Virtual Schemas and Bases*

Cássio Souza dos Santos† (Dos-santos.Souza@inria.fr)
Serge Abiteboul (Serge.Abiteboul@inria.fr)
Claude Delobel (Claude.Delobel@inria.fr)

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

Abstract

We propose the notions of *virtual schemas* and *virtual bases* as a coherent way of integrating various features in OODB views. A virtual schema is defined based on some existing (real) schema. A virtual base is obtained when a (real) base is attached to a virtual schema. We study the consequences of this simple assumption. In particular, we observe the differences between a real schema and a virtual one. We also consider an extension (that we call generic schemas) where it is necessary to specify *several* real bases to attach data to a virtual schema. We show how the flexibility provided by virtual schemas can be used to cope with various dynamic features of database systems.

1 Introduction

Views are intended to increase the flexibility of database systems and their definition in the object-oriented database (OODB) context comes as a natural extension of the original paradigm. The yet relatively young research on this topic has introduced a large variety of indispensable new features. However, the organizational aspects have been overlooked and a coherent way of integrating theses features is still missing. To solve this problem, we propose the notion of *virtual schema*. This concept is far from revolutionary: together with virtual bases, such schemas are intended to model the external level of the database exactly in the spirit of the architecture proposed by the Ansi/X3/Sparc Study Group [oDMS75]. We show how the flexibility they provide can be used to cope with various dynamic features of database systems.

It has been pointed out that enforcing users to follow fixed built-in links within the data is a drawback of OODB which seriously limits the advantages of object-orientation. In particular, objects may evolve and play different roles (even simultaneously) over time.

*Partially supported by Esprit Project GoodStep and PRC BD3.

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

Multiple inheritance has been shown to be unsuited to model multiple facets of objects dynamically and in a flexible manner [Bor88, RS91, TYH+91, MZ92]. Thus, various extensions to the paradigm have been considered, some of which explore the concept of class virtualization [TYI88, AB91, SLT91], the associated idea of defining multiple interfaces to an object [SS89, TYH+91, RS91], the definition of non-strict inheritance mechanisms [Bor88, Bor89], or alternatives to class inheritance for sharing and extending properties [Lie86, TYH+91]. That amounts to a revision of the object-oriented paradigm which is driven by database and software engineering as well as knowledge representation (AI) concerns. In this spirit, many proposals have been recently made for the introduction of a view mechanism in object-oriented databases [AB91, Ber91, HZ90, RB92, SLT91, SS89, TYI88].

An important aspect of object-oriented views is that unlike relational views [Ker86, Tem86], they are not only intended to restructure data. Operations on data play an essential role. Therefore, for instance, combining classes is far more complicated than combining relations. It leads in particular to the reorganization of aggregation (composition) and generalization (subclass) hierarchies, a currently active research topic. A view mechanism in an OODB context is thus more intricate than its analogue in a relational system. This is the price to pay to obtain more flexibility.

The main contribution of this paper is to demonstrate that a view mechanism can be integrated as a basic functionality for OODBs *at the schema level.* Starting from a (real) schema, a virtual schema is defined. A virtual base is obtained when a (real) base is attached to a virtual schema. We study the consequences of this simple assumption. In particular, we observe the differences between a real schema and a virtual one. We also consider an extension (that we call generic schemas) where it is necessary to specify *several* real bases to attach data to a virtual schema.

The examples that we show demonstrate that this mechanism is naturally suited to model the object multi-faceted nature and data integration. The multiple views are then interpreted as *differences* in points of views. The mechanism can also be used to model *changes* in points of views for a better modeling of dynamic aspects. From this observation, we argue in the last part of the paper that virtual schemas can also be useful in a variety of contexts where dynamics is crucial such as schema evolution, data versioning or integrity constraints.

This paper is a continuation of the work presented in [AB91]. We further extend that proposal with additional capabilities. More important, we propose virtual schemas as the necessary framework which was missing there as well as in other proposals. Our presentation is mostly informal and leans on a number of examples. We use (and extend) the language O_2SQL [Clu91] of the O_2 system [BDK92]. However, the concepts that we discuss clearly apply to other OODB systems as well.

We are currently implementing a prototype view mechanism for the O_2 system. This development as well as the research described in this paper are performed in the framework of the Esprit Project GoodStep. Implementation aspects are briefly discussed but a detailed presentation of the implementation is not in the scope of the present paper.

2 Virtual Schemas

In our approach, the definition of a view is similar to the definition of a schema. A *virtual schema* is, as a normal schema, an organizational unit meant to encapsulate a set of related intentional definitions. The main difference is that a real schema refers to real data described in the database whereas a virtual schema refers to a virtual world.

A view is thus in our context a special kind of schema to which some constraints on its use and definitions are imposed. We will use the term *virtual schema* as a synonym of view in what follows and the term *real schema* will be used in contrast to virtual schema to avoid confusion.

Like a real schema, a view includes definitions of classes, methods, types, functions and named objects. It may also import and export definitions from other schemas, although these mechanisms will be given a slightly different semantics. In addition, virtual definitions can be included in a virtual schema.

We next illustrate the notion of virtual schema through an example.

Let us define a real (root) schema from which we will derive our first view. The schema is partially specified in Figure 1, where the O_2 data definition language [Deu89] is used. Only relevant definitions are shown. Two classes, namely `Date` and `Person` are imported from other schemas. Two other classes, `C_Employee` and `Department` are defined in the schema along with two named sets which are roots of persistence for instances of the defined classes.

```
schema company_schema;

import schema o2kit class Date;
import schema general_schema class Person;

class C_Employee inherits Person
type tuple(salary:real,
           department:Department,
           date_of_admission:Date);
methods ...
end;

class Department
type tuple(name:string,
           boss:C_Employee);
methods ...
end;

name The_C_Employees: set(C_Employee);
name The_Departments: set(Department);
```

Figure 1: Real schema `company_schema`

Figure 2 shows a virtual schema derived from schema `company_schema`. This example contains the main components of a view definition, which are explained in the sequel.

A virtual schema is always derived from another virtual or real schema, which will be called its *root schema*. When we define a view we must thus declare its root schema. The root schema of a view determines on which bases it can be activated, namely those bases instantiated from its root schema. The root schema of view `company_view` is `company_schema`, as declared in the virtual schema definition command through its `from` clause.

```
virtual schema company_view from company_schema;

import all;

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

attribute number_of_employees:integer in class Department has value
   count(select e from e in The_C_Employees
                  where e.department = self);

attribute subordinates:set(C_Employee) in class Boss has value
   ((select e from e in The_C_Employees
              where e.department = self->department) - set(self));

attribute employees:set(C_Employee) in class Department has value
   self.boss.subordinates;

attribute number_of_subordinates:integer in class Boss has value
   count(self.subordinates);

attribute boss:Boss in class C_Employee has value self.department.boss;

hide attribute salary,boss in class Boss;
```

Figure 2: View `company_view` defined from schema `company_schema`

In a view, we can import definitions from other schemas and views as well. The command `import all` specifies that all definitions from the root schema are to be imported; that includes those definitions already imported to the root schema itself, as it is the case for class `Date` imported from schema `o2kit` to `company_schema`.

The command `import all` is a shorthand for the conventional importing command where the names of all imported definitions are explicitly listed. Its definition is motivated by the fact that when defining a view one will often want to import all definitions from its root schema and then add some additional virtual properties.

In addition, not only those definitions flagged as exportable (using the `export` command) in the root schema are imported, but actually all definitions in the root schema, regardless of their visibility. This is the only case where private definitions in a schema can be imported by another schema[1].

Classes imported to a real schema cannot be redefined but this restriction is weakened in virtual schemas so as to allow the redefinition of classes imported from the root schema only.

[1] The reader acquainted with C++ will notice a similarity with the friend class concept, which adapted to our context allows us to see a virtual schema as a friend of its root schema and to which some visibility privileges apply.

After creating the view through the `virtual schema` command and importing definitions from other schema(s), we have the appropriate framework for defining virtual data. Here, we follow the ideas of [AB91]. The core specification of the view therefore consists in the definitions of *virtual classes* and *virtual attributes* and in the *hiding* of existing definitions.

We briefly describe virtual attributes and the hiding of definitions. We will next discuss in more detail the concept of virtual classes since it raises interesting new issues.

Virtual Attributes

A virtual attribute is a special kind of function whose returned value is viewed as part of the structure of a virtual object. They can be defined in both real and virtual classes. In addition, they can be defined from other virtual attributes.

Class `Department`, for example, is extended with the addition of a virtual attribute to it, namely `number_of_employees`. The virtual attribute `employees` in class `Department`, in turn, makes use of virtual attribute `subordinates` defined in virtual class `Boss`.

Hiding Definitions

Let us assume that `company_view` is a view intended to be publicly available and that salaries of bosses are considered confidential. For this reason, attribute `salary` in class `Boss` is hidden.

Confidentiality is provided by the hiding mechanism. Not only attributes, but also methods and classes can be hidden. It is important to notice that hiding an attribute has a different effect from declaring it as private. In O_2, encapsulation provides modularity and enhances code maintainability, but it can be broken in the interactive (*ad hoc*) query mode, thus it cannot be used to implement authorization constraints, as discussed in [Clu91].

Authorization must be provided at the view level through the combined use of the hiding and importing mechanisms as well as the definition of appropriate access rights to views. O_2 allows the definition of user groups to which read and write access rights can be granted with respect to schemas and bases. Such a mechanism is naturally extended to cope with virtual schemas and bases.

Observe that hiding an attribute in a class does not imply the creation of a virtual superclass (although the new type has fewer attributes and can be considered as a supertype of the original one). In that sense, we are lead to consider a non-strict semantics for the inheritance mechanism in the style of that defined in [Bor88] where contradictions and exceptions are allowed to coexist with normal cases in the class hierarchy.

Virtual Classes

Virtual classes can be defined from other classes through two virtualization processes, namely specialization and generalization. The present example considers only class virtualization through specialization. Generalization will be considered in a following example.

A virtual class can be defined through specialization from another virtual or real class, which is called its *base class*. The definition of a specialized virtual class declares its extension, i.e. the set of objects belonging to it, through a query expression[2]. Figure 3 illustrates

[2]The query language of O_2 is called O_2SQL and is, as its name suggests, an SQL based declarative query language.

the general format for the definition of a virtual class through specialization. The class `Boss` above, is a specialization of class `C_Employee`. Every instance of `Boss` is actually an instance of `C_Employee`.

```
virtual class class_name includes query_expression
```

Figure 3: Definition of a Virtual Class through Specialization

Many different virtual classes can be defined from a given class through specialization by using different queries stating different (and not necessarily mutual excluding) properties. This approach has been considered by [AB91, Ber91, SLT91] among others and it leads to some basic issues which can be summarized as follows:

- Where should the virtual class be placed in the global class hierarchy?
- When should we compute the extension of a virtual class?
- How and when should an object be considered an instance of a virtual class?
- If an object is an instance of different virtual classes defined in the same view[3], how to solve the ambiguity?

We next consider a problem that has been overlooked so far and which leads to a distinction between real and virtual bases. We have to distinguish between two kinds of data models: those with class extension and those without them. It should be noted that most systems generally support class extensions at least at the physical level (e.g., this is the case for O_2 although the class extensions are not available at the logical level). We will not argue for or against class extensions. However, the implications are important for the specifications of virtual classes. To see them we consider three alternative definitions for `Boss`:

1. If the class extensions are available, we can use them to specify the population of `Boss` as in Figure 4. There are differences between this definition and the one given in Figure 2. An important one is that, in the original definition, the system must infer that the class `Boss` is a subclass of `C_Employee`, whereas it is already specified in this new one. Indeed, for the definition given in Figure 4, one can interpret the `where` clause as the specification of the characteristic function of class `Boss` in class `C_Employee`.
2. If the model does not provide class extensions, an analogous specification can be obtained by allowing specifications based on characteristic functions. The general structure and an example are given in Figure 5.
3. Finally, one should note that in many cases, it will be more practical to define the characteristic function as a method (e.g., Figure 6) in particular to have access to private attributes of the object or to be able to refine it in subclasses.

[3] An object can be an instance of many different virtual classes but it is always an instance of one single real class. This is the so called unique root rule, intended to increase the efficiency and decrease the complexity of data-intensive systems.

```
virtual class Boss includes
      (select e from e in C_Employee
                where exists d in The_Departments:(d.boss = e))
```

Figure 4: Definition for class `Boss` using class extension

The previous discussion suggests that a specification of the form `includes` *query_expression* is appropriate for models with class extensions, whereas the use of characteristic functions is more appropriate if class extensions are absent from the model. Observe that in a context without class extension we need not have the extension of the view. Also, in all cases, we must be able to test membership in a view (i.e. be given or derive the characteristic function) since this test is going to be at the core of the method resolution mechanism and will be thus an essential component of the performance of the view system.

```
virtual class class_name includes var in class_name where condition

virtual class Boss includes e in C_Employee
                where exists d in The_Departments:(d.boss = e)
```

Figure 5: Alternative Definition for class `Boss`

```
method ℵ_Boss:boolean in class Employee
method body ℵ_Boss:boolean {
   return(exists d in The_Departments:(d.boss = self));}
virtual class Boss includes Employee where ℵ_Boss
```

Figure 6: Characteristic Function for Virtual Class `Boss`

To conclude this section, we briefly consider an important aspect that we will not treat in too much detail, namely resolution conflicts.

Interface Resolution Conflicts

When multiple virtual classes are defined from the same class, resolution conflicts may arise since an object may become an instance of more than one virtual class. Such conflicts resemble conflicts due to multiple inheritance. However, conflicts for multiple inheritance can be detected at compile time whereas objects may become and cease to be an instance of a virtual class at run time as a consequence of an update, so the situation is more intricate for virtual classes.

Forbidding the overlap of virtual classes has been considered in some approaches [TYI88] but is clearly too restrictive [RS91]. A possible solution would be to forbid the definition of more than one virtual classes from the same class in the same view, i.e. have at most one facet of an object active at a time. Unfortunately, this approach may lead to an explosion in the number of necessary auxiliary views.

There are (like for multiple inheritance) many alternatives based on type casting[4], user intervention and use of priority levels. Observe that priorities can be implicitly assigned through nested view activations. If, for instance, the employee facet of a person must prevail on his or her student facet then the view where the employee facet is defined can be derived from that where the student one is given.

3 Virtual Bases

In this section, we show how to activate the views and consider the differences between a real base and a virtual one.

3.1 View Activation

Let us assume the existence of a base named `company_base` which has been instantiated from schema `company_schema`. The view `company_view` can be activated on such a base as illustrated in Figure 7.

```
set base company_base; /* base activation */
set schema company_view; /* view activation */
```

Figure 7: Simple view activation

This is a valid view activation since the root schema of `company_view` is the schema from which the current base `company_base` has been instantiated. The activation of a virtual schema as illustrated in Figure 7 causes a *virtual base* to be automatically created. This virtual base becomes the current base and its root schema, i.e. `company_view`, the current schema.

A virtual base can also be given a name after it has been activated, as illustrated in Figure 8. The names of virtual bases are persistent and can be used as a shorthand for further view activations on the same base, as illustrated in Figure 9. Another more elaborate use of named virtual bases will be illustrated in Section 4, where they are passed as parameters to generic view activations.

```
set base company_base;
set schema company_view;
virtual base company_view_base; /* assignment of a persistent
                                   name to the virtual base */
```

Figure 8: Creation of a Named Virtual Base

Figure 10 illustrates the activation of `company_view` on base `company_base`. Triangles denote schemas whereas cylinders denote bases. Dotted triangles and cylinders denote virtual schemas and virtual bases respectively.

[4]Casting in O_2C, the O_2 programming language, is similar to that defined for C++.

```
set base company_view_base;
```

Figure 9: Activation of a Named Virtual Base

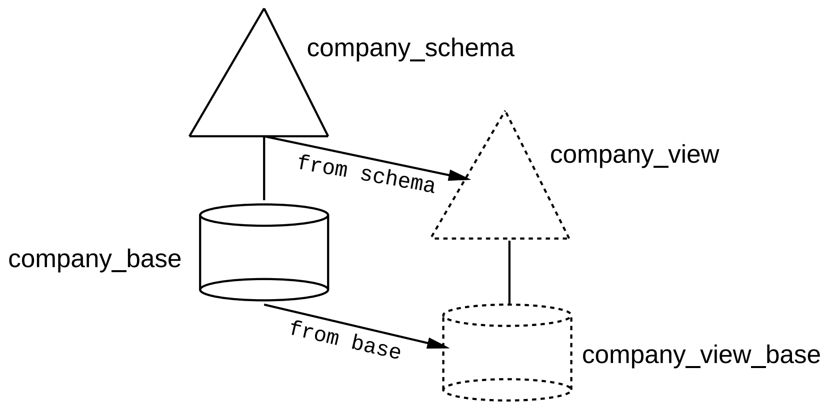

Figure 10: Relationship between Real and Virtual Entities

Conceptually, a virtual base is created every time a view is activated. From an implementation point of view, however, this does not imply the physical creation of such a base. The concept of a virtual base is implemented through the activation of a more sophisticated interface resolution mechanism in charge of testing whether instances of real classes are also instances of a virtual class (using system derived characteristic functions) and performing the appropriate bindings as well as constructing the temporary virtual structures. Virtual data are therefore dynamically derived on demand[5].

It must be noticed that the view approach proposed here is different from that defined in relational systems. In such systems, view names (which denote virtual relations) are used in the query language as a shorthand for user convenience. When a query using a view name is computed, the view is dynamically evaluated and query optimization plays an important role in assuring a good performance. In our approach, view names are used for switching the context for object structure and behavior evaluation. A good performance depends on efficient evaluation of the status that an object must acquire relatively to the currently active view at execution time.

The system must keep track of dependencies between virtual attributes and the attributes they are derived from so as to be able to automatically update virtual attributes for those objects whose representations are mapped on the screen. In addition, since virtual class membership and consequent changes in structure and behavior for a given object may depend on its value, data updates must be controlled so that object interfaces be always consistent.

When `company_view` is active, i.e. when it is the current schema, every instance of class `C_Employee` which is also a virtual instance of class `Boss` gains a new virtual structure with two additional attributes, namely `subordinates` and `number_of_subordinates`. In addition, every instance of `C_Employee` also has an additional attribute, namely `boss`, but for instances of `C_Employee` which are also virtual instances of `Boss` this attribute is hidden.

[5]Active rules such as those defined in [Sto92] can be used to implement the mechanism at a lower level.

Instances of `Department` were also given a new attribute, namely `employees`, defined from another virtual attribute (`employees` defined in `Department_Boss`).

The new structure defined for a virtual object must not be confused with the creation of a new object. The object identifier of a virtual object is the same as that of its corresponding real object. Actually they are the same object, but have multiple interfaces. Related approaches are described in [SS89, RS91], but they differ in that multiple copies of an instance variable are allowed to occur within an object [SS89] or that multiple explicitly constructed structures are stored with a single associated object identifier [RS91], whereas in our approach, only one of the multiple structures corresponds to the real (and possibly stored) object while the others add derived data which are dynamically and automatically constructed.

The construction of new objects, which result typically from a join of two or more objects through an O_2SQL expression and to which new temporary object identifiers are attached, can be achieved in our model through the definition of *imaginary classes*, whose instances are called *imaginary objects*. We will not pursue this issue in this paper but the reader can refer to [AB91] for more detail.

3.2 Nesting of Views

There is no limitation to the number of views that can be defined from a schema. In addition, views can be defined from other virtual schemas, i.e. the root schema of a view can be virtual itself. This leads naturally to the notion of a valid sequence of nested view activations.

In the previous example, let us assume that two views, namely `V1` and `V2` were defined from `company_view`, whereas another view called `V3` was defined from `company_schema`. The associated schema dependency tree is illustrated in Figure 11. If `company_schema` is active, only views `company_view` and `V3` can be activated. If `company_view` is activated, however, then `V1` and `V2` can be activated, but once one of them is activated, the other one will no longer be able to be activated until `company_view` becomes the active view again. If, for instance, `V1` is the active view, then deactivating it will cause `company_view` to be reactivated. Views are deactivated through the `drop view` command.

The activation of a view may overwrite definitions in the previously active view, if any, but this view is not deactivated, it only becomes temporarily inactive. A view activation stack is thus maintained and when a view is activated it is pushed onto the stack. The currently active view is always on the top of the stack. The bottom of the stack is always a real schema, the root of the schema dependency tree associated to the sequence of view activations recorded in the stack.

Another subtle way of nesting view activations is by passing virtual bases as parameters to a *generic view*. This will be considered in Section 4.

3.3 Real vs. Virtual Schemas

As stated in the beginning of this paper, a view is a special kind of schema to which some constraints on its use and definitions are imposed. We next point out some of the differences between real and virtual schemas.

First, a base has a unique associated real schema, whereas it may have many virtual schemas, namely all those schemas defined from its real schema. Each of them allows

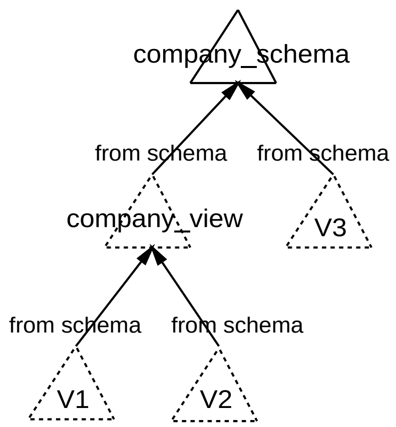

Figure 11: Schema Dependency Tree for Schema `company_schema`

the derivation of a different virtual base. The idea is illustrated in Figure 12. Two virtual schemas, namely `company_view` and `another_view` are created from the real schema `company_schema`. These virtual schemas can be activated on any base instantiated from `company_schema`. Two such bases are shown in the picture, namely `comp_base_1` and `comp_base_2`. The activation of such virtual schemas may result in the creation of four different virtual bases. In the picture, only one of these bases has been given a name.

Many constraints imposed on views come from the following essential principle:

Stored data and views must be independent.

In other words, views can be discarded or changed without compromising stored data. This is not true for real schemas. (The destruction of a schema in O_2 may happen only when all bases corresponding to this schema have been deleted.) The independence of stored data with respect to views is important for many reasons. It allows data structure and behavior to be described based only on real, i.e. conceptual schemas. This is, in a sense, the base counterpart of the unique root rule mentioned before, since it establishes that a base can be an instance of only one real schema, which uniquely determines its structure. As a consequence, views can be used for simulating schema evolution and different schema changes can be experimented without loss of data.

Remark: It may be necessary in certain applications to create objects in virtual schemas. However, one should be able to guarantee that instances of such classes cannot become persistent so as to preserve the independence of stored data with respect to view definitions. This can be achieved, for instance, by disallowing any connection in the class hierarchy between "real classes" defined in a virtual schema and the classes of a real schema. □

Finally, the following technical constraints distinguish virtual and real schemas:

1. Also to guarantee the independence of stored data with respect to views, real schemas cannot import definitions from virtual schemas. On the other hand, virtual schemas can import definitions from any other real or virtual schema.

2. Named objects defined in a virtual schema are virtual, i.e. they exist only during a view activation and are not persistent. Again for assuring database independence with respect to views.

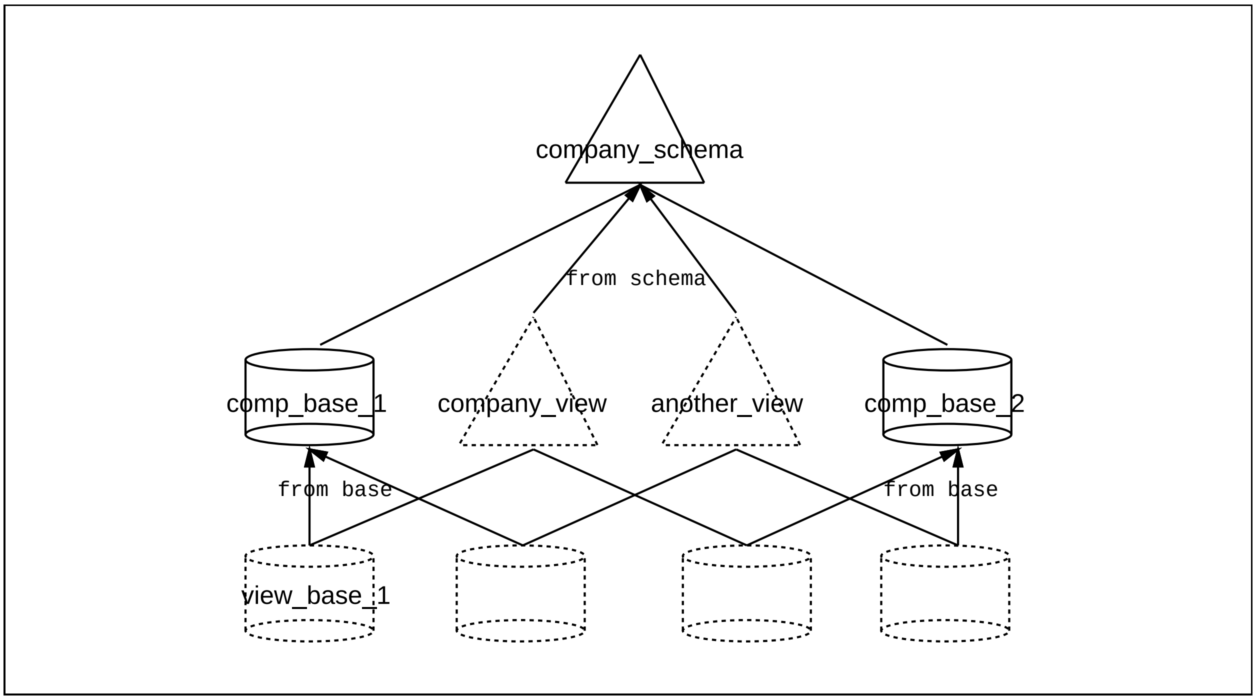

Figure 12: Relationship between Schemas, Views and Bases

3. Virtual classes cannot be explicitly instantiated via the `new` operator, available for object creation. Only instances of real classes can be explicitly created. Real instances are able to become virtual instances as a result of a view activation. In addition, virtual instances can become real instances and vice-versa as a result of an object update.

4. Imported classes can be redefined in a virtual schema but cannot be redefined in a real schema. This issue has been already discussed in Section 2.

4 Generic Views and Database Integration

In this section, we consider a mechanism which provides more flexibility in nesting views, namely the generic views. The primary use of this mechanism is for integration of views. We therefore illustrate it with an example of view integration.

Database integration through views has been considered in a number of recent publications [Mot87, SN88, HR90, KDN90, AB91, GLN91]. Some of these try to provide automatic integration tools [HR90, GLN91]. The integration we show is based on virtual schemas and *class generalization*.

Let us assume the existence of bases `company_base` and `university_base`, instantiated from schemas `company_schema` and `uni_schema` respectively. We are interested in viewing the two heterogeneous bases as a single homogeneous base.

Class `C_Employee` is defined in `company_schema` shown in Figure 1, whereas `U_Employee` is defined in `uni_schema`, as shown in Figure 13. Both are subclasses of class `Person` and therefore share some attributes but each of them defines additional different attributes.

In our first approach to the integration problem, we will define class `Comp_Uni_Employee` through the generalization of these two classes. The relevant definitions of `uni_schema` and the virtual schema `comp_uni_view`, where the generalization is defined, are given in

Figure 13. The dissymmetry of the definition is due to the existence of a root schema which is a useful notion in general but leads here to some inelegance. In some cases, however, the dissymmetry can be eliminated. One might prefer, for instance, to use `general_schema` as the root schema and import the two other schemas for symmetry.

```
schema uni_schema;

import schema general_schema class Person;

class U_Employee inherits Person
...
end;
...
name The_U_Employees: set(U_Employee);
```

```
virtual schema comp_uni_view from company_schema;

import all;
import schema uni_schema class U_Employee;
import base university_base name The_U_Employees;

virtual class Comp_Uni_Employee includes The_U_Employees, The_C_Employees;
```

Figure 13: Database Integration through Generalization

O_2 allows a schema to import names (roots of persistence) from different bases instead of importing them exclusively from other schemas. By importing a name from a schema, the user imports only its definition, no data being involved. When a name is imported from a base, however, all data which can be accessed from that named entry point in the imported base will be able to be accessed from the importing schema.

The virtual class `Comp_Uni_Employee` generalizes classes `U_Employee` and `C_Employee`. It must be noticed that named objects `The_U_Employees` and `The_C_Employees` were used instead of class names when defining the virtual class. This is due to the fact that class extensions are not available in O_2 at the logical level. In order to explicitly maintain the extension of a class, a named set must be defined to which every instance of the class is inserted whenever it is created[6].

Note that the class extension is not necessary since the definition above does provide the characteristic function. Indeed this could be combined with selective characteristic functions in definitions such as that illustrated in Figure 14, where we have a combined use of generalization and specialization.

Remark: The integration implemented through the view `comp_uni_view` defined above will work fine provided that the extensions of `U_Employee` and `C_Employee` are disjoint. But

[6]This is common practice in O_2 programs and is easily achieved by placing the piece of code that inserts newly created objects in the named set in a special method called `init`, which is automatically invoked whenever an object is created. Thus, the simulated class extension is guaranteed to contain every instance of its corresponding class.

```
virtual class Comp_Uni_Employee includes
              U_Employee where condition1,
              C_Employee where condition2
```

Figure 14: Selective Generalization

suppose that an object of U_Employee and an objects of C_Employee may correspond to the same object in real world, playing different roles in distinct databases . Therefore, the relationship between the two generalized classes is not a *category relationship*, but a *role relationship* [SN88] instead and in order to take such relationships into account we must define a *role generalization* [KDN90]. This is done in our system using the imaginary objects of [AB91]. We ignore this issue here. □

In the previous example, the import of names from foreign bases makes explicit reference to base names and commits programs defined in applications on the importing schema to a given base or set of bases. This is in contradiction to our philosophy so far to separate code (methods and application programs) from actual data.

We now introduce a mechanism to do this generically. In particular, we would like to be able to run the programs on different bases provided only at view activation.

View comp_uni_view defined in Figure 13 can be used to integrate no matter which base instantiated from schema company_schema (its root schema), but only with a single base, namely university_base, instantiated from schema uni_schema, since The_U_Employees was imported from it.

We want to modify the definition of comp_uni_view so that it can be used for the integration of any pair of bases instantiated from company_schema and uni_schema respectively.

This motivates the definition of a generic view, whose formal parameters must be replaced at view activation by actual parameters, namely the bases on which the view is to be activated[7]. The type of a formal view parameter is the name of a schema that determines the set of valid bases to be bound to that parameter when the view is activated. Thus, a database integration application can be written as a generic program and be applied to different structurally equivalent tuples of bases.

The generic definition of virtual schema comp_uni_view is given in Figure 15. View comp_uni_view can be activated on any base instantiated from company_schema (its root schema), but at view activation an additional base must be provided, which will be bound to the formal parameter uni_base. The named set The_U_Employees is then imported from this additional base, which must have been instantiated from schema uni_schema, as declared in the signature of the generic view.

Figure 16 illustrates some valid generic view activations and the creation of different named virtual bases. Figure 17 illustrates the relationship between virtual and real bases as well as their respective type schemas, for the virtual bases created in Figure 16.

[7]The base instantiated from the root schema of the view and on which the view is activated can be seen as an implicit parameter, a reasoning similar to that of considering a message receiver in a message passing mechanism as an implicit parameter passed to the method implementing the message.

```
virtual schema comp_uni_view(uni_base:uni_schema) from company_schema;

import all;
import schema uni_schema class U_Employee;
import base uni_base name The_U_Employees;

virtual class Comp_Uni_Employee includes The_U_Employees, The_C_Employees;
```

Figure 15: Defining a Generic View

```
set base comp_base1;
set view comp_uni_view(uni_base1);
virtual base vb1;

set view comp_uni_view(uni_base2);
virtual base vb2;

set base comp_base2;
set view comp_uni_view(uni_base1);
virtual base vb3;

set view comp_uni_view(uni_base2);
virtual base vb4;
```

Figure 16: Valid View Activations

5 Conclusion

In this section, we give some conclusions and sketch the use of views for allowing the control of schema and base evolution (versioning) and for defining alternative integrity constraints. Much work must still be done in order to test the validity of such ideas but they seem to arise as a common extension of the view mechanism.

We have informally presented the functionality of a view mechanism for object-oriented databases. The approach is intended to favor the dynamic nature of intensional and extensional data by allowing it to be easily and intensively explored.

A virtual schema defines a context on which application programs can operate. Its activation, which results in the creation of a virtual base, represents a context switch and provides a means for dealing with the dynamics of database applications which is not favored by conventional object-oriented models.

There are yet several issues not yet fully resolved such as the placement of virtual classes in the hierarchy [Run92, Bor88, Bor89, Lie86, RS91], optimized strategies for solving resolution conflicts, or the lazy evaluation of virtual and imaginary extensions. Furthermore, it is necessary to explore how the view mechanism can be used in several contexts. To conclude, we briefly mention some.

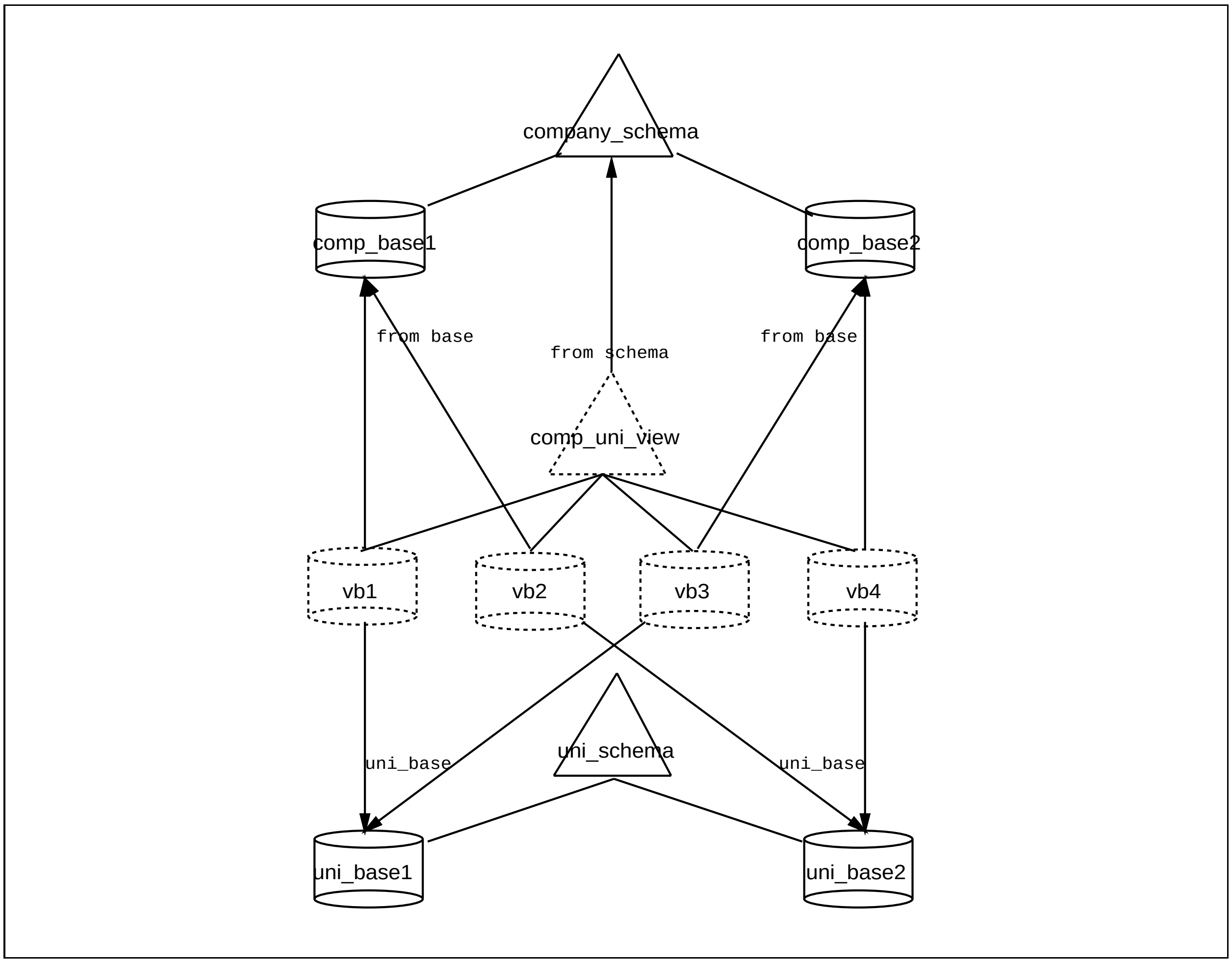

Figure 17: Multiple virtual bases instantiated from the same generic virtual schema and from different real bases.

5.1 View Materialization

The use of the view mechanism for simulating schema evolution has been mentioned. By means of a combined use of virtual attributes, hiding of properties, class specialization, and generic views, schemas may evolve without compromising existing data and application programs.

Once a virtual schema has been proved to be a good alternative for the conceptual real schema it is derived from, the user should be able to request its materialization, upon which the system would generate the new real schema as well as the associated real bases, derived from named virtual bases instantiated from the virtual schema.

For that sake, a materialization command, called `materialize` can be launched on a virtual schema or in one or more virtual bases. A specific virtual base can thereby be materialized, i.e. a duplicate base is created from it where virtual attributes are converted to real attributes and virtual classes become real classes. Objects would also migrate from real classes to new real classes. After materialization, all virtual (intensional and extensional) data has become real. (Observe that one can carefully select which data should become real in the definition of the view.) In particular, imaginary class extensions can be materialized and stored in newly created named sets.

A more sophisticated related mechanism allows multiple versions of a base and associated schema to be maintained. This time, a versioning command, called `version` can be launched on a base. The base is not duplicated but further updates to its contents cause data to be incrementally duplicated (and eventually materialized, i.e. converted from virtual to real status). An intelligent duplication mechanism would be in charge of duplicating and converting data on demand and only when necessary. Different versions can be activated from different views. A version can be thus paraphrased as an incrementally materialized view.

It remains to be seen whether this approach to versions is acceptable in terms of flexibility and performances.

5.2 Schema Evolution

We also believe that the concept of generic view could provide the basis for implementing schema evolution in a very flexible manner. For instance, suppose that new attributes must be added to a given class and that the values of such attributes cannot be derived from existing data in the system, i.e. they are stored in a new separate base. This is a common situation since, as a system evolves, new kinds of information need to be stored and integrated to existing data in a way transparent to the user. New data can be then defined through an auxiliary real schema and kept in a separate auxiliary base. The integration of the auxiliary base with existing data can be specified through a generic view, allowing for different bases to be virtually upgraded through the same view.

5.3 Integrity Constraints

Specialization can also be used to define class invariants, as they are defined in Eiffel [Mey88], for instance. The query expression used in the definition of a specialization is written in a declarative language based on first order logic and implicitly determines an invariant that instances of the class must satisfy.

Let us suppose we want to define a predicate P as the invariant of a given class C. The example is given in Figure 18.

```
virtual class C_With_Invariant includes
      (select X from X in C where not P);
```

Figure 18: Defining a Class with an Invariant

We want the invariant to be checked at certain points. We only have to redefine in class *C_With_Invariant* the methods from class C where we want a special action to be performed if the invariant is not respected. Only instances of class C for which the predicate P evaluates to *false* will be considered as instances of class *C_With_Invariant* and will be manipulated through the redefined methods. This amounts in a sense to an integrity constraint mechanism simulation. Such a mechanism can be integrated to the view mechanism in a more sophisticated manner through the use, for instance, of active rules derived from view specializations, so that, after updates, objects are checked to satisfy the active constraints and appropriate actions can be triggered whenever they are not satisfied.

Acknowledgments: Bernd Amann and Sophie Cluet are thanked for their valuable comments on earlier versions of this paper.

References

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

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

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

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

[Bor89] Alexander Borgida. Type Systems for Querying Class Hierarchies with Non-strict Inheritance. Technical report, Rutgers University, New Brunswick (NJ US), 1989.

[Clu91] Sophie Cluet. *Langages et Optimisation de Requêtes pour Systèmes de Gestion de Base de Données Orientés-Objet*. PhD thesis, Université de Paris Sud - Centre d'Orsay, Paris, France, 1991.

[Deu89] O. Deux et al. The Story of O_2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1989.

[GLN91] Willi Gotthard, Peter C. Lockermann, and Andrea Neufeld. System-Guided View Integration for Object-Oriented Databases. *IEEE Transaction on Knowledge and Data Engineering*, 4(1), Feb. 1991.

[HR90] Stephen Hayne and Sudha Ram. Multi-user view integration system (muvis): An expert system for view integration. *IEEE Data Engineering*, pages 402–409, July 1990.

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

[KDN90] M. Kaul, K. Drosten, and E.J. Neuhold. ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. *IEEE Data Engineering*, pages 2–10, July 1990.

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

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

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, London, 1988.

[Mot87] Amihai Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, July 1987.

[MZ92] Guido Moerkotte and Andreas Zachmann. Multiple Substitutability without Affecting the Taxonomy. In *International Conference on Extending Data Base Technology*, 1992.

[oDMS75] ANSI/X3/SPARC Study Group on Database Management Systems. Interim report. *ACM SIGMOD Bulletin 7,N2*, 1975.

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

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

[Run92] Elke Rundensteiner. A Class Integration Algorithm and Its Application for Supporting Consistent Object Views. Technical Report 92-50, Department of Information and Computer Science, University of Califoria, Irvine, May 1992.

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

[SN88] Michael Schrefl and Erich J. Neuhold. Object Class Definition by Generalization Using Upward Inheritance. In *Proc. IEEE Data Engineering Conf.*, pages 4–13, Los Angeles, Feb. 1988.

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

[Sto92] Michael Stonebraker. The Integration of Rule Systems and Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):415–423, Oct. 1992.

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

[TYH+91] Kazuyuki Tsuda, Kensaku Yamamoto, Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa. MORE: An Object-Oriented Data Model with Facility for Changing Object Structures. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):444–460, Dec. 1991.

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