Extending the ODMG Standard with Views

Jesús García-Molina, María-José Ortín-Ibáñez, Ginés García-Mateos
Departamento de Informática y Sistemas. Universidad de Murcia
30071 Campus de Espinardo, Murcia, Spain
{ jmolina, mjortin, ginesgm }@um.es

Abstract. Views are an important functionality provided by the relational database systems. However, commercial object-oriented database systems do not support a view mechanism because defining the semantics of views in the context of an object-oriented model is more difficult than in the relational model. Indeed, views are not included in the ODMG standard. In this paper, we present a proposal aimed at including views in the ODMG, by extending the object model and the object definition language (ODL). We consider object-oriented views as having the same functionality as relational views. Views are included in the object model in such a way that i) views are a new kind of data type definition, just as are classes, interfaces and literals, ii) an *IS-VIEW* relationship is introduced in order to specify the derivation of a view from its base class, and iii) a view instance preserves the identity of its base instance. A view can import attributes, relationships and operations from its base class, and it can also add new operations, derived attributes and derived relationships. The extent of the view is defined by an object query language (OQL) predicate. We also describe a C++ binding showing the practicability of the proposed model.

Keywords. View model. View management. Object-oriented database system. ODMG standard.

1. Introduction

Since the object-oriented (OO) data model is more complex than the relational model, commercial object-oriented database management systems (OODBMS) do not provide some of the important functionalities supported by relational database systems. This fact has contributed to OODBMS not achieving the success expected [4]. Views are an example of such functionalities which have not been made available in OODBMS. In the past decade, a lot of effort has been devoted to defining the view concept for OO data model and several OO view models have been proposed [1,6,7-11]. However, despite this research, today the commercial systems do not support views and the ODMG standard [5] does not include views, although they are recognized as an indispensable functionality in OODBMS.

In this paper, we present a proposal to include views in the ODMG and in such a way that they preserve the same functionality as relational views. The inclusion of views in the standard requires extending its components: the object model, the object definition language (ODL) and the binding to the programming languages. In our proposal, *views are a new kind of data type* added to the types already existing in the ODMG object model: classes, interfaces and literals. Views are types which are defined by an *IS-VIEW* relationship applied to a base class (or view). The instances of a view preserve the identity of the instances of the base class and thus a view does not generate new instances. The definition of a view is made up of three components: i) list of features (attributes, relationships and operations) imported from its base class, ii) list of additional features (operations, derived attributes and derived relationships) and iii) an object query language (OQL) predicate that determines the extent of the view. Any proposal for extending the ODMG has to show how the extensions are mapped into the programming languages. In our case, the proposed view model has been

mapped into the C++ language.

Other proposals for extending the ODMG standard have been published. In [2,3] the ODMG object model is extended with time and composite objects respectively. To our knowledge, the object-oriented view models proposed to date have not been defined in the context of the ODMG standard. Moreover, little work has been done with regard to the implementation aspects of OO view models. Therefore, the contribution of our work is twofold: we have defined a view model for the ODMG and we have dealt with implementation issues by defining a C++ binding which shows the practicability of the proposed view model.

The rest of this paper is organized as follows. In Section 2, we present our main design choices. Next, we summarize the basic concepts of the ODMG object model. In section 4, we describe the features of the proposed view model by showing how to extend the ODMG object model and the ODL to include the definition of views. In Section 5, we explain the binding between ODL specifications including views and C++. In Section 6, we describe how to access the objects stored in the database through views. Finally, in Section 7, we present our conclusions and future work is outlined.

2. Main Design Choices

Throughout this paper we use the following terminology. The term *base class* denotes the class (or view) from which a view is derived. The global *database schema* contains all the classes, whereas an *external schema* contains a set of views and a set of classes imported from the global schema. We distinguish between *stored objects* (instances of a class) and *view instances* (instances of a base class which also belong to the view).

As is indicated in [6,7], the two main dimensions in the design of an OO view mechanism are the *placement of views in the database schema* (a class hierarchy), and the *nature of view instances*. The design choices in each dimension are influenced by the data model and the view functionality chosen. Obviously, our data model is the ODMG object model. With regard to the view functionality, we consider it appropriate that OO views can be used for the same purposes as relational views, that is: i) definition of external schemas (data independence), ii) content-based authorizations (data protection), and iii) shorthand for queries.

Some proposals have explored the applicability of views to the simulation of database schema evolution [7,8,9,11]. However, as is pointed out in [8], we think that only a range of changes to a database schema should be simulated by views. Since a view is a mechanism for deriving data from the stored database, the simulation of changes involving database restructuring –such as the definition of new attributes not derived from existing data– complicates the view model excessively. These modifications are expensive and make the dynamic addition and removal of views very difficult, whereas such operations are common in relational systems. We think that a view model should preserve the inde-

pendence between the conceptual schema and the view mechanism, that is, the structure of objects stored in the database should not be modified when new views are defined.

A view mechanism supporting the definition of external schemas upon which it is possible to develop OO applications, requires OO views to have the same nature as classes and interfaces, just as a relational view can be used as a table. Therefore, a view should be a new kind of data type. In this way, a view can be used in any context on which a class or interface may appear, for example, if static typed is assumed, then the type of a program entity (attribute, parameter or local variable) could be a view.

2.1 Placement of Views in the Database Schema

Since a view is a data type and the database schema is a type hierarchy, when solving the view placement problem the semantic of the subtype/supertype inheritance-based relationships has to be kept in mind. A detailed discussion about possible solutions is presented in [7], where a new view hierarchy, separated from the class hierarchy, is put forward as the most appropriate solution; each view is connected to its base class by the *view derivation* relationship defined in [1] and a *view inheritance* relationship, similar to the class inheritance relationship, is introduced. This approach has also been considered in other models [8,11]. In our proposal, we introduce both the view derivation relationship and the view inheritance relationship in the ODMG object but only one single class hierarchy is obtained in the C++ binding and there is no separated hierarchy for views.

2.2 Nature of the View Instances

As is indicated in [7,10], two kinds of views can be identified: *object-preserving views*, whose instances have the same object identifier as the base instances from which they have been generated, and *object-generating views*, which create new instances, generating new object identifiers. Object-preserving views facilitate the implementation of the view mechanism and the convenience of using them is described in [10], where it is indicated that object preservation is crucial for a flexible and straightforward view definition facility. In fact, in an OO model the view update problem is closely related to the nature of view instances [7,8,10] because of the existence of object identifiers (object identity). In an object-preserving model, an update on a view instance is automatically propagated to the base instance, whereas in an object-generating model, it is necessary to hold the correspondence between every generated view instance and its base instances, and some updates are not possible. Obviously, object-generating views are necessary for supporting the simulation of the complete range of database schema changes. Bearing in mind the view functionality that we have chosen, our proposal includes object-preserving views and so, a view can add only operations, derived attributes and derived relationships.

3. The ODMG 3.0 Object Model

In this section we summarize the basic concepts of the ODMG 3.0 object model. In the following section, we describe how views can be included in this object model. A detailed description of the ODMG 3.0 standard, including the Object Definition Language (ODL) and the Object Query Language (OQL), can be found in [5].

Objects and Literals. The basic modeling primitives are the object and the literal. An object has a *state*, defined by the values of its properties, and a *behavior*, defined by the set of operations that can be invoked on the object. Each object has a unique *object identifier* which does not change for the entire object lifetime, even if its attribute values or relationships do. A literal does not have identifier and its value cannot change. Both objects and literals are categorized by their *types*.

Literal Types and Object Types. The ODMG object model defines *literal types* and *object types*. *Atomic object types* are user-defined types, defined by *type specifications*. A type specification may be either an *interface*, defining only the abstract behavior of an object type, or a *class*, defining both abstract behavior and abstract state of an object type. By contrast, a *literal definition* defines only the abstract state of a literal type.

Throughout this paper we use as running example a database schema representing employees' information managed by a company (i.e. a university); this schema could include the *Person* interface and *Department*, *Employee* and *Professor* classes, whose ODL declarations are shown in Fig.s 1 and 2.

```
interface Person {
 attribute string name;
 attribute string address;
 attribute enum gender { male, female };
 attribute set<string> phones;
 unsigned short age ( );
};
```

```
class Department
( extent departments )
{
 attribute string name;
 attribute string officeId;
 relationship set<Employee> workers
      inverse Employee::works_for;
};
```

Figure 1. ODL Specifications of *Person* and *Department* types

A type specification defines the external characteristics of the object type –visible to users of the type– as the *operations* that can be invoked on its instances, and the *properties* whose values can be accessed. The ODMG object model defines two kinds of properties: *attributes* and *relationships*. As is shown in the example, attribute declarations may appear in either a class or an interface specification. In a class, attribute declarations define the abstract state of its instances, whereas in an interface defines only abstract behavior. Relationships are defined between two object types –n-ary relationships are not supported– and may be one-to-one, one-to-many or many-to-many, depending on how many instances of each type participate in the relationship. The definition of a one-to-many relationship between *Department* and *Employee* types is shown in Fig.s 1 and 2: each employee is associated to a single department via *works_for* traversal path, and each department is connected with a set of employees via the *workers* traversal path. A relationship declaration in a class defines the abstract state for storing and the set of operations for accessing the relationship,

whereas within an interface it defines only the operations of the relationship, not the state.

Subtyping and Inheritance of State and Behavior. The ODMG object model defines two subtype/supertype relationships: *IS-A* and *EXTENDS*. The *IS-A* relationship defines the inheritance of behavior between object types. Only interfaces and classes can inherit from interfaces. The object model supports multiple inheritance of object behavior. On the other hand, the *EXTENDS* relationship defines the inheritance of state and behavior between classes and it is a single inheritance relationship. In our running example, the class *Employee IS-A Person* and the class *Professor EXTENDS Employee*, as is shown in Fig. 2.

```
class Employee : Person
( extent employees )
{
 attribute Date birthDate;
 attribute float salary;
 attribute Date hireDate;
 attribute Employee manager;
 relationship Department works_for
       inverse Department::workers;
 void raiseSalary (in float amount);
 unsigned short yearsInCompany ( );
};
```

```
class Professor extends Employee
( extent professors )
{
  attribute enum rank {full, assoc, assist};
  attribute set<string> degrees;
  short grant_tenure ( );
};
```

Figure 2. ODL Specifications of *Employee* and *Professor* classes

The *extent* of a type is the set of all instances of the type within a particular object database management system. Thus, every instance of the *Professor* class is a member of the extent of *Professor*, named *professors* in the corresponding ODL specification.

The subtype/supertype relationships (*ISA* and *EXTENDS*) involve polymorphism: where an object of the supertype can be used, an object of the subtype can be used instead. The extent of a subtype is a subset of the extent of its supertype; in the example in Fig. 2, *professors* is a subset of *employees*. An object's *most specific type* is the type that describes all features of the instance.

4. Introducing Views in the ODMG Object Model and ODL

To support a new concept in the ODMG object model, it is necessary to make some changes or extensions in the rest of its components. In our case, the inclusion of the view concept has meant extending the ODL and the bindings to the programming languages. However, it has not been necessary to make any modification in the query language, OQL, because views have the same nature as the rest of ODMG data types. In this section we describe how the object model and the ODL have been extended to include views.

We propose to introduce views as a *new kind of data type* and thus views could be used in the definition of a database schema. Just like classes and interfaces, views define the abstract state and/or the abstract behavior of an object type. Both the behavior and state of a view can include a portion of the behavior and state of its base class, and can also add new operations, derived attributes and derived relationships. The extent of the view, a subset of the extent of its

base class, is defined by an OQL predicate which we denominate *view invariant*[1].

4.1 Definition of Views

The structure of a view specification is shown in the following ODL code defining *V* as a view of the *C* base class:

```
view V is_view_of C
{  imported_features {
            list-of-attributes
            list-of-relationships
            list-of-operations
   }
   additional_features {
            list-of-derived-attributes
            list-of-derived-relationships
            list-of-operations
   }
   invariant is_V { OQL predicate };
};
```

Thus, a view definition may include the following:

1. The view identifier
2. A specification of the base class from which the view has been derived. A view can have only one base class, because the view instances preserve the identity of the stored objects. Nevertheless, we will show later how it is possible to express views derived by a join operation.
3. A list containing the base class features which have been included in the view.
4. An abstract specification of the additional features, when the view adds new operations, derived attributes or derived relationships.
5. The view invariant, named in the form *is_<view-name>*.

We note that *imported_features, additional_features* and *invariant* declarations are optional but at least one of them has to be specified. The schema of our running example could include a view of the *Employee* class, named *SeniorEmployee*, the definition of which is given in Fig. 3.

```
view SeniorEmployee is_view_of Employee
{
 imported_features {      // features extracted from the base class
    attribute string name;
    relationship Department works_for;
    void raiseSalary ( in float amount );
 }
 additional_features {
   attribute unsigned short startYear { self.hiredate->year };
   attribute SeniorEmployee seniorManager { self.manager };
   void retire ( );
 }
 invariant is_SeniorEmployee { self.yearsInCompany > 10 };
};
```

Figure 3. ODL specification of the *SeniorEmployee* type, as a view of the *Employee* class

[1] We use the *view invariant* term because it corresponds to the concept of *class invariant*. We think that this term is more appropriate than the usually used *view query* term because the ODMG object model considers that classes have only an intensional –but not extensional– nature, and thus the queries are applied on object collections.

4.2 Nature of the View Instances and Placement of Views

Just like interfaces, *views are types that are not directly instantiable*. An instance of an interface is always a direct instance of a class that implements the interface. In a similar way, a view instance is a direct instance of its base class. In the previous example, *SeniorEmployee* is a view of *Employee*, so *SeniorEmployee* instances take their identity from the corresponding *Employee* instances. In this way, the objects stored in the database are not affected by the introduction of a new view in the schema. It is just that a stored object can act as an instance of a view. For example, an instance of the *Employee* class satisfying the *is_SeniorEmployee* predicate could act as an instance of the *SeniorEmployee* view. Therefore, from the point of view of the type compatibility, a view is compatible with its base type (i.e. an object whose type is *SeniorEmployee* may be connected to an *Employee* type variable).

With regard to the problem of inserting views in the schema, we use the *view derivation relationship* presented in [1,7]. We refer to this relationship as *IS-VIEW*. The alternative choice is to integrate every view in the class hierarchy as a subclass of its base class [9,10], but this strategy requires a data model with multiple classification because a view instance would also be an instance of its base class. Moreover, the extent of a view is a subset of the extent of its base class, but the set of features of a view does not have to be a subset of the features of its base class.

4.3 Features of a View

Importation of Features by a View. A view may import attributes, relationships and operations already defined in the base class, albeit directly or inherited from any supertype. In the example, *SeniorEmployee* imports the *name* attribute, the *works_for* relationship and the *raiseSalary* operation.

Given a relationship *R* between two classes *C1* and *C2*, a view derived from *C1* or *C2* can import *R*. For example, *SeniorEmployee* imports the one-to-many relationship defined by the *works_for* and *workers* traversal paths. When a view imports a relationship, the users of the view can only access a subset of the instances of the original relationship. The *works_for* relationship imported by *SeniorEmployee* consists in pairs <*set of Employee*, *Department*> where the employees are instances of the view –and not all the objects of *Employee*. Thus, the original relationship between *Employee* and the *Department* class is viewed as a relationship between *SeniorEmployee* and *Department*. It is worth noting that the inverse of *workers* in *Department* is still *works_for* in *Employee.*

It should be observed that only the features imported from the base class are accessible in the view because the *IS-VIEW* relationship is not an inheritance relationship. Thus, *SeniorEmployee* does not include *salary*, *hireDate*, *manager* or *yearsInCompany* features of *Employee*, nor other features of the supertype *Person*.

Definition of New Features in a View. A view may add new features. In the example, *SeniorEmployee* adds a new operation named *retire*, and two new attributes, *startYear* and *seniorManager*. Every new attribute introduced in a view

always has to be a derived attribute whose value we know how to obtain. Thus, *startYear* is derived from the *hireDate* attribute by calculating the year of joining company, and *seniorManager* is derived from the *manager* attribute by returning either the manager, if he or she is also a senior employee, or the *null* value. OQL can be used to express the derivation expressions of new operations and derived attributes.

A view may include *nested properties* of the base class. That is, a view can add an attribute defined by a path expression starting from the base object. In our example, the *SeniorEmployee* view could include an attribute *workOffice* whose definition would be the following:

```
attribute string workOffice { self.works_for.officeId } ;
```

In addition to operations and derived attributes, a view may add new derived relationships. Given a relationship *R* between two classes *C1* and *C2*, and letting *V1* and *V2* be two views derived from *C1* and *C2* respectively, then it is possible to define a new relationship *R'* between *V1* and *V2* if and only if *R'* is derived from the original relationship *R*. A *derived relationship R'* defines a subset of the possible logical connections between the objects participating in the original relationship *R*. In our example, the relationship defined between the *Employee* and *Department* classes could be used to derive a new relationship between *SeniorEmployee* and a view of *Department*. A derived relationship, in a similar way to derived attributes, does not mean the addition of any new field to the view instances.

4.4 The Invariant of a View

The *view invariant* is an OQL predicate expressing the constraints that an instance of the base class must satisfy in order to act as a view instance. In the *SeniorEmployee* view, the invariant expresses that a senior employee is an employee who has worked at the company for more than ten years, so the extent of the *SeniorEmployee* view is the subset of employees whose *yearsInCompany* operation returns a value greater than 10. Just like an operation, a view invariant is always applied on an instance of the base class (the *base object*), and thus a path expression within the OQL predicate may start from either a variable connected to an object or from the reserved word *self* denoting the base object. An example of each kind of path expression appears in the *FloralPerson* view, shown below, whose view invariant contains *f.name* and *self.name* expressions. If a path expression does not start from either a variable or *self*, the base object is assumed.

As we mentioned before, a view definition may not to include any invariant declaration. In that case a *true* is assumed and therefore all the instances of the base class also belong to the view.

Joins in the View Invariant. It is worth noting that the view invariant specification allows us to express explicit joins, and thus the view can involve some more classes in addition to the base class. In the ODMG standard the explicit join is illustrated by an example selecting “the people who bear the name of a flower”. In our approach, we could define the

following view derived from the *Person* class:

```
view FloralPerson is_view_of Person {
  imported_features  { ... }
  additional_features  { ... }
  invariant is_FloralPerson {  exists f in Flowers: f.name = self.name  };
};
```

where *Flowers* denotes a collection of instances of the *Flower* class. In this view, the invariant includes an explicit join expressing that the name of a flower must be equal to the name of the person.

Views with Features Imported from More than One Class. Since a view has only one base class, all features imported by the view belong to its base class. Therefore, the *FloralPerson* view could only import features from the *Person* class but not from the *Flower* class. This restriction is originated by our choice of a view semantics with object-preservation. In any case, operations or derived attributes could be used in order to include in a view operations and attributes belonging to those types involved in the view invariant and different to its base class. In the example, the *FloralPerson* view could include a *scientificName* operation that, given a *Person* instance belonging to the *FloralPerson* view, will return the value of *scientificName* attribute in the corresponding instance of the *Flower* class of the same name.

4.5 Views and Inheritance

A view defines the state and behavior of its instances. Both of them are always extracted from the base class and so they do not belong to the view itself. In addition, just like an interface, a view is not instantiable. Therefore, according to the nature of views, it does not make any sense to define subtype/supertype relationships between classes (or interfaces) and views. However, a subtype/supertype relationship between views could be defined. Hence, we have considered the *view inheritance relationship* introduced in [7], which has been revised and adapted to the ODMG context. This inheritance relationship has been named *V_ISA* and is orthogonal to the *view derivation* relationship. The whole set of possible supertype/subtype relationships in ODMG is summarized in Table 1. It is worth noting that the *V_ISA* relationship has a different semantics to the *ISA* relationship, although there are some common properties such as multiple inheritance and a polymorphic nature.

Table 1. Subtype/supertype relationships in ODMG extended with views

	Supertype		
Subtype	interface	class	view
interface	IS-A	-	-
class	IS-A	EXTENDS	-
view	-	-	V_IS-A

The following figure, which is an adaptation of Fig. 2-2 in [5], shows the possible relationships between views, interfaces and classes in the ODMG object model extended with views and it indicates whether multiple or single inheritance is allowed; it also includes the view derivation relationship.

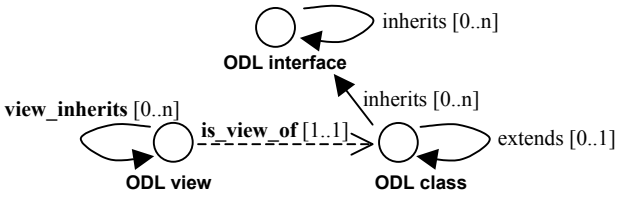

Figure 4. Class-Interface-View Relationships

In order to describe the *V_ISA* relationship, we use the ODL type hierarchy shown in Fig. 5 and based on our running example, where the *SeniorEmployee* view inherits from *ReliablePerson.*

The *V_ISA* relationship is defined as follows. Let a view be a subtype (a *subview*) of another view (the *superview*), then i) the subview inherits all the features of the superview and the subview may add new features (that are always either imported from its own base class or new operations and derived properties), ii) the *V_ISA* relationship involves polymorphism: an instance of the subview can act as a superview instance, and iii) the instances of the subview are those instances of its base class that satisfy both the subview and superview invariants; therefore, the subview instances are a subset of the superview instances. To ensure this, the following constraints have to be satisfied: i) *the base class of the subview must be either the base class of the superview or a subclass of the base class of the superview,* and ii) *the invariant of the view must be added to the supertype invariant*.

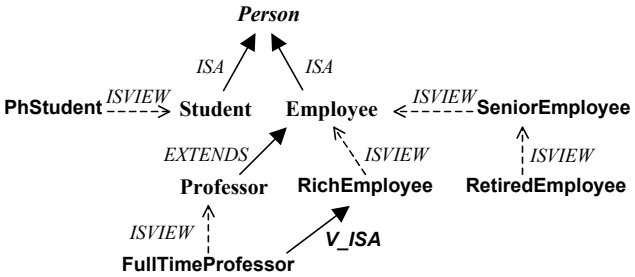

Figure 5. An ODL type hierarchy including views

In our example, the *FullTimeProfessor* view has been declared as a subview of the *RichEmployee* view. The *FullTimeProfessor* view could also inherit from *SeniorEmployee* or *RetiredEmployee* views. However, it could not be defined as a subview of the *PhStudent* view because this originates a situation in which multiple classification is required: supposing that *FullTimeProfessor* inherits from *PhStudent*, then every *FullTimeProfessor* instance –a *Professor* instance– is also a *PhStudent* instance, that is a *Student* instance.

It is worth noting that we could define a new view of *Employee*, named *RichSeniorEmployee*, which is a subview of both *RichEmployee* and *SeniorEmployee* views.

Finally, we can point out the following properties of our proposed view concept. Let *VC* be a view whose base class is *C*, then:

- The extent of *VC* is a subset of the extent of *C*. The instances of *VC* are the instances of *C* that satisfy the view in-

variant. Obviously, as the instances of any subtype of *C* belong to the extent of *C*, they will also be instances of *VC* if they satisfy its invariant.

- The features of *VC* are those declared in its definition. These can be features imported from *C* or additional features (operations, derived relationships and derived attributes). *VC* does not implicitly inherit other features from *C* or from the supertypes of *C*.
- The *VC* type is compatible with *C*, that is, an object of type *VC* may be assigned to a variable of type *C*. The assignment of an object of type *C* to a variable of type *VC* requires a dynamic check of the view invariant for the assigned object.
- The ODMG supports parametrized collection objects such as *Set<T>* or *Bag<T>* which implement the *Collection<T>* interface. Types *Collection<VC>* and *Collection<C>* are compatible with each other. The assignment between them is always allowed. Since *VC* instances are also *C* instances, the assignment of an object of type *Collection<VC>* to a variable *cc* of type *Collection<C>* does not suppose any problem, whereas the assignment of an object of type *Collection<C>* to a variable *vcc* of type *Collection<VC>* requires the dynamic check of the view invariant for all objects in the assigned collection because *vcc* must contain only the objects that satisfy the invariant. The semantic of this assignment is discussed later in Sections 5.6 and 6.

Optionally, if *VC* is declared as a subtype of another view *V*, then:

- *V* must be either a view of *C* or a view of any (direct or indirect) supertype of *C*.
- An inheritance relationship between views is established (*V_ISA*).
- *VC* inherits all the features of *V*, and the invariant of *VC* includes the *V* invariant.

5. Mapping Views Into C++ Classes

The ODMG standard defines the binding between the object model and the ODL and C++, Smalltalk and Java programming languages. Therefore, our proposal must extend these bindings in order to map the view concepts introduced in the object model into specific constructs of these OO programming languages. This mapping has to solve issues such as: how to place views within the class hierarchy, how to ensure that the user of the view only accesses a portion of the information, and how to solve compatibility problems between types. In this section, we describe the architecture of an OO database system supporting views and the mapping of views into C++ classes.

5.1 Database System Architecture

Fig. 6 shows a global view of the architecture of a database system supporting the proposed view model. This figure is a new version of figure 5-1 in the specification of ODMG [5]. It is modified in order to manage ODL specifications with views, and to map views into C++ classes.

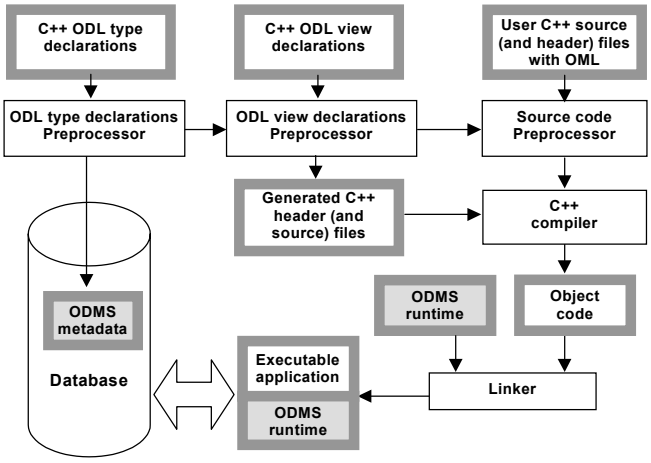

Figure 6. ODMG DBMS architecture including views

The database schema is defined by ODL declarations and the database applications are written in a high level programming language (we assume C++). As Fig. 6 shows, we separate the standard ODL type declarations from the view declarations because views are derived once the conceptual schema has been established. Therefore, the declaration of classes and interfaces in the conceptual schema is not dependent on the set of defined views. The database conceptual schema is not modified when a new view is defined.

Fig. 6 includes two preprocessors related to views: the *ODL view declarations preprocessor*, and the *Source code preprocessor*. The first one translates view declarations into C++ class definitions. A class implementing a view could be implemented either within the database application code (if the proper access privileges on the base class have been granted to the programmer) or by the database designer–the class here being imported into the application. The ODL view declarations preprocessor receives the result of the ODL type declarations preprocessor as input, because it needs to know the class declarations from which the views are derived. The ODL type declarations preprocessor included in the standard also has to be modified to perform the mapping into C++ classes in a slightly different way on account of the problem explained below in Section 5.2. The output of the ODL view declarations preprocessor is a set of header and source files to be included in the C++ application.

The translation of views into class declarations must be completed by also preprocessing the application source code. This step ensures that an application only uses the types on which it holds the proper access privileges. Finally, the compiler receives the generated C++ header and source files with the structure of the ODL types (including views), and the application source code.

5.2 Mapping ODL Classes

The ODMG standard establishes a syntactic equivalence between attributes of ODL types and data member of C++ classes, so an ODL class attribute is mapped into a public data member of the corresponding C++ class. Here, we introduce a modification to the standard because, as we will see later, it is necessary for a subclass to be able to modify the

C++ data member export status. Let *B* be a subclass of *A*, C++ allows *B* to redeclare any function member by changing its export status, but it does not allow this kind of redeclaration for data members.

Therefore, in our model each ODL class attribute is mapped into a protected data member and two public function members[2], one that returns the attribute value and another that allows the direct modification of the attribute (*get* and *set* methods). To increase efficiency, the compiler may detect calls to public methods that only return the value of an attribute and may handle them just like a direct access to such an attribute. Fig. 7 illustrates how ODL class declarations are translated into C++ class definitions. It should be noticed that all methods are declared as virtual, and since repeated inheritance will arise in the C++ class hierarchies obtained, virtual inheritance is used.

ODL Classes	C++ Classes

class A
{
  readonly
  attribute long q;
};
	Class A: public d_Object {
	 public:
	  virtual d_Long q (void);
	 protected:
	  d_Long _q;
	};

class B extends A
{
  attribute short p;
};
	Class B: public virtual A {
	 public:
	  virtual void p(d_Short value);
	  virtual d_Short p (void);
	 protected:
	  d_Short _p;
	};

Figure 7. An example of the mapping of ODL classes into C++ classes

5.3 Mapping ODL Views

In our work, one of the main issues that we have tackled is the translation of views into classes in the chosen programming language. In particular, it is necessary to find out how to translate the *IS-VIEW* relationship into C++ constructs. Bearing in mind the aforementioned problems arising when integrating a view as a subclass of its base class (primarily, multiple classification is needed), we have devised the following mapping.

A view *VC* derived from the base class *C* is mapped into a C++ abstract class with the same name. In the C++ hierarchy, this class is a new superclass of the class *C* and a subclass of all the classes corresponding to the supertypes of *C* within the ODL type hierarchy. That is, *VC* is placed between *C* and its superclasses. Obviously, this approach needs multiple inheritance because both *C* and *VC* classes may have one or more superclasses. From now on, we refer to the C++ abstract class corresponding to an ODL view as *class-view*.

Since a class-view is an abstract class, it includes a set of abstract methods and is not instantiable. Therefore, the extent of a class-view consists of direct instances of its base class or its subclasses, which are objects stored on database. Fig. 8 shows a graphic representation of an example of mapping of ODL type declarations into C++ classes. In this example, we have derived three views from the class *B* introduced in Section 5.2: *V1* and *V2* are derived from the class

[2] In the C++ context, we use the more general terms *method* and *attribute* instead of *function member* and *data member,* respectively.

B, and *V3* is a view of *V2*. The "*" symbol denotes an abstract class, whereas the "+" symbol denotes an effective (non-abstract) class. Throughout this section, we use as running example the hierarchies presented in Fig. 8.

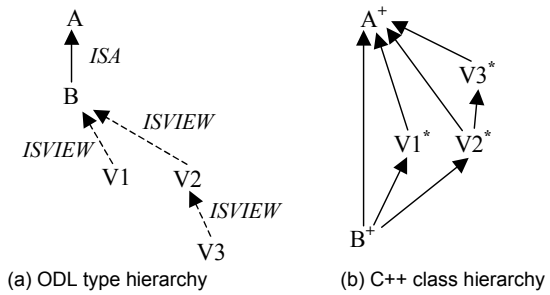

Figure 8. An ODL type hierarchy and its mapping into C++

The definition of views *V1* and *V2* requires the declaration of the C++ class *B* to be modified in order to include the two views as superclasses:

```
class B : public virtual A,public V1,public V2 {
... };
```

Bearing in mind that the ODL class *B* has the *q* attribute, inherited from *A*, and the *p* attribute, which is directly introduced in *B* (see Fig. 7), then we can distinguish several mapping cases according to whether the attributes *p* and *q* are imported or not by the *V1* view, and whether the view introduces new features or not. Although only attributes have been considered, the discussion for each case is also suitable for operations because the mapping of the ODL attributes involves operations.

- ***V1* imports the *p* attribute from *B***

  The *V1* class-view has to declare the methods corresponding to this attribute as public and abstract. The subclass *B* makes them effective, without any further modifications.

```
class V1 : public virtual A {
   public:
      virtual void    p (d_Short value)=0;
      virtual d_Short p (void)=0;
   ...};
```

- ***V1* does not import the *p* attribute**

  The *V1* class-view does not have to include any method concerning *p*.

- ***V1* imports the *q* attribute introduced in *A***

  The *V1* class-view already inherits the *q* attribute from *A*, so it should not include any method related to *q*. There may exist one or more views derived from *B* and each corresponding class-view establishes an inheritance path between *B* and *A*. Virtual inheritance is applied between each class-view and *A* so that the attributes of *A* are not replicated in *B*.

- ***V1* does not import the *q* attribute introduced in *A***

  C++ public methods corresponding to the attribute *q* have to be redefined in the *V1* class-view. Their export status must be changed from public to protected in order to hide them from the clients, and the method body must be replaced by an empty block. Since such methods are inherited as protected by *B*, they must be redeclared as public in *B*.

```
class V1 : public virtual A {
 protected:
   virtual d_Long q (void){ }; //nothing is done
 ...};

class B : public virtual A,public V1,public V2{
 public:
   virtual d_Long q (void) { return A::q() };
 ...};
```

- ***V1* introduces a new operation or a new derived attribute *t***

  Since a view can only add operations and derived attributes, only methods can be added. The *V1* class-view declares a new virtual method *t*. Since the *B* class inherits this method, *B* has to hide it from its clients and subclasses and therefore, the *t* method is redeclared as private with the same implementation.

```
class V1 : public virtual A {
  public:
    virtual d_Object t (d_String x);
...};

class B : public virtual A,public V1,public V2{
  private:
    d_Object t (d_String x) {return V1::t(x)};
...};
```

With regard to relationships, we have to distinguish between importation and addition. In ODMG, a relationship is mapped into an attribute (used to refer to the other end of the relationship) plus several methods on the relationship target classes. These methods form and drop members from the relationship, provide access and manage the required referential integrity constraints. Therefore, the mapping of an imported relationship is the same as the mapping described for imported attributes and operations. When a derived relationship is added within a view, each relationship target view-class has to include a method which returns the other end plus the methods managing the relationship and which delegate their tasks to the corresponding methods (generated from the ODMG mapping) in the base class.

The existence of the *V2* view does not affect the mapping established for each case, because we have considered virtual inheritance, so the *B* class would share all attributes and methods inherited from each class-view. We have verified that ambiguous situations do not arise. Moreover, these mapping rules can be applied regardless of base type of view – be it a class or a view– within the ODL declaration. In the running example, *V3* is a view of *V2*, then *V3* is mapped into a new superclass of the *V2* class-view and a subclass of the superclass of *V2* –the *A* class.

5.4 The View Invariant

In an ODL view specification, the view invariant is an OQL predicate that expresses the constraints which must be

satisfied by a view instance; let the view *V* be derived from the base class *C*, then every instance of *C* satisfying the view invariant of *V* is also an instance of *V*. Therefore, the view invariant defines the extent of a view as a subset of the extent of its base class.

According to the specialization semantic of the *IS-A* relationship, every instance of a class *C* can act as an instance of any superclass of *C*, thus the extent of a class *C* is a subset of the extent of any superclass of *C* –the extent of *C* is made up of the direct instances of *C* and the instances of its subclasses. However, our approach for mapping views into C++ classes does not follow this semantics strictly because a class-view is a superclass of the base class of that view, and so an instance of the base class can only act as an instance of the class-view if it satisfies the invariant. Nevertheless, our solution works since the class-view is an abstract class which has a single subclass, the base class of that view, and so all the instances of the class-view will also be instances of the single subclass. An entity (variable, parameter) whose type is a view, could only reference instances of the base class, with the only condition that it will be necessary to check the view invariant whenever an instance of the class attempts to act as an instance of a class-view (in a similar way as a type cast). So, we think that the object-oriented paradigm is preserved.

Given the following variable declarations

```
ob: B; ov1: V1
```

where *V1* and *B* are the classes of the example showed in Fig. 8, the assignment

```
ov1 = ob
```

would be legal, although at runtime the object connected to *ob* will only be connected to *ov1* if that object satisfies the view invariant of the *V1* view. For example, an entity whose static type is *SeniorEmployee*, can only be connected to an *Employee* instance satisfying the invariant of the *SeniorEmployee* view.

In the C++ binding, a view invariant is mapped into several C++ methods. First, the *d_Object* class, whose purpose is to enable a definition of a persistent-capable class, is extended with a new method named *check_predicate* whose signature is

```
d_Boolean check_predicate(const char* OQL_predicate);
```

This method has the OQL predicate expressing the view invariant as input and returns *true* or *false* depending on whether the predicate is satisfied or not by the target object. Secondly, every class-view will include an automatically generated method, named like the view invariant in the ODL declaration (i.e. *is_<view-name>*), whose purpose is to check whether the view invariant predicate is satisfied or not. This method has no arguments and invokes the *check_predicate* method and provides it with the view invariant predicate as argument. Finally, each C++ class will include a non-virtual method named *view_invariant* whose purpose is either to invoke the *is_<view-name>* method, in the case of a class-view, or to return *true*, in the case of a C++ class corresponding to an ODL class or interface. In this

way, methods which need to check whether a stored object belongs to a given view, perform the checking through a chain of three invocations: the *view_invariant* method is invoked first, and which in turn invokes the method *is_<view-name>*, which finally invokes the *check_predicate* method.

The implementation of the view invariant through two methods (*view_invariant* and *is_<view-name>*), instead of through a single method, is due to two facts: i) a class can have several views, and ii) some classes, such as iterators or collections, managing instances of any view, need to apply a generic method instead of an *is_<view-name>* specific method to check the view invariant.

Fig. 9 shows the ODL definition of the *V1* view introduced in our running example, and the corresponding class-view in C++. The *B* class inherits the methods *is_v1*, *is_v2* and *is_v3* from the superclasses *V1*, *V2* and *V3*, respectively, and has a *view_invariant* method returning *true*. It is worth noting this method is not declared as virtual because the dynamic type of an entity whose static type is a view will always be the base class of the view or some subclass. In our example, given an entity *ov1* whose static type is *V1*, the dynamic type of *ov1* will be *B* or some subclass of *B*. By defining *view_invariant* as a non-virtual method, the message *ov1->view_invariant()* will always check the invariant of *V1*, instead of returning *true*.

```
view V1 is_view_of B
{
  imported_features {
    attribute short p;
    attribute long q;
  }
  invariant is_V1 { p > 10 };
};
```

```
class V1 : public virtual A {
 public:
  virtual void    p (d_Short value)=0;
  virtual d_Short p (void)=0;
  d_Boolean is_v1 (void) {
            check_predicate("p>10");};
  d_Boolean view_invariant (void) {
            return is_v1();};
};
```

Figure 9. ODL definition of the V1 view and the mapping into a C++ class

For example, given the method *insert_element (const T &element)* belonging to the template class *d_Set<T>,* and supposing the instantiation *d_Set<SeniorEmployee>,* then the invocation of the message *element->view_invariant()* during the execution of the *insert_element* method will provoke the checking of the invariant of the *SeniorEmployee* view on the object connected to the *element* parameter. Management of collections is described in Sections 5.6 and 6.

A view *V3*, whose base type is another view *V2*, will include the invariant of *V2*:

```
d_Boolean is_v3 (void){...};
d_Boolean view_invariant (void) {
     return V2::view_invariant() && is_v3();};
```

In the same way, if a view *V3* is declared as a descendant of another view *V2*, then the invariant of *V3* is and-ed to the invariant of *V2*. Moreover, if both relationships are given at the same time, i.e. if a view *V* is derived from another view and is a subtype of a third view, then all invariants are and-ed within the view invariant of *V* class-view.

5.5 View Access Control

A view mechanism facilitating the definition of external schemas must apply two controls aimed at constraining users to

access to the portion of authorized information: i) the user application can only use the types (classes, interfaces and views) included in the external schema, and ii) an entity whose type is a view can only reference an instance satisfying the view invariant. The first control is performed at compilation time by the C++ source code preprocessor which takes both the application source code and the external schema as input. The second control is performed at runtime by checking the view invariant in order to evaluate when a stored object can act as a view instance.

The ODMG C++ binding defines a template class *d_Ref<T>* to allow *references*[3] (*smart pointers*) between objects (either persistent or not). A variable of type *d_Ref<T>* is a reference to an instance of type *T*, which can be de-referenced using * and -> operators, or assigned to another variable with the = operator. According to the standard, if *D* is a superclass of *C* then a *d_Ref<C>* can be assigned to a *d_Ref<D>*. This kind of assignment is possible through the *d_Ref_Any* class, which is used as an intermediate step in the conversion between different instantiations of *d_Ref*. We have modified the *d_Ref<T>* class in order to check a view invariant. Thus, the source code preprocessor ensures that references to instances of a view *V* are always performed through variables whose type is *d_Ref<V>*, i.e. pointer variables are not used. The view invariant is checked each time that a variable whose type is a view is manipulated: i) it is initialized, ii) an object is assigned to it (the assignment operator has been modified in the *d_Ref* template), and iii) a de-reference is applied on it (* and -> operators have been also modified in *d_Ref*).

If a view invariant is not satisfied, then a *d_Error_RefInvalid* exception is raised, indicating an attempted access to unauthorized information. In any case, the client code could avoid this runtime exception by checking the view invariant before executing the instruction that raises the exception; this checking could be performed by means of the invocation of the method that implements the view invariant predicate in the class-view. Some changes in the state of a view instance may make the view invariant not be satisfied in the new state. For this reason, the view invariant is also checked when a *d_Ref<T>* is de-referenced. This test adds a time overhead to the use of views which is proportional to the time required to execute the invariant. In any case, optimization techniques could be used to partially avoid this increment in the execution time.

5.6 Mapping Collections

Collections play an important role in object-oriented databases because users normally use them to access to the stored information. Now, it is possible to declare collections whose elements are of a view type, for instance *d_Set<SeniorEmployee>*. Since the ODMG standard enforces the compatibility between a collection of a type *T* and a collection of a supertype of *T*, a user will be able to access a collection containing instances of a class *C* through a vari-

[3] ODMG specifies that "a reference in many aspects behaves like a C++ pointer, but provides an additional mechanism that guarantees integrity in references to persistent objects".

able declared as a collection of instances of a view derived from *C*. For example, a user who is allowed to access *SeniorEmployee* but not *Employee*, can obtain a reference to a collection *d_Set<Employee>,* but through a *d_Set<SeniorEmployee>* reference. It is necessary to ensure that the user can only see the employees satisfying the *SeniorEmployee* view invariant.

With regard to the implementation, now a collection needs to know if its elements are being considered as instances of some view. In other words, the behavior of the methods in the classes associated to collections (*d_Set*, *d_Bag*, *d_List*, *d_Array*, *d_Varray*, *d_Dictionary*) and in the class of the iterators (*d_Iterator*) has to be modified to check, when necessary, the view invariant of the type of the collection. This modification only causes some changes in the generic implementation of the collections and iterator templates. These issues are explained in the next section.

6. Accessing Stored Objects through Views

According to the ANSI/SPARC three-level architecture, an external schema offers a vision of the database schema for a particular user or group of users. Therefore, all users associated to an external schema have to hold access privileges on all the types included in that schema. In this section we discuss the definition and use of external schemas defined by the view concept proposed for the ODMG. We pay attention to the issues related to the access from a programming language to the stored objects.

Most object-oriented view models proposed have considered the definition of external schemas [7,9]. In our approach, an *external schema* is defined by the derivation of views and the importation of types (classes, interfaces and views) from the conceptual database schema. As is well-known, an external schema must satisfy the *closure property*: for every type included in the external schema, all types used in the declaration of the attributes and operation parameters of that type must also belong to the schema. A detailed discussion about external schemas and the closure property can be found in [7].

Next, we consider a C++ application accessing a database (a collection of objects belonging to classes included in the base schema) through an external schema and we analyze the issues related to the manipulation of the database from the application.

In ODMG, a logical database[4] is an instance of the *Database* type. The *Database* interface includes the operations for manipulating the *Database* object, such as *bind* or *lookup*. The operation *bind* allows us to connect a name to a stored object. The objects in the database are accessible to an application through these named objects which are the *entry point* to the content of the database. The operation *lookup* allows retrieval of the stored object bound to a name

[4] ODMG specifies that "an OODBMS may manage one or more logical databases, each of which may be stored in one or more physical databases".

supplied as a *String* argument. This operation returns the object as an instance of the root of the class hierarchy, that is the *Object* type. Therefore, an application retrieving objects from the database must perform a type cast in order to manipulate the objects extracted. For instance, let *employees* be the name of a set of instances of the *Employee* class, the following C++ code shows how the *employees* object could be retrieved and assigned to a variable *es*, and the application of the *cardinality* method on this set. The first variable declaration means that *es* is a reference to a set of references to instances of the *Employee* class.

```
d_Ref<d_Set<d_Ref<Employee>>> es;
d_Database db;
…
es = (d_Ref<d_Set<d_Ref<Employee>>>)
                  db.lookup ("employees");
…
int i = es->cardinality ();//get the number of elements in the set
```

When a user is restricted to developing applications on an external schema, then he or she may only use the types included in that schema. Given an external schema including the *SeniorEmployee* view but not the *Employee* class, then the previous C++ code would be written as follows:

```
d_Ref<d_Set<d_Ref<SeniorEmployee>>> ses;
d_Database db;
…
ses = (d_Ref<d_Set<d_Ref<SeniorEmployee>>>)
         db.lookup ("employees");
…
int i = ses->cardinality ();//get the number of elements in the set
```

Now, the *employees* set is manipulated from the viewpoint provided by the *SeniorEmployee* view. Both *es* and *ses* variables reference the same set of *Employee* objects, but the static type of the actual parameter of the collection is different: *Employee* class and *SeniorEmployee* class-view respectively (as we explained above in Section 5.6, the types *d_Set<Employee>* and *d_Set<SeniorEmployee>* are compatible). Whereas all instances of the *employees* set can be accessed through a *d_Set<Employee>* reference, only the instances that satisfy the *SeniorEmployee* view invariant can be accessed through a *d_Set<SeniorEmployee>* reference: the *ses->cardinality()* message returns the total number of senior employees, instead of the total number of employees returned by the *es->cardinality()* message.

Therefore, the checking of the view invariant should be included within the implementation code of the methods that manipulate collections (*insert_element*, *remove_element*, etc.) because, given a class-view *V*, a *d_Collection<V>* reference allows access to a collection containing instances of the base class of *V*, but this reference should only allow access to instances of view *V*.

However, collections are not the only objects that can be stored in a object-oriented database. Objects of any type can be persistent (orthogonal persistence). For example, let us suppose that a named object *john*, referencing an instance of the *Person* class, is defined in the *db* database instance. A user restricted to using the *SeniorEmployee* view could retrieve the object *john* by executing the following assignment:

```
d_Ref<SeniorEmployee>  se;
```

```
…
se = (d_Ref<SeniorEmployee>)db.lookup("john");
```

Just as happens with a variable referencing a collection of view instances, the *se* variable allows access to the *john* object through the features of the view *SeniorEmployee*.

During the execution of a type casting, an exception is raised when the dynamic type of the object is not compatible with the static type of the variable. Furthermore, the assignment may now raise another exception if the assigned stored object violates the view invariant. In order to emphasize this difference, this type conversion may be called *view casting*, a particular case of type casting. It should be noticed that a reference whose static type is a view-class (for example, *SeniorEmployee*) can always be assigned to a variable whose type is the base class of that view (*Employee*).

7. Conclusions and Future Work

In this paper, we have presented a proposal for the inclusion of views in the ODMG standard. Firstly, we have discussed our main design decisions, assuming that OO views have the same functionality as relational views. We have presented the basic concepts of the ODMG object model and then we have described how the object model, the object definition language and the C++ binding have been extended.

Views have been included in the object model in such a way that views are a new kind of ODMG data type and the *IS-VIEW* relationship specifies the derivation of a view from its base class. The definition of a view specifies the list of features imported from its base class, the list of additional features and the view invariant (an OQL predicate defining the view extent). In a view model supporting capacity-augmenting views (that is, views that may add non-derived attributes), object restructuring is needed and object updates are more complicated. Hence, in our view model we have considered object-preserving views –that is, each view instance preserves the identity of its base instance– and views can only add derived attributes (and relationships) and new methods. Although in our approach, a view can only have a single base class, we have shown how several classes could be involved in a view declaration. Since views are a new kind of data type, it is not necessary to change the object query language, OQL.

Regarding the binding to the programming languages, we have described a C++ binding. Each view is mapped into a C++ abstract class which is placed, in the class hierarchy, between the C++ base class and the superclass of this base class. With this implementation technique it is possible that a view instance preserves the identity of its base instance. We have discussed how the polymorphism is affected by integrating the view as a superclass of its base class: an instance of a base class can only act as a view instance if the view invariant is satisfied, in a similar way to a type cast. The process to generate C++ code from ODL declarations has been modified by introducing two additional preprocessors in order to manage views in ODMG. Finally, we have described how to manage collections containing view instances.

We believe that the C++ binding obtained is practical. It can be implemented on a commercial OO database system because it does not impose hard restrictions. Besides, the object-oriented paradigm is preserved and the database programmer may write applications in the usual way. However, it is difficult to translate this binding into Smalltalk and Java, because our approach to mapping views into C++ classes is based on multiple inheritance of classes and on some specific properties of C++, such as the use of non-virtual methods (in the implementation of the view invariant). At present, we are working on the definition of Java and Smalltalk bindings. In the Java binding, the multiple inheritance of interfaces could be useful in order to include views in the class hierarchy.

Our future work also includes, i) to extend the ODMG metadata descriptions, and ii) to deal with the formalization of the view model in a similar way as is done in the formal view model presented in [7].

8. References

[1] Bertino, E.: A View Mechanism for Object-Oriented Databases. Proceedings of the 3rd International Conference on Extending Database Technology (EDBT), (1992) 136-151

[2] Bertino, E., Ferrari, E., Guerrini, G., Merlo, I.: Extending the ODMG Object Model with Time. Proc. of 12th European Conference on Object-Oriented Programming (ECOOP), Bruxelles (Belgium), July 20-24 LNCS, No.1445, Springer-Verlag (1998).

[3] Bertino, E., Guerrini, G.: Extending the ODMG Object Model with Composite Objects. Proceedings of ACM Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98) Conference, Vancouver (Canada), Oct. 18-22, ACM Press (1998)

[4] Carey, M.J., DeWitt, D.J.: Of Objects and Databases: A Decade of Turmoil. Proceedings of the 22nd VLDB Conference, Bombay, India (1996) 1-12

[5] Cattell, R.G.G. et al.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann Pub. (2000)

[6] García-Molina, J., Guerrini, G., Bertino, E., Catania, B.: Dimensions in the Design of an Object-Oriented View Mechanism. Proceedings of the 1st Spanish Conference on Databases JIDBD, A Coruña (1996) 119-129

[7] Guerrini, G., Bertino, E., Catania, B., García-Molina, J.: A Formal Model of Views for Object-Oriented Database Systems. Theory and Practice of Object Systems, Vol. 3(3), (1997) 157-183

[8] Kim, W., Kelley, W.: On View Support in Object-Oriented Database Systems. In: Kim, W. (ed.): Modern Database Systems. ACM Press (1995) 108-129

[9] Kuno, H.A., Rundensteiner, E.A.: The MultiView OODB View System: Design and Implementation. Theory and Practice of Object Systems, Vol. 2(3) (1996) 203-225

[10] Scholl, M.H., Laasch, C., Tresch, M.: Updatable Views in Object-Oriented Databases. Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases. LNCS, No. 566 (1991) 189-207

[11] Souza dos Santos, C.: Design and Implementation of Object-Oriented Views. Proceedings of the 6th International Conference DEXA. LCNS, No. 978 (1995) 91-102