from`" *class_name*
  ["`extension`" *extension_name*]
  ["`virtual attribute`" *virtual_attribute_list*]
  ["`hide attribute`" *attribute_name_list*]
  "`includes`" *query_expression*

Creates a virtual class definition under the virtual schema. The `from` clause gives the *root class* of the virtual class[5]. 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).

Note that the syntax is not minimalist, in that some redundant information is sometimes required. For instance, the root class of a virtual class can be inferred from the defining query but we preferred to have it explicitly declared by the user. On one hand, we believe that forcing the user to declare the root class is a good programming discipline. In addition, this has considerably eased the effort spent in the first implementation of the prototype.

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 ("`The_`*class_name*`s`"). This name implements the π' function of the data model.

Methods (behavior) and attributes (structure) are "inherited" through delegation from the root class. In other words, in the current implementation, each instance of a virtual class holds a pointer to an instance of its root class (which can be a real or a virtual object). Methods and attributes which are not defined or redefined in the virtual class are thus automatically delegated to the root object. This can be viewed as a special case of instance inheritance (delegation) which is provided automatically by the compiler. The implementation of this mechanism is described in Section 5.2.

In order to loosen the constraints imposed by the commonly adopted "*first define then use*" rule, the definition of a virtual class can be given incrementally, through a separate command for adding a virtual attribute to a class, which is shown below, so that classes can be mutually dependent (e.g. two virtual classes can depend one on the other through the type declaration of their virtual attributes). The only mandatory clauses are the root class and defining query ones.

Imaginary Classes

- "`imaginary class`" *class_name* "`:`" "`(`" *core_attributes* "`)`"
  ["`extension`" *extension_name*]
  ["`virtual attribute`" *virtual_attribute_list*]
  ["`hide attribute`" *attribute_name_list*]
  "`includes`" *query_expression*

Creates an imaginary class definition under the virtual schema. The query expression must return a unique set of tuple values whose type is that defined by the core attributes. The `extension` clause works as for virtual classes, that including the rule for deriving default extension names.

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. This is already guaranteed by the fact that the query must return a unique set of tuples, but further attempts to extend the extension of an imaginary class will not create redundant objects, i.e. if an imaginary object already exists for a given tuple, it will be returned, otherwise a new object is created for that tuple.

The definition of an imaginary class can also be given incrementally, for the same reason as for virtual class definitions. Virtual and imaginary classes can be mutually dependent (through the

[5] Class generalization is not yet implemented, therefore a single root class must be given.

root class clause of the virtual class or through type declarations of their respective virtual and core attributes). View definitions can thus be composed to any extent (intra-view and inter-views), provided the "*first define then use*" rule is observed.

Virtual Attributes

- "`virtual attribute`" *attribute_name* "`:`" *type_expression*
  "`in class`" *class_name* "`has value`" *query_expression*

Adds a virtual attribute to the virtual or imaginary class given in the clause `in class`. The type of the virtual attribute must be given and the query expression giving the value of the virtual attribute (its defining query), must be of the declared type. Note that the type of virtual attributes could be inferred (as the root class of a virtual class) from the defining query.

5 The Implementation of O_2Views

O_2Views has been implemented on top of O_2. In our presentation, we highlight the features that an OODBMS must support for implementing a view mechanism. In that sense, we stress the boundaries between the current O_2Views system and a more integrated and advanced implementation, showing which extensions to the O_2 engine would be necessary and important for the implementation of a more efficient view mechanism, as well as at which level with respect to the current O_2 system stratified implementation these should be implemented.

Our implementation allows the definition of most of the (meta) data structures which are likely to be reused by a schema manager in a more advanced implementation of the mechanism, as well as the experimentation with the view definition language in real applications so as to assess the modeling capabilities of the language and to tune it up. We will not be able to describe the whole implementation due to space limitations. Instead, we will focus on the implementation of virtual and imaginary classes, the main primitives of the language. Before, we comment on the required features of an OODBMS for the implementation of the view mechanism and contrast them to those supported by O_2.

5.1 OODBMS Required Features

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[6]
- 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

[6] The algebra allows the retrieval of existing objects but not the construction of new objects.

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 O_2SQL, 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 O_2C functions and called from an O_2SQL expression. The characteristic functions ($\aleph_c$ for each virtual class c) can be easily derived from the defining queries, as described in Section 5.4.

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.

The modularization mechanism of O_2 is extended in our model to cope with virtual definitions. A view is a module where inter-related definitions are collected to define a virtual base. Nevertheless, visibility between a virtual and its corresponding root schema should be defined differently from the way it is defined between two real schemas in O_2, so that visibility privileges could be granted to a virtual class w.r.t. its root schema.

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

5.2 Implementing Virtual Objects as Surrogates

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. This allows a disjoint property assignment. 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 and the system keeps track of the relationship between such an object and its corresponding root object (the corresponding instance of the root class of whom it is an image through the view).

Virtual classes are immediate subclasses of a special system supplied class called `Virtual_Object`. This special class implements the functionalities common to all virtual objects, which are explained below.

Inheritance is a means of sharing structure and behavior among classes. Delegation [Lie86] can also be used to achieve such a sharing and that is the way a root class shares its structure and behavior with its virtual class(es). Delegation is also called instance inheritance, to contrast to the more conventional class inheritance concept.

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 a* instance of its superclass(es), and the relationship between a virtual class and its corresponding root class is called 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 is related to the 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` as the root class. If an object *o* in a virtual class is a surrogate for some object *o'*, the `root_object` attribute of *o'* will contain *o*.

Let us consider the virtual class `Adult` whose definition is given in Figure 2. Virtual attribute `age` is added to the tuple type of the class whereas (`date_of_birth`) is removed. Note that the type of the virtual class is neither a subtype nor a supertype of the type of the root class. Also recall that new methods can be defined in the virtual class and some methods "inherited" from the root class can be redefined (they may have the same name but not necessarily the same signature).

Class `Adult` is implemented as a subclass of `Virtual_Object` in the view implementation schema generated by the compiler and it inherits the attribute `root_object` from its superclass and redefines its type. This is shown in Figure 5. Auxiliary methods put in the public interface of this class are explained in the sequel.

```
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
```

Figure 5: Implementation of Virtual Class `Adult`

By keeping virtual classes in a different hierarchy, we solve 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). There is a trade-off in that substitutability is not always preserved, i.e. an instance of a virtual class cannot always occur where an instance of its root class is expected (this is only possible in case of a type conservative extension, i.e. subtyping). Although this seems to be a serious limitation since it limits polymorphism, in many cases, which can be detected at compile-time, virtual class types are compatible with respect to their root class type, so that substitutability is preserved (but this is no longer a default case). As pointed in [WZ89], the requirement of substitutability and the associated notion of subtype and behavioral compatibility is too strong in many practical situations. In such cases, it is the subset relation involving class extensions that must determine the *is-a* relationship, rather than the subtype relationship. Of course, this can prevent some useful compile-time optimizations to be performed and compromise reusability. In addition, existing programs may be invalidated.

Program maintenance can be seriously affected if we allow a type-compatible virtual class definition to become type-incompatible due to a schema update, so we can force type-compatibility to be a monotonic property of a virtual class definition. This leads us to distinguish between two types of *may-be* relationship: the type-compatible one (*may-be*) and the type incompatible one (*may-be-like*). It is important to observe that type compatibility is related not only to attribute types but also to method signatures.

Another incompatibility which may emerge if we try to relate virtual and root classes through inheritance is related to 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).

The *may-be* relationship defines new class hierarchies (one for each real root class in the virtual schema), which will be called virtual class hierarchies. The root of such an hierarchy is always a real class and all other classes in the hierarchy are virtual and represent different points of view of the topmost real root class[7]. 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()`[8], 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 is summarized below:

- Convertion function `As_`*Virtual_Class*`(obj:`*Root_Class*`)`:*Virtual_Class*: allows the instantiation of the virtual class from an instance of its root class. This is a real function, in that for a given instance of the root class, the same corresponding virtual object will be returned by every call to this function (functional transparency). In a sense, the convertion function implements the method $val_{Virtual\_Class}$ of the data model, but the value of the virtual object is encapsulated into a new object (equivalence between this new object and the original object can be tested using the equivalence test method given below).

- Characteristic function `Is_`*Virtual_Class*`(obj:`*Root_Class*`):boolean`: allows to test 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, or in other words, if the root object is included in the set returned by the evaluation of the defining query[9]. It implements method $\aleph_{Virtual\_Class}$ as it is defined in the data model.

- Consistency check method `is_consistent:boolean`: tests whether a virtual object has become inconsistent due to an update or a side-effect.

- Virtual structure update method `update_attributes`: updates the values of virtual attributes. These can become obsolete due to side effects.

- Virtual object equivalence test method `is_same_as(obj:Object):boolean`: allows to test whether two given objects are different points of view of the same real object. The object passed as parameter can be itself a real object. This method makes it up for the fact that (in the current implementation) the identifier of a virtual object is not the same as that of its corresponding root object, contrary to what is specified in the data model. In that sense, it simulates the equality operator, i.e. it is an extension of such operator to test equality between objects in the view.

- Attribute update methods. 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. Public and read attributes are "inherited" from the root class as read attributes.

[7]An analogy with *role* objects [RS91, ABGO93] can be considered, whereby the *may-be* relationship defines a role relationship rather than just a different view of an object. A role of an object can not only restructure an object but also extend it with additional information, i.e. new stored attributes

[8]System supplied functions are generated by the O_2Views compiler.

[9]Testing the membership of a given object in the set returned by the evaluation of the defining query of a virtual class is a rather naive and expensive way of implementing the characteristic function. As it will be seen later, more efficient test functions can be derived from the defining query by avoiding query evaluation to a large extent.

Figure 6 illustrates the relationship between a view definition and its corresponding view implementation schema.

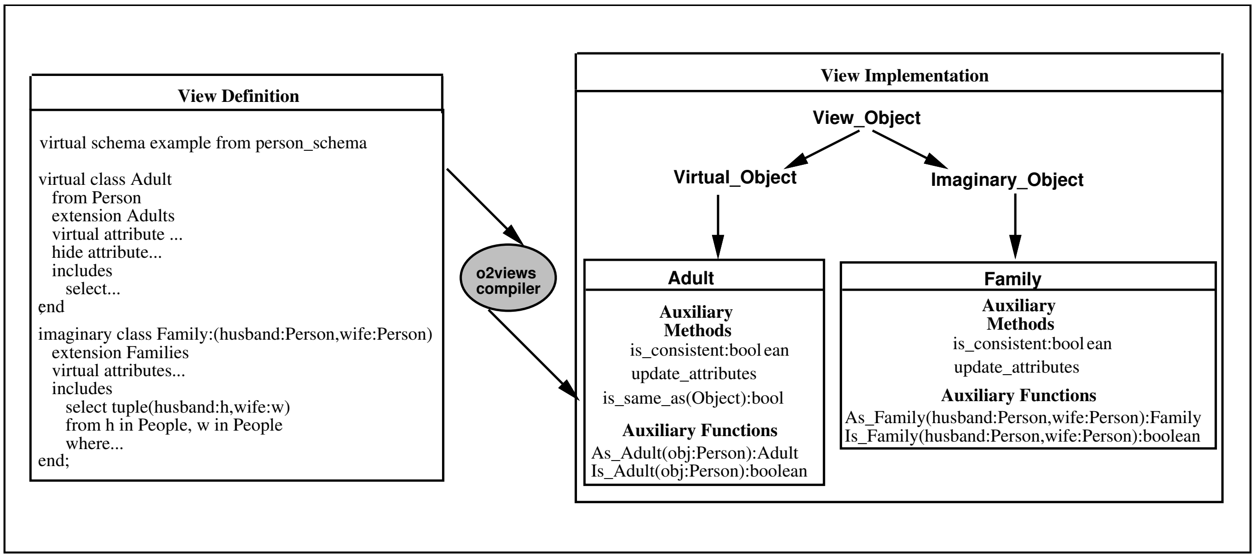

Figure 6: The view implementation schema generated by the O_2Views compiler

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 [Run92]. 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 [Bor88]) of existing concepts instead. In this prototype, we adopted the second alternative since it seems more intuitive and also gives more flexibility to define virtual structures and behaviors, not to mention the simplicity of its implementation compared to that of a complete (and rather complex) classification algorithm.

5.3 Implementing Imaginary Objects

In a sense, the implementation of imaginary classes is simpler than that of virtual classes, since there is no inheritance relationship between imaginary classes and existing classes in the hierarchy. On the other hand, the identity of imaginary objects poses another problem, that of guaranteeing the uniqueness of imaginary objects with respect to their core attributes. 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.

Imaginary classes are immediate subclasses of a special system supplied class called `Imaginary_Object`. This special class implements the functionality common to all imaginary objects. Common structure and behavior of virtual and imaginary objects were factored out into a common superclass of `Virtual_Object` and `Imaginary_Object` called `View_Object`, as illustrated in Figure 6.

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

- Convertion function `As_`*Imaginary_Class*(*core_attributes_signature*) : *Imaginary_Class*
- Characteristic function `Is_`*Imaginary_Class*(*core_attributes_signature*) `:boolean`
- Consistency check method `is_consistent:boolean`
- Virtual structure update method `update_attributes`

Their semantics is similar to that provided for virtual objects.

5.4 Characteristic Function Generation

Deriving optimal characteristic functions is crucial since such functions are intensively used by the resolution as well as the view object construction mechanism. For each virtual class, the compiler generates a characteristic function from the defining query. The type of the value returned by a defining query must be checked and only queries returning a set of objects can be used in the definition of a virtual class. Given a virtual class *VC*, the O_2SQL query predicate `x in VC` is translated as $\aleph_{\mathrm{VC}}(x)$. Virtual class extensions are thus not necessarily evaluated (unless explicitly stated) but may be referred to in query expressions as if they were.

A query is an expression of the form $f(x_1, x_2, ..., x_n)$, where the x_i's are free variables. They are assigned parameters (objects or values) when they are evaluated. The result of a query may be an object or a value, but as mentioned above, only queries which return a set of objects are accepted for virtual class definitions.

Given a virtual class *VC* with defining query q, the extension of VC is the set returned by the evaluation of q. The specification of the characteristic function can be given by

$$\aleph_{\mathrm{VC}}(x) = x \texttt{ in } q$$

where q denotes the set returned as a result of the query evaluation. We want to derive a characteristic function from the query expression so that the virtual class extension does not have to be explicitly constructed. Let us consider some examples.

Consider the virtual class `Adult` defined in Figure 2. The compiler can infer that the type of the extension is `set(Person)`. The characteristic function of class `Adult` is given in Figure 7.

```
function body Is_Adult(obj:Person):boolean {
   return(obj->age ≥ 21);}
```

Figure 7: Characteristic function of virtual class `Adult`

Some private attributes and methods may be used in the query expression and should be accessible to the characteristic function in spite of visibility constraints. In addition, only instances of the root class and its subclasses should be tested to be instances of the virtual class(es) defined from it. These considerations seem to suggest that the characteristic function should be implemented as a method of the root class. Currently (and this is a minor limitation of the prototype) this is not the case because we are not able to update the definition (in this case, add a new method) of a class imported to the view implementation schema.

In the definition above, not only instances of class `Person` can be tested to be instances of `Adult` but also instances of `Person` subclasses. From the defining query, however, this is not clear, since the named set `People` may contain or not instances of subclasses of class `Person`. In a model with extensions, however, the semantics of resolution based on the defining query and on the characteristic function (as a method of the root class) would be equivalent.

The definition of class `Boss` below along with the associated characteristic function are taken from an example given in [SDA94].

```
class Boss includes
      (select d.boss from d in The_Departments)

function body Is_Boss(obj:Employee):boolean }
   {return(exists d in The_Departments:(d.boss = obj));}
```

Figure 8: Class `Boss` and associated characteristic function.

Let us proceed with a case analysis on the syntactic structure of the defining query q of a given virtual class `VC` and see how the characteristic function is derived in each case.

- $q =$ `n`, where `n` is a named set.

  The type of `n` must be `set(C)`, where `C` is a class. The body of function `Is_VC(obj:C):boolean` is { `return(obj in n);`}.

- $q =$ `n.a`, where `n` is a name of an object and `a` is one of its attributes.

  The type of attribute `a` must be `set(C)`, where `C` is a class. The body of `Is_VC(obj:C):boolean` is { `return(obj in n.a);`}.

The derivation of the characteristic function is analogous for other basic queries returning a set of objects, such as `first(x)`, `last(x)`, `flatten(x)`, `x`$\leftarrow$`m(`x_1`, `x_2`, ...,`x_n`)`, `x union y`, `x minus y`, `x inter y`. We consider the union case below and take the other definitions for granted.

- $q =$ `x union y`, where x and y are named set of objects or are returned by a function.

  There must be a class `C` which is the common least superclass for objects in `x` and `y`. This is the root class of the virtual class defined by the query. The body of `Is_VC(obj:C):boolean` is given by { `return(obj in x or obj in y);`}. Note that the union is not computed and that the operator `or` is not strict so that if `obj in x` evaluates to true then the second operand is not evaluated.

Now let us see the select-from-where case, which is the most useful in the definition of virtual classes.

- $q =$ `select` $r(x, \vec{X}_n)$
  `from` x `in` y
  `where` $p(x, \vec{X}_n)$,
  where $r(x, \vec{X}_n)$ and $p(x, \vec{X}_n)$ are query expressions and every x_i in $\vec{X}_n$ is present as a free variable.

  The root class of the virtual class defined by such query is the least common superclass of the elements of the set returned by the query. This set is the image by q of the subset of y satisfying p. Instead of a set, y may be a class name if this class is the same as the root class inferred from the query. In the current implementation, this allows the definition of virtual classes from classes for which named sets are not explicitly maintained to simulate their extensions (a common programming practice in O_2). The `where` part of the query may be omitted, in which case it is equivalent to a query with a `where true` clause (no filtering).

Given that the type of the value returned by the query is `set(C)`, we define the body of function `Is_VC(obj:C):boolean` as {`return(exists x in y:(r(x,`$\vec{X}_{\mathrm{n}}$`)=obj and p(x,`$\vec{X}_{\mathrm{n}}$`)));`}.

The general format for $\aleph_{\mathrm{VC}}$ given above can be simplified if r is the identity function applied to x and y is the root class extension, a very common case (e.g. class `Adult` in Figure 2). In this case, the body of $\aleph_{\mathrm{VC}}$ can be given by { `return(p(obj,`$\vec{X}_{\mathrm{n}}$`));`}.

Assuming the general definition of $\aleph_{\mathrm{VC}}$, the body of `Is_Adult` defined in Figure 2 would be given by

```
{return(exists x in People:(x = obj and x.age ≥ 21));}
```

but if we take the simplified version we obtain the definition given in Figure 7, which is semantically equivalent but can be evaluated more efficiently.

The grouping operator is not used in the definition of virtual classes since it returns a set of tuples and not a set of objects. It can be however useful in the definition of imaginary classes. The sorting operator is also not necessary for the definition of virtual classes (it can be nevertheless necessary in the elaboration of nested queries of the defining query, as for the grouping operator).

6 Implementation Issues

In this section we discuss some implementation issues in the light of the limitations of the current prototype. Then we overview a temptative preliminary architecture which integrates the view mechanism at a lower level of the O_2 system.

6.1 Enhancing the Current Prototype

The current prototype is a preliminary work and does not completely implements the semantics presented in Section 3 nor that discussed in [SDA94]. Some of these limitations have to do with the fact that the prototype has been entirely implemented on top of the O_2 system. In that sense, some features of the view mechanism could not be implemented since they require modifications at lower levels of the O_2 system. Other limitations result from the fact that we did not intend to implement the whole functionality of the view mechanism in the first prototype, in order to simplify the task and be able to experiment rapidly with its main features. Extensions to the prototype are already under development and must become available soon.

The main limitations and some techniques to overcome them are discussed below.

A great level of control is left to the programmer. The current prototype encourages a defensive programming style, whereby the programmer is sometimes required to perform tests (through the characteristic functions) before performing conversions (through the conversion functions) in order to guarantee the soundness of his or her transactions. Also, virtual data consistency must be explicitly checked by the programmer. This can be improved if we rely on active rules that trigger appropriate actions whenever some conditions hold. Triggers can be automatically installed (based on the data dependency graph inferred from the defining queries in a virtual schema) on update actions for guaranteeing consistency. We are studying the use of a triggering mechanism [CHCA93] to that end. For instance, invalid updates (updates making virtual objects inconsistent w.r.t. their definition) can be blocked or the virtual object can become obsolete.

The view implementation schema is a standard O_2 schema. In particular, we can not prevent the user from defining persistent data on it. On one hand, the user should use the implementation schema as any other schema for defining applications on the view in a transparent manner. On the other hand, some restrictions should be applied on the use of such schema so as to preserve the view semantics, but these are currently not imposed by the system and it is up to the user to make good use of the view schema. In an ideal situation, a virtual schema should be managed by the schema manager at a lower level of the stratified O_2 system implementation so that restrictions on its use could be enforced by the system itself. The main restrictions are:

- Data defined in a virtual schema cannot become persistent. It is important to distinguish between persistent and materialized data. The first are data stored on a base and which persist between database sessions and can be referred by other persistent data, whereas the second are derived data which may be temporarily and transparently stored (materialized) for improving performance (by avoiding unnecessary recomputations) but cannot be accessed from any root of persistence.

- Definitions in a virtual schema cannot be imported from real schemas. This has to do with the independence of real schemas and bases with respect to virtual definitions. For those familiar with the deductive database terminology, virtual schema definitions are analogue to intensional predicates defining an intensional database (the virtual base) whereas real schemas and bases corresponds to extensional predicates and bases respectively.

The type closure of a view is not inferred and must be provided by the user. This implies a number of necessary import of definitions. The view designer should be relieved from such mundane tasks. The type closure of a view is the set of types necessary for the compilation of a view implementation schema. In particular, all class types involved in the defining query, virtual attributes and view methods must be imported to the view. Also, external named types must be redefined in the view (recall that such types cannot be imported in the O_2 system).

A virtual schema should be able to access all definitions from its root schema. In that sense, we can think of a schema as a class and its definitions (classes, types, functions, programs and names) as class properties. Pursuing the analogy, a base can be thought of as an instance of such a class and views as methods turning a base into its virtual image (the virtual base). As a method, it can see every property of its class (the root schema), even private, i.e. not exported ones. Clients of such a method (applications written on top of the view) cannot however see private properties of the class (the root schema) but only access them indirectly through its methods (the view definitions). This analogy suggests a visibility hierarchy for views similar to that defined in the object-oriented model, with an inheritance relationship defined by the virtual-root schema relationship and a composition graph defined by entries in such schema.

Dependencies among view definitions are not checked by the system. Maintaining such dependencies is important not only for maintaining consistency of data in the view but also for deriving such data, since the construction of virtual data may depend on other virtual data. A stratification of the virtual object construction process based on the dependency graph derived from the virtual schema must guarantee its completion. Cyclic dependencies must be handled, whenever possible, by special mark-up algorithms to avoid looping.

Updates are performed on a limited basis. In addition to the view update problem, some restrictions to updates through the view in the current prototype have to do with the way objects and values imported from an external base can be updated through the importing base. Objects must be created in the context (base) where they are to be stored. This, as in distributed databases, should be made transparent to the end user as well as to the view designer.

Automatic interface resolution is not implemented. The implementation of the automatic interface resolution implies changes to the resolution mechanism of O_2 that could not be implemented on top of it. Automatic resolution is related to the ability that objects have to acquire virtual structure and behavior automatically (without explicit calls to convertion functions) as a result of a view activation. This amounts to changing the dereferencing operator (`->`) and performing dynamic reclassification of objects. Ambiguity due to multiple definitions of virtual classes from the same root class must be avoided and different alternative solutions to cope with this problem are considered in [SDA94].

To sum up, the view implementation schema simulates many of the tasks that must be performed by the schema manager on virtual schemas. On one hand, this simulation limits both the performance and the semantics of the language implementation. On the other hand, it makes the prototype a portable tool, in that it can be easily loaded in any running O_2 system.

6.2 Integration of O_2Views into O_2

In this section we sketch an architecture for the view mechanism and briefly consider the main issues behind its implementation.

View Activation At view activation, some machinery must be turned on so as to bring virtual and imaginary objects into play. Activating a view is equivalent to context switching. Once activated, only those objects (re)defined in the view can be accessed and they acquire the interface defined in the view (structure and behavior). A view acts thus as a framework under which data can be manipulated.

View Manager Architecture The view machinery, which we will be called the *view manager* from now on, is composed of the following main inter-communicating modules:

- view builder
- view materializer
- consistency checker
- update manager

The view manager is part of the O_2 schema manager. Its architecture is depicted in Figure 9.

View Materialization Materialization of virtual structures must be performed to avoid unnecessary recomputation. Once a virtual structure is constructed, it is kept by the system (as if it were a persistent object) as long as its consistency is not affected so that further references to it do not entail its recomputation. Materialization can be performed on a lazy basis, so that virtual structures are constructed and materialized on demand. This may imply partial construction of view objects (e.g. delayed evaluation of virtual attributes). In addition, the rematerialization overhead incurred by updates to a base can be reduced by applying techniques such as those described in [KKM91], where the characteristics of the object-oriented paradigm (encapsulation and object identity) are largely exploited.

Virtual Object Storage In the presence of materialized views, some storage strategies can be defined to reduce retrieval time for virtual structures. The system can, for instance, keep track of the relationship between a virtual structure and its root structure so as to avoid redundant storage, and the object manager can adapt its clustering strategies accordingly. Of course, it remains to be seen if such strategies really pay.

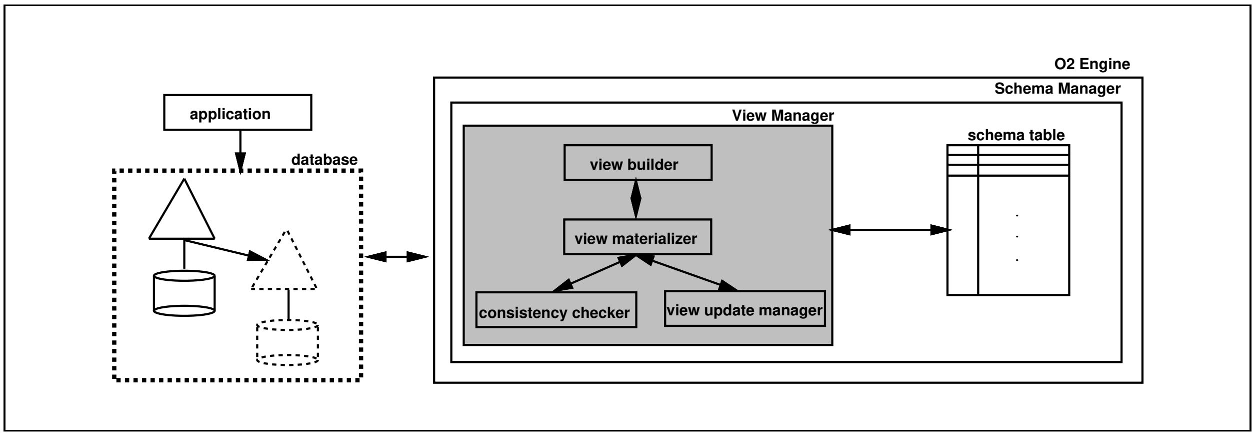

Figure 9: Architecture of the View Manager

Structural Cyclicity Since virtual structures are functionally specified through O_2SQL, mutual recursion (dependency) can arise and must be handled by the system, whenever possible. In general, it is not possible to detect which are the tractable cycles in the dependency graph[10] but a good level of flexibility is already provided if cycles at the class extension level are forbidden whereas cycles at virtual attribute definition level are allowed, in which case a many-phase object construction algorithm can be applied (one phase for each stratum derived from the dependency graph).

Updates The propagation of updates from a virtual to its corresponding real object is eased in the presence of oids. This is also true for object attributes of an imaginary object and also for value attributes whose origin can be inferred from the defining query (as in the relational context). Updates to a virtual attribute cannot always be propagated however, since it is not always possible to define the inverse of its defining function (query). Such inverse function must be provided by the user or inferred by the system, whenever and whichever possible.

View Consistency In order to keep data in a view consistent, if we want to see a view not only as a database snapshot but as a dynamic image of an existing database through transforming functions, we need to keep track of data interdependencies so as to be able to check consistency and propagate updates accordingly. Active capabilities must be provided to trigger appropriate actions in response to changes to the root base (performed through the view or resulting from a side effect). This requires sophisticated client-server communication facilities so that at any time a client of a view can have a consistent image of the underlying database.

Optimization Different optimization techniques can be applied to the view management system. Indexing can be automatically performed on root objects and core attributtes to speed up the evaluation of characteristic functions. Also, nested query optimization techniques for object bases [CM93] can be adapted to nested view definitions (which implicitly define nested queries) so as to obtain more efficient characteristic functions. Lazy materialization also plays an important role in making the view mechanism effective and should be largely exploited.

7 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. The data model supported is an extension of

[10] For instance, mutual dependencies may be based on the use of aggregation functions such as `average` or `count`.

the O_2 data model. We have considered the main issues concerning the implementation of the prototype and described how the mechanism was implemented to a satisfactory degree. Then we have discussed the limitations of the current prototype and how these can be overcome, as well as other optimizations which are likely to be adopted in a more advanced version.

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. All that is achieved without affecting existing applications, due to a clean separation between real and virtual contexts.

The prototype is currently publicly available and is being used by members of the R&D Esprit Project Goodstep in the construction of software engineering environments. A more advanced release is under implementation and will be incorporated to the final Goodstep platform.

A complete formal specification, taking into account the organizational aspects of the model as well as updatability through views, is being developed and is already guiding the ongoing implementation of the mechanism.

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 [BS91], the integration of schema update facilities [Zic91] with the view mechanism, the integration of active rules as they are being defined in the framework of project Goodstep [CHCA93] and the maintenance of files through an object-oriented view [ACM93].

Acknowledgments: I would like to thank Joelle Madec from O_2 Technology for her technical support and Serge Abiteboul, Claude Delobel and Emmanuel Waller for their valuable comments and fruitful discussions on the material of this paper. Emmanuel Waller is also thanked for the design of a first application in O_2Views, that was very helpful in debugging the prototype.

References

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 238–247, 1991.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An Introduction to the Database Programming Language Fibonnacci. Technical report, FIDE/93/64, University of Glasgow, 1993.

[ACM93] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and updating the file. In *VLDB Conference*, 1993.

[BDK92] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[Ber91] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *International Conference on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.

[Bor88] Alexander Borgida. Modeling Class Hierarchies with Contradictions. Technical report, Rutgers University, New Brunswick (NJ US), 1988.

[BS91] T. Barsalou and N. Siambela. Updating Relational Databases through Object-Based Views. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 248–257, 1991.

[CHCA93] C. Collet, P. Habraken, T. Coupaye, and M. Adiba. Active Rules within an Object-Oriented Database System. Technical report, LGI-IMAG, University of Grenoble, Grenoble - France, 1993.

[CM93] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. Technical report, $FIDE_2$ ESPRIT Project, 1993.

[HZ90] S. Heiler and S.B. Zdonik. Object Views: Extending the Vision. In *IEEE Data Engineering Conference*, pages 86–93, Los Angeles, 1990.

[Ker86] Brigitte Kerhervé. *Vues Relationnelles : Implantation dans les Systèmes de Gestion de Bases de Données Centralisés et Repartis.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 1986.

[KKM91] Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Function Materialization in Object Bases. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 258–267, 1991.

[KR93] Harumi A. Kuno and Elke Rundensteiner. Developing an Object-Oriented View Management System. University of Michigan, 1993.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. OOPSLA, Portland, OR*, 1986.

[RB92] Elke Rundensteiner and Lubomir Bic. Automatic View Schema Generation in Object-Oriented Databases. Technical Report 92-15, Department of Information and Computer Science, University of Califoria, Irvine, Jan. 1992.

[RS91] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *Proc. ACM SIGMOD Conference on Management of Data*, pages 298–307, 1991.

[Run92] Elke 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.

[SDA94] Cassio Santos, Claude Delobel, and Serge Abiteboul. Virtual Schemas and Bases. In *International Conference on Extending Data Base Technology*, Cambridge, March 1994.

[SLT91] M. H. Scholl, Christian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *Proc. DOOD, Munich, Germany*, 1991.

[SS89] John J. Shilling and Peter F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In *Proc. OOPSLA, New Orleans, Louisiana*, 1989.

[SW93] Cassio Santos and Emmanuel Waller. *The O_2 Views User's Manual, version 1.0*, December 1993.

[Tem86] Nicole Temmerman. *Les Problèmes de Vues et d'Intégrité dans les Bases de Données Relationnelles et le SGBD Relationnel SABRE.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 1986.

[TYI88] Katsumi Tanaka, Masatoshi Yoshikawa, and Kozo Ishihara. Schema Virtualization in Object-Oriented Databases. In *Proc. IEEE Data Engineering Conf.*, pages 23–30, Los Angeles, Feb. 1988.

[WZ89] Peter Wegner and Stanley B. Zdonik. Models of inheritance. In *Proc. of the Second International Workshop on Database Programmig Languages*, Oregon, 1989.

[Zic91] Roberto Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *IEEE Data Engineering Conference*, Kobe, Japan, 1991.