Views for Information System Design without Reorganization

Z. Bellahsene[1] P. Poncelet[2] M. Teisseire[1]

[1] LIRMM - UMR CNRS 9928 - Université Montpellier II
161 rue Ada 34392 Montpellier Cedex 5 FRANCE
E-mail: {bella,teisseir}@lirmm.fr
[2] LIM - URA CNRS 1787 - Université de la Méditerranée
E-mail: poncelet@lim.univ-mrs.fr

Abstract. In this paper, we propose an approach for managing structural evolutions through views. Evolutions are not actually performed but simulated, thus the database schema is preserved for both users and existing applications. Furthermore, the approach avoids costly database reorganizations. This results from the multi-schema architecture which is introduced and from the isolated management of additional data.
In our approach, the concept of virtual class is complemented through the notion of virtual schema, encompassing several classes and their possible inheritance or derivation relationships with their own properties. The inter-view visibility is enhanced with the main advantage of improving view re-use.

1 Introduction

In relational systems, views are widely used for controlling access, simplifying queries, or providing users with a mere and accurate vision of data as regards their needs.
In an object-oriented context, the application scope of views has been extended [1, 5, 14, 10, 11, 17]. In particular, some approaches consider views as a relevant mechanism when managing structural evolutions [5, 10, 17]. Actually, modifications of an Object-Oriented Database (OODB) schema can be captured through views, without altering the DB schema and thus without reorganizing the database and maintaining existing application programs. In fact, structural evolutions are simulated through views and not actually performed within the DB schema.

Views definition for OODBMS has been addressed in different approaches among which two trends can be distinguished depending on whether the views are included in the database schema or not, i.e. managed in isolation. For the first trend, considering views as parts of the DB schema means inserting new virtual classes in the inheritance hierarchy of the application [11, 10, 17]. Although base classes are preserved, the schema itself no longer reflects the real world since it is enforced with additional classes, perhaps not meaningful for users in particular if they stand for "some external vision" of data. Furthermore, these virtual class insertions complicate the inheritance hierarchy [5]. The second trend [5] diverges significantly from the first one by considering two clear cut levels: on one hand the base classes describing the application schema, and on the other hand the virtual classes representing views. This clear distinction is particularly relevant when considering the cumbersome phenomenon of view proliferation which happens over time, all along the application life.

The proposed approach is rather close to the second described trend but it boosts the separation between views and database schema by introducing a middle level, called the "*Federated Schema*", which offers an overview of both base classes of the schema and virtual classes of views. This schema provides a relevant knowledge of both structural and behavioural aspects of applications. More precisely, schema modifications are not actually performed but are captured by defining views. Thus the database schema is never altered (likely until an entire reorganization becomes necessary). We believe that in such a framework, sharing and re-using views could be significantly improved. Furthermore, views are not only derived from a single class but they could result from any query involving several classes. When additional information (i.e. non derived) is captured through the creation of new properties in a virtual class, it is not actually stored in the database. In fact, the structural enrichment is reflected in the federated schema, while inserted values are stored in the federated base. The latter base is an additional storage for data added all along the application life. With this approach, we can insert new properties in any virtual class even if it is derived from several classes.

The paper is organized as follows. Section 2 proposes an overview of our approach and focusses on the multi-level schema architecture that we define. Schema definition is also addressed. Section 3 is devoted to the manipulation language provided to handle views. An implementation experiment is described in section 4. In section 5, we give an overview of related work. As a conclusion, we compare our proposal to other approaches.

2 The Federated Approach: An overview

2.1 Preambule

In our approach, designers can handle both base classes defined in the application schema and virtual classes initially derived from the former classes. Virtual classes can capture evolutions concerning a single class, such as deletion, modification or insertion of properties which can be either attributes or methods. But they can also represent creations of new relationships between classes, with their possible own properties ("relationship" is intented in the sense of the Entity/Relationship model). In such cases, the underlying virtual classes are derived from several base classes.
Thus users can handle the virtual classes through their virtual schemas. A virtual schema is a set of virtual classes which are related by IS_A links or reference links.

Thus, such a schema captures a set of modifications applying to different classes (including references between classes) but also IS_A links. Furthermore, each virtual class in a virtual schema is associated to at least another class belonging to the base schema or being virtual, through derivation links. Inheritance and derivation hierarchies are both constrained by construction rules which avoid, in particular, cycles when defining virtual schemas.

Virtual schemas defined by designers as well as the DB schema are integrated in the federated schema. This schema offers an overview of all the available structural components of an application. Thus it provides designers with a framework in which new virtual schemas can be created. When evolutions capture new information, i.e. new properties, it is actually stored in the federated schema. The associated values are managed in isolation from the original database. In fact additional data are stored in the federated database. The possibility of including additional properties in a virtual class allows the views to be augmented independently from the base classes. Figure 1 depicts the proposed approach for managing views.

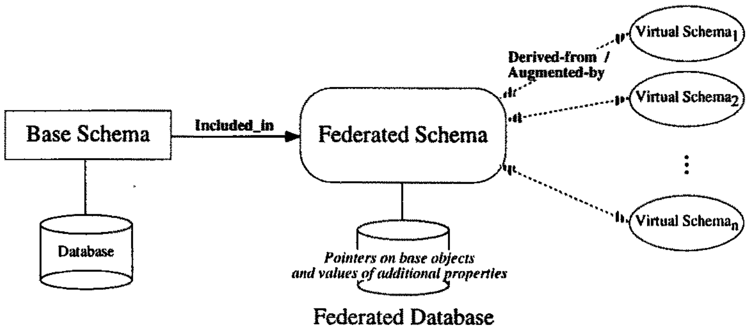

Fig. 1. The Federated Approach

2.2 Definitions

Since our approach fits in an OO context, we assume that the DB schema is an inheritance hierarchy of classes (above called base classes) with possible references between classes (expressed through attribute domains). For setting the groundwork of our manipulation language, we formally define the described concepts.
In this paper, we consider, like in [6], a typical object model: An object is defined by properties which may be either attributes or methods. Objects sharing

common properties, i.e. same attributes and behaviour, are grouped into classes. Classes are related by IS_A links through an inheritance hierarchy or by reference links (an attribute with a domain being another class). Multiple inheritance is allowed but the following condition must hold: two super-classes of a class have necessarily a common super-class. A base schema, or merely a schema, is defined as follows.

Definition 1 Schema
A database schema S is defined as a directed acyclic graph $S = (C, IS\_A)$ where C stands for the set of classes and IS_A for the IS_A links such that:

- *there is no IS_A cycle in the graph;*
- *two directed paths of IS_A links sharing the same origin have to be extended to a common super-class.*

A virtual class is derived from one or more *source* classes which may be either base or virtual classes. Nevertheless, the derivation process is constrained according to the following definition.

Definition 2 Derivation links
The derivation links must respect the following condition:
the virtual class includes an attribute referencing each one of its source classes.

A virtual schema includes several classes possibly related by IS_A links, and initially derived from a base schema.

Definition 3 Virtual Schema
A virtual schema S_v is defined by a triplet $S_v = (C_v, IS\_A_v, Deriv_v)$ such that:

- *(C_v, IS_A_v) satisfies Definition 1;*
- *$Deriv_v$ is a set of derivation links (according to Definition 2);*
- *there is no derivation cycle in S_v.*

The federated schema is then defined from a base schema and the various virtual schemas "derived" from it.

Definition 4 Federated Schema
The federated schema $S_F = (C_F, IS\_A_F, Deriv_F)$ of a base schema $S = (C, IS\_A)$ from which n virtual schemas $S_{v_i} = (C_{v_i}, IS\_A_{v_i}, Deriv_{v_i})$ are derived is defined by:

$$C_F = C \bigcup_{i=1}^{n} C_{v_i},\ IS\_A_F = IS\_A \bigcup_{i=1}^{n} IS\_A_{v_i},\ Deriv_F = \bigcup_{i=1}^{n} Deriv_{v_i}.$$

3 Creating and Manipulating Views

The following section describes, in an intuitive way[3], some of the most interesting operations provided for creating and manipulating views.

Example 1 *Our illustrating example, all along the paper, is managing university students. Let us consider the following base schema, graphically described in Figure 2, and from which we will derive our view examples. It is complemented by a textual*

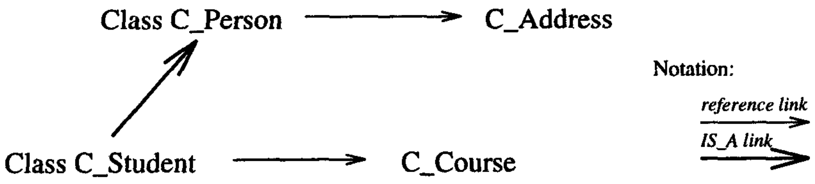

Fig. 2. The *University* Schema

description:

```
schema University;

class C_Person
type tuple (name: string,
 firstname: string,
 address: C_Address );
methods ···
end;

class C_Student inherit C_Person
type tuple (id: string,
 courses: set(C_Course));
methods ···
end;

class C_Address
type tuple (number: integer,
 street: string);
methods ···
end;

class C_Course
type tuple (title: string,
 discipline: string);
methods ···
end;
```

□

3.1 Schema Definition Language

This section is devoted to the schema definition language, provided for creating virtual schemas and classes. This language can be used by designers while end-users are provided with the manipulation language described in section 3.2.

[3] The evolution operations are fully described in [4].

Federated Schema Creation The federated schema results from the union of the base schema and its associated virtual schemas (Cf. Definition 4). Therefore, on an implementation level, the federated schema is necessarily initialized to the base schema. Then virtual schemas being derived are integrated in the federated schema. Creating a federated schema is performed through the following statement:

> `create federated schema` *FederatedSchemaName*
> `from` *BaseSchemaName*;

At this step, the federated schema *FederatedSchemaName* is merely the same than the base schema[4].

Example 2 *Let us consider the federated schema generated by:*

```
create federated schema UniversityFed from University;
```

where `UniversityFed` *is the federated schema name associated to the base schema* `University`. *The principle of such an initialization operation is illustrated in Figure 3. It has no effects on the base schema nor on the database where data are actually stored (*`University` *Base). Through the federated schema, designers can handle*

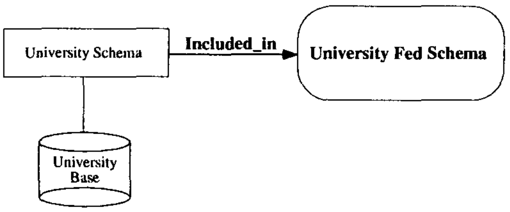

Fig. 3. The *UniversityFed* Federated Schema

the base classes and derive from them new virtual classes which enrich the federated schema: `UniversityFed` *Schema. When additional information is captured through virtual classes, it will be managed in isolation from the original database (as illustrated in Example 4)* □

Virtual Schema and Class Creation The federated schema provides the groundwork in which new virtual schemas can be defined, by using the following creation operation:

[4] In fact, on an implementation level, pointers are initialized to the base classes of the specified schema.

create virtual schema *VirtualSchemaName*
from *FederatedSchemaName*;

Example 3 *Within the* UniversityFed *Schema, we consider a virtual schema* Marks, *yielded by the operation.*

```
create virtual schema Marks from UniversityFed;
```

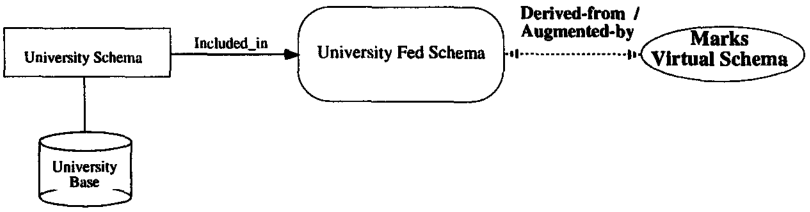

Fig. 4. The *Marks* Virtual Schema

In this virtual schema, initially empty, new virtual classes can then be created, as shown in Figure 4□

When defining new virtual classes which enrich the virtual schema, designers specify queries necessarily applying to classes in the federated schema. Thus virtual classes are provided with a structure or schema (attributes specified in the *Select* clause of the query and which reference other classes) and an extension (objects retrieved by the query). Virtual class creation is merely performed by:

create virtual class *ClassName* from *listofClassNames* as *Query*;

where *Query* is a retrieval order which could be expressed by using whatever DB manipulation language. For exemplifiying this principle, we choose a language similar to O_2SQL [16].

Example 4 *In the virtual schema* Marks, *we imagine that the designer wants to provide users with a virtual class* Student_Mark *capturing marks of each student for each taken course. For defining this virtual class, the first step necessarily specifies "what is derived", i.e. which attributes in which virtual or base classes are considered. This creation is performed as follows:*

```
create virtual class Student_Mark from C_Student, C_Course as
 (select s:C_Student, c:C_Course
 from s in C_Student, c in C_Course
 where c in s.courses);
```

Resulting from this operation, the virtual class Student_Mark *encompasses objects having their own identifier and defined as tuples of references which are stored in the database as illustrated in Figure 6. More precisely, each object captures a relation-*

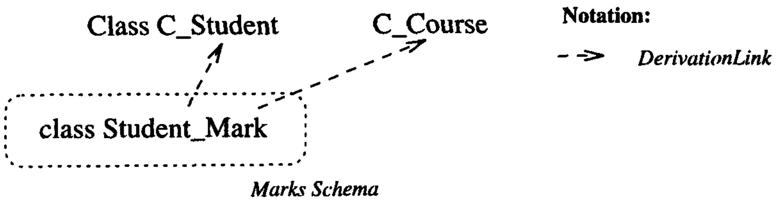

Fig. 5. The *Mark_Student* Virtual Class

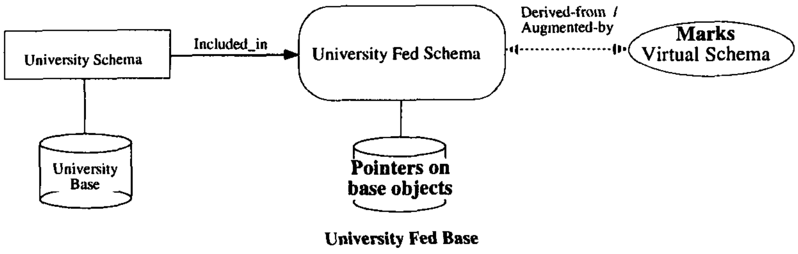

Fig. 6. The *Mark_Student* Objects in the Federated Database

ship between an existing student (for instance, object o_{s1}) and a course. For example, objects $o_{ms1} = (id_{ms1}, (*o_{s1}, *o_{c1}))$ and $o_{ms2} = (id_{ms2}, (*o_{s1}, *o_{c2}))$ capture the fact that student o_{s1} has taken the courses o_{c1} and o_{c2}. Thus objects in `Student_Mark` *are defined through their identifier and a couple of pointers referencing the classes* `C_Student` *and* `C_Course` *(Cf. Example 1). Figure 5 partially illustrates the federated schema* `UniversityFed` *as yielded by the operations described so far. For the sake of clearness, we do no fully represent the federated schema, i.e. the inheritance hierarchy and composition graph of base classes (Cf. Figure 2). We just consider the two base classes:* `C_Student` *and* `C_Course`*. The latter classes are related by derivation links to the virtual class* `Student_Mark` *which is the single class in the virtual schema* `Marks` □

Inserting additional attributes in a virtual class is the mechanism for supporting view augmentation. By this way, real world changes consisting of data enrichment are taken into consideration. The associated statement is the following:

`add attribute` *Attribute* : *AttributeDescription* `to` *ClassName*;

Example 5 *Let us resume the virtual class* `Student_Mark`*, derived in the previous example. Since no marks are stored in the original database, the virtual class must be complemented to fully capture the semantics intented by designers. Thus a new attribute marks is added by the following operation:*

```
add attribute marks: set(real) to Student_Mark;
```

Furthermore, a new method is required for getting values of the added attribute. This is done by:

```
add method getmarks (C_Student,C_Course) to Student_Mark;
```

Mark values of the virtual class are then stored in the federated database, as illustrated in Figure 7. When defining additional attributes, pointers on objects in the associated

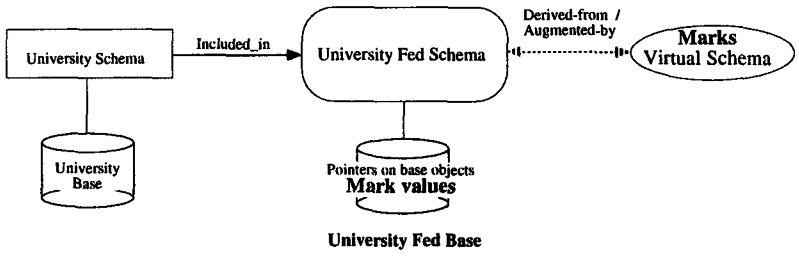

Fig. 7. Adding Values in the Federated Database

virtual class become persistent. In addition, persistency can apply to the class, merely by specifying it as a persistency root, in a similar way than in O_2 [16]:

```
name themarks: set(Student_Mark);
```

□

This approach requires an accurate management of update operations applying to the original database, because these operations must be propagated to virtual classes. When such propagations involve persistent virtual classes, it is necessary to evaluate once again the definition query of the considered class. We do not develop this aspect since the object update propagation is beyond the scope of this paper.

When attributes of referenced classes are not necessary in the virtual class, they can be hiden. Thus the designer can refine the external vision of data, offered by the new class, to fully meet user's expectations. In the associated statement, specified attributes can only be attributes of classes related by derivation links to the virtual class under examination:

`hide` *listofAttributeNames* `to Mark` *ClassName*;

Example 6 *When defining the virtual class* `Student_Mark`*, the designer aims to provide a mere view of student's results. In particular, attributes courses in* `C_Student` *and discipline in* `C_Course` *are not necessary and thus they are discarded from the user's view of* `Student_Mark` *by:*

```
hide (courses, discipline) to  Student_Mark;□
```

All the operations previously described can be grouped together within a single declaration of virtual class, including the query specification, attribute and method insertion, and attribute hiding.

```
create virtual class ClassName from listofClassName as Query;
add attribute  AttributeName : AttributeDescription;
add method  MethodName : MethodDescription;
hide  listofAttributeNames;
end;
```

Example 7 `Student_Mark` *could have been defined as follows:*

```
create virtual class Student_Mark from C_Student, C_Course as
 (select s:C_Student, c:C_Course
 from s in C_Student, c in C_Course
 where c in s.courses);
add attribute marks: set(real);
add method getmarks (C_Student,C_Course);
hide (courses, discipline);
end;
```

□

Additional Features In this section, we briefly present two additional features of our approach. The first one is the inter-view visibility. With such a capability, it is possible for a virtual schema to re-use classes defined in another virtual schema. Sharing classes is a relevant mechanism for avoiding redundant definitions of virtual classes but also for simplifying derivations.
An existing class in a virtual schema can be visible, and then used, in another schema through the following export order:

```
 export virtual class ClassName;
```

In our approach, a second feature to be underlined is the creation of virtual attributes. Virtual attributes are just attributes of a virtual class yielded by its definition query (and not user-defined, Cf. Section 3.2). A virtual class can be complemented by inserting new virtual attributes. Let us notice that virtual attributes can merely reflect base attributes. But they can also capture additional semantics hiden in the original database, by performing calculus. In such cases, the attribute must be given a name by user.

Example 8 *We imagine that a virtual schema* `Averages`*, illustrated in Figure 8, is defined, including a single class* `Course_Average`*. This class is derived from the virtual class* `Student_Mark` *(which must be previously exported) through a query performing an object grouping. The performed orders are the following:*

```
export virtual class Student_Mark;
create virtual schema Averages from UniversityFed;

create virtual class Course_Average from Student_Mark as
 (group sm in Student_Mark
 by (t: sm.course.title));
```

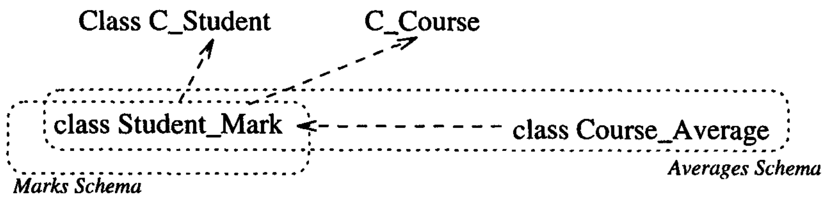

Fig. 8. Two Virtual Schemas

The class `Course_Average` *is complemented by adding a new attribute* `average` *reflecting the average marks of students for courses. This attribute is virtual since its values can be computed from* `Student_Mark` *through a mere query, as shown in the following order:*

```
add virtual attribute average as
(select avg(sm.partition.marks) from sm in Course_Average);
hide (address,name,firstname,libelle,id);
```

where partition is obtained from the previous query □

3.2 Schema Manipulation Language

For updating their virtual schemas, users are provided with evolution capabilities, enhanced through the proposed manipulation language. In fact, each DB programmer or user only accesses his own schema and update operations must not alter schemas of other users.
In an OO context, the evolution issue has been widely addressed [2, 3, 8, 9, 18]. In particular, a set of primitives proved to be minimal and complete is proposed by R. Zicari in [19]. By combining these primitives, any more complex evolution can be expressed. In our approach, we resume these basic operations and adapt them to our view context. These operations are insertion or deletion of property, class and IS_A link.
In the following sub-sections, we present, in an informal and illustrated way, the different operations provided for updating virtual schemas, along with their possible effects on other virtual schemas and the federated schema.

Adding Property Adding a property (attribute or method) in a virtual class cannot be merely seen as the modification of some class in the federated schema because such an operation can have consequences for other existing virtual schemas. Critical side-effects can occur when the updated class is exported or derived for a re-use concern. Thus it is necessary to restrict the application scope of property insertions to the single virtual schema in which the concerned virtual class is defined. With this constraint, virtual schemas using the complemented class are garanteed to be consistent. For enhancing such a constraint, our approach redefines the virtual class to be modified. In fact, the property insertion is not actually performed in the original class but in a simulated copy (by a virtual class) of it. This copy is created and managed by the system, and it is not visible for the user. From the designer's viewpoint, he is provided with an alias of the virtual class, denoted by the key word *is-used-as*.

Example 9 *Let us imagine that, for meeting an application or user need, the final marks of students for the taken courses must be captured. An attribute insertion in the virtual class* `Student_Mark` *can be performed in the following way:*

```
    add attribute finalmark: real to Student_Mark;
```

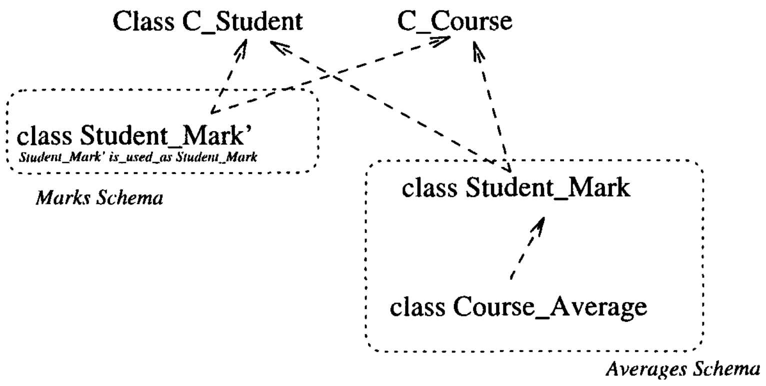

Fig. 9. The Final Virtual Schemas after an Insertion

However, since `Student_Mark` *is re-used by the virtual class* `Mark_Average`*, the insertion does not actually alter* `Student_Mark`*, but a virtual copy of it:* `Student_Mark'`. *Figure 9 proposes the low-level representation of our federated schema example, but for designers the federated schema remains the very same (apart from the provided alias)* □

IS_A Link and Class Creation New classes can be created by specifying their inheritance relationship with one or several existing classes, by using the following order:

virtual class *InheritedClassName* inherit *ClassName*;

In addition, a class can be declared as being the super-class of another one:

virtual class *SuperClassName* SuperClass of *ClassName*
[hide *listofAttributes*];

Example 10 *In Figure 10, the virtual class* `Honours_Student`*, in the virtual schema* `Marks`*, is defined as a sub-class of* `Student_Mark` *merely by specifying an inheritance link between them:*

```
    virtual class Honours_Student inherit Student_Mark;
```

□

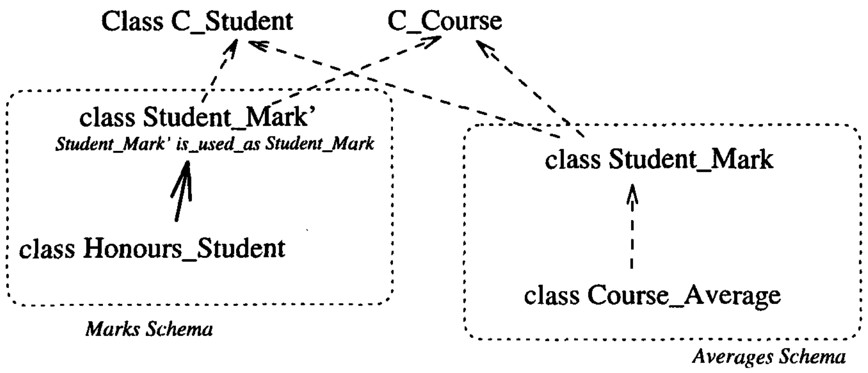

Fig. 10. The *Honours_Student* Virtual Class

Deleting a Property Deleting property from a virtual class must preserve the consistency of all the virtual schemas within the federated schema. Like for property insertion, if the virtual class is re-used, the operation is performed on a copy of the concerned class by hiding the property to be deleted. It is propagated to possible sub-classes within the considered virtual schema, once again by using the hide clause.

Example 11 *If the user wants to discard* `Address` *from the virtual class* `Student_Mark`*, he can use:*

```
hide (address) to Student_Mark;
```

As a result, the attribute is no longer visible for the user, in particular in the sub-class `Honours_Student`*, but the operation applies on the copy* `Student_Mark'`*. Thus the single virtual schema to be modified is* `Marks` □

Deleting a Class In an inheritance hierarchy, deleting a class is usually performed by removing it while relating its possible sub-classes to its super-class(es) [19]. However, in our context, class deletion must be examined by paying particular attention to derivation links. In fact, exactly like for property insertion or deletion, the operation effects must be located within the single virtual schema from which a class must be removed. In our approach, class deletions are guaranteed to be without side-effets, once again by using the copy mechanism. Now, let us consider the modified virtual schema. If the user intends to remove a virtual class having sub-classes, the deletion is necessarily propagated to the sub-classes. Actually, sub-classes cannot be preserved, since a main part of their semantics is captured through the definition query of their super-class.

Example 12 *If the user deletes the virtual class* `Student_Mark` *in the virtual schema* `Marks`*, its sub-class* `Honours_Student` *is automatically removed (Figure 11). In fact deletion applies to the virtual copy* `Student_Mark'`*, thus the virtual schema* `Averages` *is preserved.*

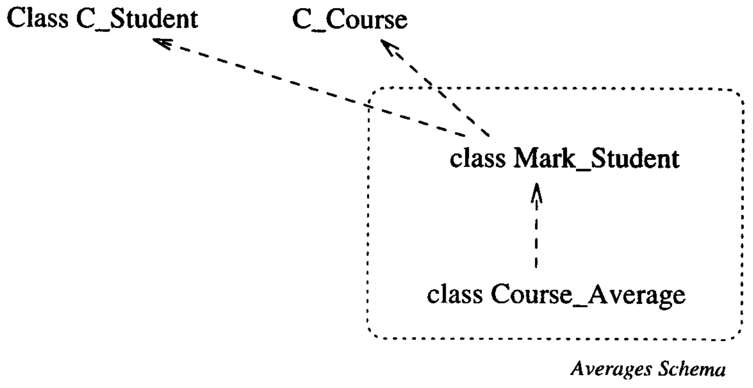

Fig. 11. Federated Schema after *Student_Mark* Deletion

4 An Implementation using the O₂ DBMS

4.1 System Architecture

Based on the approach described in this paper, a prototype [7] is currently developped on top of the O_2 DBMS: *SETV* (Schema Evolution Through Views). It provides both the view definition and manipulation languages. Figure 12 gives an overview of *SETV* architecture. Within the system, the component Virtual Object Manager maintains relationships between virtual objects and the associated base objects.
When a virtual class is derived from several classes, its objects must be provided with their own OID, created and managed by the system all along the worksession. For additional attributes added in virtual classes, values are get through an "init" method which includes a "new" operator.
There are two pure ways to perform query processing against views: materialization and query transformation. The first one is to materialize all derived classes referenced by the query. The other consists of transforming a query into an equivalent set of subqueries refering to base classes.
In our implementation, we adopt a dynamic materialization for composite virtual object. However, the materialized virtual objects must be maintained for reflecting the update operations on the database. In our system, this consistency is verified through triggers related to update actions in the same way as in [3].

4.2 System Classes

The components of *SETV* are implemented as system classes. A part of the *SETV* Meta-Schema hierarchy is presented in Figure 13. We just give an example including system classes representing a virtual class with the O_2 data definition language:

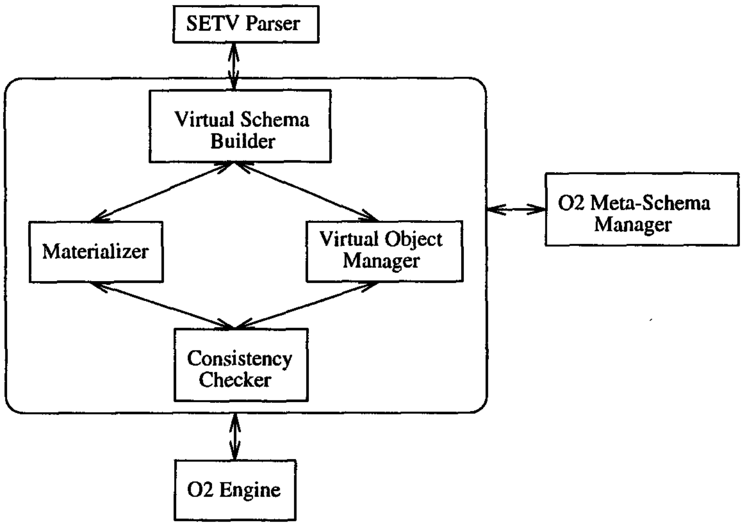

Fig. 12. The *SETV* Architecture

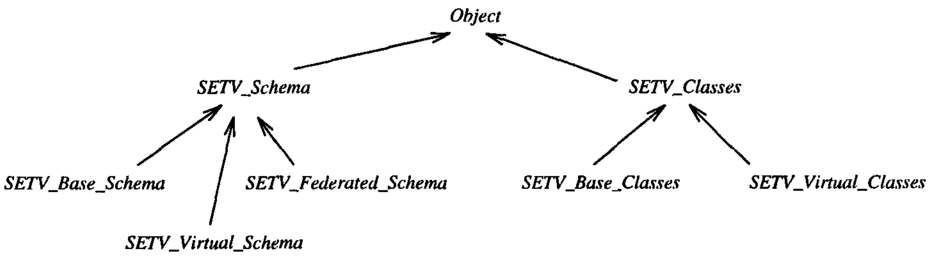

Fig. 13. The System Classes

```
class SETV_virtual_classes inherit SETV_classes
public type tuple(
range: string, /* class visibility */
from: set(SETV_base_classes), /* base class names */
extension: SETV_extensions, /* virtual class extension */
attributes: set(SETV_attr), /* virtual class attributes */
methods: set(SETV_methods), /* virtual class methods */
has_additional: integer, /* flag for additonal property */
has_init: integer) /* flag for init method */
end;

class SETV_extensions
```

```
public type tuple(
name: string, /* extension name */
class: SETV_virtual_classes, /* related virtual class name */
is_materialized: integer, /* flag for materialization */
query: string, /* for populating virtual class */
is_persistent: integer, /* flag for peristent extension */
end;
```

5 Related Work

The concept of schema virtualization is introduced by K. Tanaka et al. [15]. A more sophisticated view language is proposed in [1] for the O2 object model. It includes a set of view primitives (creation of virtual classes, imaginary classes, virtual attributes, OID creation for imaginary objects). The concepts described in [1] are extended and implemented in the prototype O2Views on top of the OODBMS O2 [13]. However, capacity augmenting views are not allowed. Therefore, O2Views cannot be used for simulating schema evolutions.
By using the view mechanism, the approach proposed in [5] enhances various DB features such as: dynamic sets, partition of classes, schema evolution and versions. The author shows how most of the schema changes can be simulated by view creation. In [17], the simulation process is enhanced and included in an external schema definition. An external schema is defined as a subset of base classes and a set of views corresponding to the schema modifications. Besides, this external schema has to be closed. More recently, the Multi-view system [12] is enhanced for supporting schema evolutions [11]. However, the issue concerning capacity-augmenting views derived from several base classes is not addressed.

A very important question to be answered concerning the simulation of schema evolution is: can all schema modifications be simulated? An answer can be found in [17] where schema modifications are classified according to their impact on the object modelling capacity. In fact, three kinds of transformations are considered depending on whether they preserve, reduce or augment capacity. All theses transformations could be simulated with views. But, when considering views as integrated in the base schema, capacity augmenting operations necessarily require a data reorganization [17, 11]. Let us notice that integrating views in the base schema reveals another defect: fully artificial classes can be necessary when a view is derived from several classes [17]. These artificial classes can be cumbersome for users since they only result from intermediary steps when computing views. By using the proposed federated approach, such problems are avoided.

6 Conclusion

In this paper, we propose an approach for managing structural evolutions through views. Evolutions are not actually performed but simulated, thus the DB schema

is preserved for both users and existing applications. Furthermore, the approach avoids costly database reorganizations. This important advantage results from the multi-schema architecture which is introduced and from the isolated management of additional data (within the federated base).
In our approach, the concept of virtual class (widely used in related work) is complemented through the notion of virtual schema, encompassing several classes and their possible inheritance or derivation relationships. Furthermore, through the concept of federated schema, the inter-view visibility is enhanced with the main advantage of improving view re-use. With the proposed clear cut separation between the federated and base schemas, none operation needs an actual database reorganization, even if it is capacity augmenting. By this way, our approach boosts the simulation capabilities when compared to [11]. Furthermore, artificial classes are not necessary for defining views.
View definition and manipulation languages are defined for designers and users. They provide the required functionalities for handling views. Furthermore when manipulating classes, modification operations are guaranteed without side-effects for other users. With this approach, each user is provided with a great autonomy for updating his virtual classes while taking advantage of other user's manipulations through the federated schema.

References

1. S. Abiteboul and A. Bonner. Objects and Views. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD*, pages 238–247, Denver,USA, May 1991.
2. J. Banerjee, W. Kim, K.J. Kim, and H. Korth. Semantics and Implementation of Schemes Evolution in Object-Oriented Databases. In *Proceedings of the International Conference on Management of Data (ACM SIGMOD)*, San Francisco, USA, May 1987.
3. Z. Bellahsene. An Active Meta-Model for Knowledge Evolution in an OODBMS. In *Proceedings of the 5th International Conference on Advanced Information Systems (CAiSE)*, Lecture Notes in Computer Science, Paris, France, June 1993.
4. Z. Bellahsene. View Mechanism for Schema Evolution. Technical Report 95035, LIRMM CNRS/ University of Montpellier II, France, May 1995.
5. E. Bertino. A View mechanism fo Object-Oriented Database. In *Proceedings of the 3rd International Conference on Extending Database Technology (EDBT)*, pages 136–151, Vienne, Austria, March 1992.
6. E. Bertino and L. Martino. Object-Oriented Database Management Systems: Concepts and Issues. *Computer*, 24(4):37–47, April 1991.
7. G. Lucato. Evolution de schéma au travers des vues. Technical report, LIRMM CNRS/ University of Montpellier II, France, June 1995.
8. D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. October 1987.
9. P. Poncelet and L. Lakhal. Consistent Structural Updates for Object-Oriented Design. In *Proceedings of the 5th International Conference on Advanced Information Systems (CAiSE)*, Lecture Notes in Computer Science, Paris, France, June 1993.

10. Y-G. Ra, H. Kuno, and E.A. Rundensteiner. A Flexible Object-Oriented Database Model and Implementation for Capacity Augmenting Views. Technical Report CSE-TR-215-94, Electrical Engineering, Computer Science and Engineering Division, University of Michigan, Ann Arbor, May 1994.
11. Y-G. Ra and E.A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In *Proceedings of the IEEE Conference on Data Engineering(ICDE)*, Tapei, Taiwan, 1995.
12. E.A. Rundensteiner. Multiview: a Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Vancouver, Canada, 1992.
13. C.S. Dos Santos. Design and Implementation of an Object-Oriented View Mechanism. Technical Report GOODSTEP N-7, INRIA, Rocquencourt, March 1994.
14. C.S. Dos Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *Proceedings of the International Conference on Extending Data Base Technology (EDBT)*, Lecture Notes in Computer Science, Cambridge, UK, March 1994.
15. K. Tanaka, M. Yoshikawa, and I. Koso. Schema Virtualization in Object-Oriented Databases. In *Proceedings of the IEEE Conference on Data Engineering(ICDE)*, February 1988.
16. O2 Technology. The o_2 User's Manual Version 3.3. Technical report, March 1992.
17. M. Tresch and M.H. Scholl. Schema Transformation without Database Reorganization. *ACM SIGMOD Record*, 22(1):21–27, March 1993.
18. R. Zicari. A Framework for O_2 Schema Updates. In *Proceedings of the 7th IEEE International Conference on Data Engineering(ICDE)*, pages 146–182, April 1991.
19. R. Zicari. Primitives for Schema Updates in an Object-Oriented Database System: A Proposal. *Computer Standards & Interfaces*, pages 271–28, 1991.