Object Views Constructed with an Object Algebra

Zoé Lacroix,* Claude Delobel,† Philippe Brèche
Projet Verso - I.N.R.I.A. Rocquencourt
Domaine de Voluceau. BP. 105. 78 153. Le Chesnay Cedex - France
tel: 33 (1) 39 63 56 79 fax: 33 (1) 39 63 56 74
lacroix@saul.cis.upenn.edu, Claude.Delobel@inria.fr , Philippe.Breche@inria.fr

Abstract

We present a formal data model for views in Object DataBase Systems (ODBS) which is a continuation and a formalization of the work presented in [AK89, AB91, ADdS94, dS95a]. The main contribution of this paper is the formalization of the views as a transformation mechanism for databases using a generalization of *referent* proposed in [dS95b]. The view-based transformation is achieved in two steps: an extension of the source base followed by a projection of the extended base. The extension and projection can be carried out using five object algebraic operators which are described in detail in this paper.
Equally important, this paper highlights the challenges involved supporting our view mechanism. We show how this basic model for views point to new perspectives, namely, extension of OQL, transformation of query on the view in a query on the source schema and base, and finally characterization of the class of view-transformations our algebra captures.

`Keywords:` object database systems, views, data model, object algebra, multiple-instantiation, database transformation.

Nous présentons un modèle de données formel pour les vues des Systèmes de Bases de Données Objet qui est une extension et une formalisation des travaux présentés dans [AB91, ADdS94, dS95a]. La principale contribution de cet article est la formalisation des vues comme mécanisme de transformation de bases en utilisant une généralisation du *référent* proposé dans [dS95b]. Cette transformation de vue est effectuée en deux étapes: une extension de la base source suivie d'une projection de la base étendue. L'extension et la projection sont effectuées en utilisant cinq opérateurs de l'algèbre, opérateurs décrits en détail dans cet article.
De plus, cet article met en avant différentes questions soulevées par notre mécanisme de vues. Nous montrons en particulier comment ce modèle de base offre de nouvelles perspectives, à savoir, extension de OQL, transformation d'une requête sur la vue en une requête sur la source, et en dernier lieu, caractérisation de la classe de transformations de vue capturée par notre algèbre.

`Mots-clés:` systèmes de base de données objet, vues, modèle de données, algèbre objet, instanciation multiple, transformation de base de données.

*IRCS, University of Pennsylvania, Suite 400 A, 3401 Walnut ST, Philadelphia PA 19104, USA – Work supported in part by NSF STC grant SBR-8920230 and partly done in the Verso project in INRIA.
†University of Paris XI, LRI, 91 405 Orsay Cedex.

1 Introduction

Database views are a well-known technique used by applications to customize shared data objects without affecting other applications. Views provide the user with the flexibility of defining objects that may evolve and play different roles over a period of time. New virtual schemas and bases are generated through view specifications and constitute to the provision of logical independence to data.

Various proposals in the relational database framework, e.g. *Superviews* [Mot87] and its powerful set of advanced primitives, as well as in the object database domain [AB91, Ber91, TYH$^+$91, SS91, Run92, ADdS94], have been put forward but are supported by very few implementations. In the relational context, commercial databases have implemented restricted subsets of view mechanisms whereas so far, none of the commercial object databases provides user with such mechanisms. To the best of our knowledge, only two prototypes exist in this domain, namely, MultiView/TSE [KR96] and O_2Views[dS95a, dS95b].

Very little has been said about a formal model for object views. The main contribution of this paper is to propose a formal model inspired by the work presented in [dS95b], in line with IQL [AK89] and based on the data model in [AHV95][1] and O_2 [BDK92, LRV88]. This is a continuation of the work presented in [dS95a, dS95b]. We further extend these proposals with the construction of complex referents, that can express multi-instantiation and, consequently new definitions for schema and base. We also propose a view as a transformation of bases following a two-step approach: an extension of a source base followed by a projection of the extended base. This later point is detailed through object algebraic operators that are specified following these two steps.

It seems first natural to consider a view as a query, as in the relational world [AB91, KKS92]. A language defining views in the object model, has to deal with schemas as well as instances of a schema. In our framework, a source schema-base and its view are considered with the following main features: multiple-inheritance, overloading at the schema level, and creation of new objects at the instance level. In our model, classes are populated with *referents*, generalizing the ones introduced in [dS95b]. A referent is a pair (object identifier, class name), that enables us to relax the usual adopted constraint that is an object belongs to a unique class with which the object was created. Object identifiers are also generalized to *complex identifiers*. A complex identifier identifies a new object (imaginary object in [AB91]) as well as specifies the links from which the object is generated. This notion is close to the id-function in [KKS92].

We focus on views obtained from a source schema-base by applying algebraic operators in two steps, a cascade of *extension* and *projection* transformations. This functional approach allows to define views from other view(s). We characterize the class of *view-transformations*, as a subset of the class of db-transformations [AK89]. We provide our model with five basic algebraic operators that express globally transformations on the schema and on the data. This is a simple algebra restricted to the operators *projection*, *natural-join*, *join-specialization*, *specialization* and *union* to express views.

In the sequel, the main assumptions are that views are virtual (as opposed to materialized) and may be considered as queries (composed or not) applied against objects of a source database. Basically, in an algebraic perspective, classes and objects in the view are built up combining the set of object algebra operators. A virtual base is derived from a source. A new class is classified in the hierarchy of classes and populated with respect to the class(es) it derived from. A new attribute is also derived in the sense it is a query on the source. As opposed to [Ber91, Run92], schema evolution cannot be simulated in our context. In that sense, a view is "read only".

The rest of the paper is organized as follows. Section 2 introduces the object data model highlighting the notion of referent, and the possibility of multi-instantiation. We motivate our work with an example which is further developed in the ensuing sections. Section 3 presents how to build a view based on our example. At the same time, we briefly introduce the basic object algebraic operators used (eventually combined) to issue our views. The notion of extension and projection is introduced through examples. Section 4 considers view transformations and formalizes the operators of the algebra. Section 5 concludes on issues and related

[1] There is neither standard nor norm for an object database model.

work points to future work.

2 Model

2.1 Syntax

We consider the set $\mathbf{C}=\{c_1, c_2, ...\}$ of class names, the set $\mathbf{V}=\{integer, string, float, boolean, char\}$ of value class names[2], the set $\mathbf{A}=\{a_1, a_2, ...\}$ of attribute names and the set $\mathbf{M}=\{m_1, m_2, ...\}$ of method names. $\mathbf{C}$, $\mathbf{V}$, $\mathbf{A}$ and $\mathbf{M}$ are pairwise disjoint. Let $\mathbf{D}$ be an infinite set of constants from which are populated the value classes. In the following, when there is no ambiguity, we do not distinguish classes, methods and attributes from their names.

A *signature* is an expression $c \rightarrow \gamma(c')$ where c and c' are class names (only c' may be a value class name and $\gamma(c')$ denotes either a class c' or a set of classes $\{c'\}$. A *definition* of a method m at c (resp. an attribute name a at c) is a pair $(m, c \rightarrow \gamma(c'))$ (resp. $(a, c \rightarrow \gamma(c')))$ where $c \rightarrow \gamma(c')$ is a signature. A method m (resp. an attribute a) is *defined* at c if there exists a definition of m (resp. a) at c. A *class hierarchy* is a 3-tuple $(C, \prec, \Delta)$ where C is a finite subset of $\mathbf{C}$, $\prec$, a binary relation on C (called the *sub-classing relationship* on C) and Δ a finite set of attribute and method definitions only at class names in C. For any c_1, $c_2 \in C$, $c_1 \prec c_2$ expresses that c_1 is a subclass of c_2 or, equivalently, that c_2 is a supclass of c_1. The transitive closure of the sub-classing relationship $\prec$ on C is denoted by $\prec^*$. A set Δ of definitions is *unambiguous* if for each definition of m (resp. attribute a), and each $c \in C$ there is at most one definition of m (resp. a) at c. A method m (resp. attribute a) is *overloaded* if there exists at least two definitions of m (resp. a) at two different classes along an inheritance path. A method m (resp. an attribute a) is *applicable* at c if it is defined at c or defined at some c'' such that $c \prec^* c''$. Δ obeys the *covariance* rule if for each $n \in \mathbf{M} \cup \mathbf{A}$ and for each $c_1, c_2 \in C$ and c'_1, c'_2 of classes in $C \cup \mathbf{V}$, if $(n, c_1 \rightarrow \gamma(c_2))$ and $(n, c'_1 \rightarrow \gamma(c'_2))$ are in Δ such that $c_1 \prec^* c'_1$ then $\gamma(c_2) = \gamma(c'_2)$ otherwise $c_2 \prec^* c'_2$ and $\gamma(c_2) = \{c_2\}$ iff $\gamma(c'_2) = \{c'_2\}$. A class hierarchy $(C, \prec, \Delta)$ is *well-formed* if Δ obeys the *unambiguity* and *covariance* rules. A schema describes the structure of the data stored in a database.

Definition 2.1 [Schema]
A *schema* is a 5-tuple $S = (A, M, C, \prec, \Delta)$, where A and M are respectively finite sets of attribute and method names, and $(C, \prec, \Delta)$, where all attribute and method names occurring in Δ are respectively in A and M, is a well-formed hierarchy. □

It is noted that the hereabove proposed model covers most of the features found in classical object data models. However, for simplification purpose, the types involved in the specification of attributs and methods are limited to class or set of class, and methods have no argument.

2.2 Motivating example: source schema

For illustration purpose, we confront in the present section our definitions to the process of creating a schema and a base. From now on, they are denoted *source* schema S and *source* base I. The source schema (and later on the view construction in section 3) is taken from the example in [MP96]. Roughly, a model is specified for a Conference. It details the participants to the conference who are various kinds of persons, among with authors of papers and panelists. The example is extended as follows: attendants to the conference may be accompanied by people (family or friends) who do not attend the conference. In addition, facilities are provided with persons to rent a car or a motor cycle.

According to Definition 2.1, Figure 1 presents the source schema S as a 5-tuple $(A, M, C, \prec, \Delta)$, corresponding to the schema `Conference_schema` of Figure 2 specified following the O_2 terminology [Tec95]. In the source schema, `PCategory` is a *string* value class (modulo the renaming) taken its values in the constrained domain $\{$`industrial, student, academic`$\}$. Attributes `seater` in `RentACar` and `cubic_capacity`

[2]Modulo the renaming; classes of values constrained on their domain are also valid value class names

in `RentAMotorCycle` take their values in the domains {`2,4`} and {`125,400,600`} (resp.). Since we are not concerned with new instance creations (read only), the method **new** of the various classes does not(!) belong to the set of method names $M = \{$ age, notify, cost $\}$.

C	{Person, AccompaniedGuest, Participant, AuthorP, Panelist, Conference, RentACar, RentAMotorCycle, Date, Address, PCategory}
$\prec$	Participant $\prec$ Person, AccompaniedGuest $\prec$ Person, AuthorP $\prec$ Participant , Panelist $\prec$ Participant
Δ	{ name: Person $\rightarrow$ string , address: Person $\rightarrow$ Address , birthdate: Person $\rightarrow$ Date , age: Person $\rightarrow$ integer , touristPackage: AccompaniedGuest $\rightarrow$ boolean , registrationCode: Participant $\rightarrow$ integer , participatesAs: Participant $\rightarrow$ {PCategory}, notify: Participant $\rightarrow$ string, title: AuthorP $\rightarrow$ string , presenter: AuthorP $\rightarrow$ boolean , inSession: Panelist $\rightarrow$ string , confName: Conference $\rightarrow$ string , confDate: Conference $\rightarrow$ Date, participants: Conference $\rightarrow$ {Participant}, registration: RentACar $\rightarrow$ string , driver: RentACar $\rightarrow$ string , seater: RentACar $\rightarrow$ integer, cost: RentACar $\rightarrow$ integer , registration: RentAMotorCycle $\rightarrow$ string , driver: RentAMotorCycle $\rightarrow$ string , cubic_capacity: RentAMotorCycle $\rightarrow$ integer, cost: RentAMotorCycle $\rightarrow$ integer, month: Date $\rightarrow$ integer , day: Date $\rightarrow$ integer, year: Date $\rightarrow$ integer, street: Address $\rightarrow$ string , zip_code: Address $\rightarrow$ string , city: Address $\rightarrow$ string , country: Address $\rightarrow$ string }

Figure 1: Class hierarchy $(C, \prec, \Delta)$ of the source schema S, `Conference_schema`.

2.3 Semantics

Traditionally, in object models, such as [AdS95, AHV95] objects are associated with only one class, the class where they are defined, have a unique identifier and their attributes are valued. An object **obj** is commonly a pair (**oid**, **val**), where **oid** is an object identifier (*oid*) and **val** its value (the value of all its attributes). In our model, we consider new classes populated with new objects defined from several objects. We generalize the notion of identifier in order to assign to these new objects an identifier which encodes the way it is defined. The set **O**$= \{o_1, o_2, \ldots\}$ of *atomic* identifiers coincides with the set of object identifiers generally considered in the traditional object model. Intuitively, a new object is identified by $\langle o_1, c_1\rangle \wedge \langle o_2, c_2\rangle$ if it is defined from[3] the objects of identifiers o_1 in class c_1 and o_2 in class c_2. We distinguish atomic identifiers $o \in \mathbf{O}$ from *complex* ones of the form $\langle o_1, c_1\rangle \wedge \ldots \wedge \langle o_n, c_n\rangle$. Note that $\langle o_1, c_1\rangle \wedge \ldots \wedge \langle o_n, c_n\rangle$ only denotes the result of the application of an encoding function to $o_1, \ldots, o_n$, which is invertible. The encoding function is not known by the user (who cannot distinguish complex oids from atomic ones either). The id-function defined in [KW89, KKS92] is similar to our encoding function. Contrary to [KW89], new identifiers may occur in distinct classes. The set of all identifiers, atomic and complex ones will be denoted by $\hat{\mathbf{O}}$.

We now consider the concept of *object referent*, previously introduced in [dS95b], which extends the ODBS data model. In a traditional object model, it is commonly accepted that the object identified by o belongs to a single class, where it was created. In this context, it is not necessary that an object explicitly refers to its class. The concept of view we will define later will possibly show the same object in different classes. It is then necessary to explicitly link objects with the different classes they may belong to. The notion of *referent* expresses this link. A referent is a pair $\langle o, c\rangle$, where $o \in \hat{\mathbf{O}}$ and $c \in \mathbf{C}$. Note that in [AdS95] an object may be seen in different contexts of a class and the notion of referent is implicit (it appears when the value assignment is defined since the value of an object may depend on the context). The sets of identifiers and referents are defined as follows:

Definition 2.2 [Identifiers and referents]
The set $\hat{\mathbf{O}}$ of all identifiers and the set $\mathbf{R}$ of all referents are simultaneously defined by the following grammar. For all $i \in \mathbf{O}$ and $c \in \mathbf{C}$, $\hat{\mathbf{O}}$ and $\mathbf{R}$ are respectively the sets of o and r such that

$o \ ::= \ r \mid o \wedge o$ and $r \ ::= \ \langle o, c\rangle \mid \langle i, c\rangle$. The words $r_1 \wedge r_2$ and $r_2 \wedge r_1$ denote the same referent, as expressed by the rule $r_1 \wedge r_2 \equiv r_2 \wedge r_1$, for all $r_1, r_2 \in \mathbf{R}$.
□

[3] The "defined from" relationship is detailed later.

```
        /* Source schema */
create schema Conference_schema;

        /* Source base */
base Conference_base;

        /* Class definitions */

class Person
 public type tuple (
        name: string,
        address: Address,
        birthdate: Date)
 method
        public new()
        public age(): integer
end;

class Participant inherit Person
 public type tuple (
        registrationCode: integer,
        participatesAs: set(PCategory))
 method
        public notify(): string,
        public new()
end;

class AuthorP inherit Participant
 public type tuple (
        title: string,
        presenter: boolean )
 method
        public new()
end;

class Panelist inherit Participant
 public type tuple (
        inSession: string)
 method
        public new()
end;

class AccompaniedGuest inherit Person
 public type tuple (
        touristPackage: boolean )
 method
        public new()
end;

class Conference
 public type tuple (
        confName: string,
        confDate: Date,
        participants: set(Participant))
 method
        public new()
end;

class RentACar
 public type tuple (
        registration: string,
        driver: string,
        seater: integer)
 method
        public new()
        public cost():integer;
end;

class RentAMotorCycle
 public type tuple (
        registration: string,
        driver: string,
        cubic_capacity: integer)
 method
        public new()
        public cost():integer;
end;

class Date
 public type tuple (
        month: integer,
        day: integer,
        year: integer)
 method
        public new();
end;

class Address
 public type tuple (
        street: string,
        zip_code: string,
        city: string,
        country: string)
 method
        public new();
end;
```

Figure 2: The schema `Conference_schema`.

Note that we do not consider complex identifiers or referents defined with constants of $\mathbf{D}$. A referent $< o, c >$ is *flat* if o is an atomic oid, otherwise, it is *nested.*

In this new model, referents are not mutable and identify objects as the object identifiers do it in the traditional model. In particular, a referent points to an object *as it is seen* and *as it behaves* in a given class. The object is still uniquely identified by its oid, but the same object can belong to several different classes and have different values and behaviors. This is a form of *multiple instantiation*, as opposed to the concept of single instantiation, according to which an object is an instance of one and only one single class.

We now introduce the notion of population generalized to referents.

Definition 2.3 [**Class population**]
Let $(A, M, C, \prec, \Delta)$ be a schema. A *referent assignment* $\pi : C \rightarrow 2^{\mathbf{R}}$ associates to each class name in C a finite set of referents such that for each $o \in \hat{\mathbf{O}}$ and $c, c' \in C$, if $\langle o, c' \rangle \in \pi(c)$ then $c' = c$, that is two populations associated to two different classes are disjoint. In addition, the set $\pi^*(c) = \pi(c) \cup_{c' \prec^* c} \pi(c')$ denotes the *extension* of a class c.

A *constant assignment* $\nu : \mathbf{V} \rightarrow 2^{\mathbf{D}}$ associates to each value class name in $\mathbf{V}$ a finite set of constants such that for each $v, v' \in \mathbf{V}$, if $v \neq v'$, then $\nu(v) \cap \nu(v') = \emptyset$.
□

We now introduce a function assignment μ which is a function mapping definitions of Δ to functions the following way.

Definition 2.4 [**Function assignment**]
Let $(A, M, C, \prec, \Delta)$ be a schema. A *function assignment* μ is a function $\mu : \Delta \rightarrow \mathbf{R}^{\mathbf{R}} \cup (2^{\mathbf{R}})^{\mathbf{R}}$ which associates a function to each definition d in Δ such that[4]:

1. if $d = (n, c_1 \rightarrow c_2)$ then $\mu(d) \in \pi^*(c_2)^{\pi^*(c_1)}$;
2. if $d = (n, c_1 \rightarrow \{c_2\})$ then $\mu(d) \in (2^{\pi^*(c_2)})^{\pi^*(c_1)}$;
3. if $d = (n, c_1 \rightarrow \gamma(c_2))$ and $d' = (n, c'_1 \rightarrow \gamma(c'_2))$, and $c_1 \prec^* c'_1$ and $c_2 \prec^* c'_2$ then for all $o \in \hat{\mathbf{O}}$, if $\langle o, c \rangle \in \pi^*(c_1)$ for some $c \prec^* c_1$ and $\langle o, c' \rangle \in \pi^*(c'_1)$ for some $c' \prec^* c'_1$ then $\mu(d)(\langle o, c \rangle) = \mu(d')(\langle o, c' \rangle)$.

□

The third point insures the consistency of function assignment with overloaded methods or attributes on an inheritance path as well as the consistency of function assignment on objects which are in different classes on which common attributes and/or methods may be applicable.

An *instance* I of a schema $S = (A, M, C, \prec, \Delta)$ is a triple $(\pi,\ \mu,\ \nu)$ where π, μ and ν are respectively a referent, a function and a constant assignment. The set of all instances of the schema S is denoted by *inst*(S). Let $I = (\pi,\ \mu,\ \nu)$ be an instance of the schema $S = (A, M, C, \prec, \Delta)$. The set of all referents occurring in I is $R_\pi = \{r \in \mathbf{R} \mid \exists c \in C,\ r \in \pi(c)\}$ (R_π is also denoted by R_I). An object **obj** is now a pair (**ref**, **val**), where **ref** is a referent $\langle o, c \rangle$ and **val** is the value of all attributes and methods applicable at c.

2.4 Motivating example: source base

Given the example of Section 2.2, said schema S is populated. The instance I of the schema S is the base `Conference_base` given in Figure 8 of the appendix. Figure 3 figures out the corresponding class extensions π^* of Definition 2.3. Class extensions are addressed through names similar to the names of the classes.

[4] It is reminded that if E and F are sets, the notation F^E denotes the set of functions from E to F.

Class extension π^* deduced from the referent assignment π	
Class Name c	$\pi^*(c)$
Person	{ < $oidag_1$, AccompaniedGuest>, < $oidag_2$, AccompaniedGuest>, < $oidag_3$, AccompaniedGuest>, < $oidp_1$, Participant>, < $oidp_2$, Participant>, < $oidp_3$, Participant>, < $oidp_4$, AuthorP>, < $oidp_5$, AuthorP>, < $oidp_6$, AuthorP>, < $oidp_7$, AuthorP>, < $oidp_8$, Panelist>, < $oidp_9$, Panelist> }
AccompaniedGuest	{ < $oidag_1$, AccompaniedGuest>, < $oidag_2$, AccompaniedGuest>, < $oidag_3$, AccompaniedGuest> }
Participant	{ < $oidp_1$, Participant>, < $oidp_2$, Participant>, < $oidp_3$, Participant>, < $oidp_4$, AuthorP>, < $oidp_5$, AuthorP>, < $oidp_6$, AuthorP>, < $oidp_7$, AuthorP>, < $oidp_8$, Panelist>, < $oidp_9$, Panelist> }
AuthorP	{ < $oidp_4$, AuthorP>, < $oidp_5$, AuthorP>, < $oidp_6$, AuthorP>, < $oidp_7$, AuthorP> }
Panelist	{ < $oidp_8$, Panelist>, < $oidp_9$, Panelist> }
Conference	{ < $oidc_1$, Conference>, < $oidc_2$, Conference> }
RentACar	{ < $oidrc_1$, RentACar>, < $oidrc_2$, RentACar> }
RentAMotorCycle	{ < $oidrmc_1$, RentAMotorCycle>, < $oidrmc_2$, RentAMotorCycle> }
Date	{ < $oidd_1$, Date>, < $oidd_2$, Date>, < $oidd_3$, Date>, < $oidd_4$, Date>, < $oidd_5$, Date>, < $oidd_6$, Date>, < $oidd_7$, Date>, < $oidd_8$, Date>, < $oidd_9$, Date>, < $oidd_{10}$, Date>, < $oidd_{11}$, Date>, < $oidd_{12}$, Date> }
Address	{ < $oida_1$, Address>, < $oida_2$, Address>, < $oida_3$, Address>, < $oida_4$, Address>, < $oida_5$, Address>, < $oida_6$, Address>, < $oida_7$, Address> }

Figure 3: Class extensions of the base `Conference_base`.

3 Defining a View

In this Section, we first specify a view of the source schema and base of Sections 2.2 and 2.4. Then, we introduce the notions of *extension* and *projection* of schema and base. For that purpose, we use five basic object algebraic operators which make up our basic tool to define a view and show on our example how they participate to the extension and projection stages. These operators are more formally defined in Section 4.

3.1 Example of View

A **view** is specified from the source schema S and its base I to encompass the following information:

- it imports class `Person`, `AccompaniedGuest`, `RentACar`, `RentAVehicle`[5] and all the classes necessary for the type closure in the view (namely, classes `Address` and `Date`).
- class `YoungGuest` is derived from class `AccompaniedGuest`. The extent of `YoungGuest` contains those of the accompanied guests whose age is smaller than 18, moving them to the class `YoungGuest`.
- the view class `ViewParticipant` is derived from class `Participant`. The attribute `birthdate` is hidden as well as the method `age`. The extent of `ViewParticipant` equals to that of `Participant`.
- a view class `ActiveParticipant` is derived from `ViewParticipant`, with, in addition, a method `conference():string`. The extent of `ActiveParticipant` is the union of all panelists and those participating authors who are also presenters.
- a new class `ConfPaper` is introduced, holding an association of conference name and presenting author. The instances of `ConfPaper` do not have corresponding instances in the base schema.
- another new class `RentAVehicle` is introduced that collects the common attributes and methods of `RentACar` and `RentAMotorCycle`. The extent of `RentAVehicle` is the union of all rents both cars and motor cycles.

[5] Classes which are not explicitly imported are implicitly masked in the view.

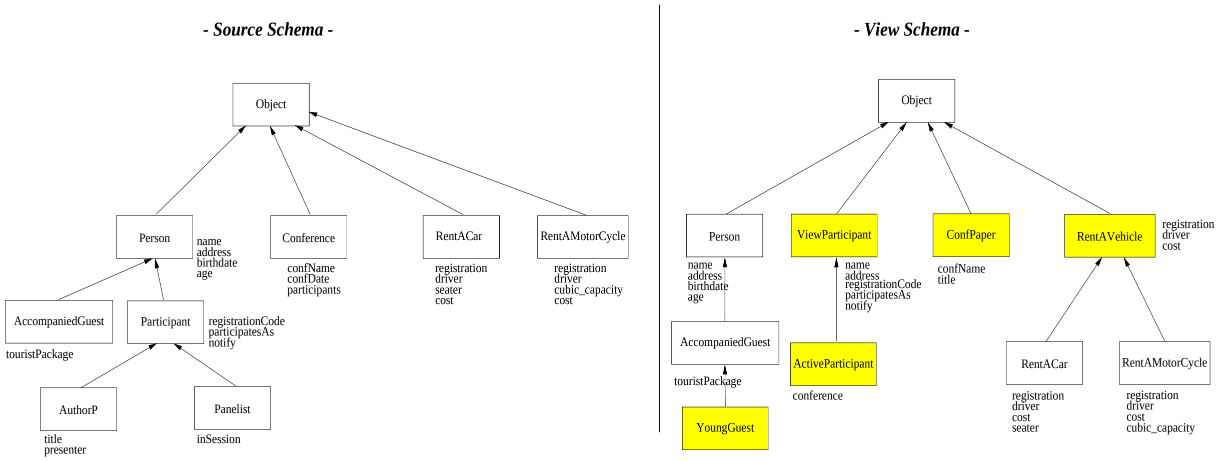

Figure 4: Class hierarchies of the source and view schemas.

Figure 4 represents the class hierarchies of the source schema S and of view schema S'. Classes `Address` and `Date` are omitted.

The resulting class hierarchy of the view-schema is given in the appendix in Section A.2.1.

The classes are populated as described in Figure 5. The class `YoungGuest` is populated with a single nested referent $<< oidag_3$, AccompaniedGuest$>$, YoungGuest $>$, defined from $< oidag_3$, AccompaniedGuest$>$. It corresponds to an accompanied guest, younger than 18 in the source base. Since the referent $< oidag_3$, AccompaniedGuest$>$ is moved to the virtual class `YoungGuest`, subclass of `AccompaniedGuest`, two referents only are retained in `AccompaniedGuest`. Finally, since all the participants are viewed from the new class `ViewParticipant` and since the classes `Participant`, `Author` and `Panelist` are hidden, objects of `Participant` are migrated automatically up to the extension of `Person`. In the view, this extension is still populated with three referents from `AccompaniedGuest` and the ones migrated from the extension of `Participant`.

The referents of class `ViewParticipant` as well as `ActiveParticipant` defined from the source-class `Participant` or its subclasses are also nested.

The virtual class `ConfPaper` is populated with new objects. Each object is a paper defined from a conference and an author (the one who presents the paper).

3.2 Basic operators to define Views

A view is the result of two successive steps: an *extension* followed by a *projection* step. As a base tool to specify the view found in the previous section, we propose the operators *specialization*, *join-specialization*, *join*, *generalization* and *projection*. The first four express the extension of the schema and base, the last one expresses the projection of the schema and base. The following sections detail how we come up with the new classes (virtual classes) in our example of view, extending and projecting the source schema and base (respectively described in Figure 1 and Figure 3).

3.2.1 Specialization

Intuitively, a virtual class is a *specialization* of a class c if it is a subclass of c, populated with objects of class c (these objects are moved to the virtual class) and/or the extension of c (these objects are "duplicated" in the virtual class). New attributes and/or methods may be defined at the virtual class. Note that when no new method or attribute is defined at the virtual class, the specialization realizes a selection on objects in class c.

The view issued from the source schema defines class `YoungGuest` as a virtual subclass of class `AccompaniedGuest`, populated with the guests younger than 18. The class `YoungGuest` is populated querying the source base: the oids in `AccompaniedGuest` whose age is smaller than 18 are selected to be

Class extension π_V^* deduced from the referent assignment π_V	
Class Name c	$\pi_V^*(c)$
Person	$\{ < oidag_1$, AccompaniedGuest$>$, $< oidag_2$, AccompaniedGuest$>$, $< oidag_3$, YoungGuest$>$, $< oidp_1$, Person$>$, $< oidp_2$, Person$>$, $< oidp_3$, Person$>$, $< oidp_4$, Person$>$, $< oidp_5$, Person$>$, $< oidp_6$, Person$>$, $< oidp_7$, Person$>$, $< oidp_8$, Person$>$, $< oidp_9$, Person$> \}$
AccompaniedGuest	$\{ < oidag_1$, AccompaniedGuest$>$, $< oidag_2$, AccompaniedGuest$>\}$
YoungGuest	$\{< oidag_3$, YoungGuest $> \}$
ViewParticipant	$\{ << oidp_1$,Participant$>$,ViewParticipant$>$, $<< oidp_2$,Participant$>$,ViewParticipant$>$, $<< oidp_3$,Participant$>$,ViewParticipant$>$, $<< oidp_6$,AuthorP$>$,ViewParticipant$>$, $<< oidp_4$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_5$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_7$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_8$,Panelist$>$,ActiveParticipant$>$, $<< oidp_9$,Panelist$>$,ActiveParticipant$>\}$
ActiveParticipant	$\{ << oidp_4$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_5$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_7$,AuthorP$>$,ActiveParticipant$>$, $<< oidp_8$,Panelist$>$,ActiveParticipant$>$, $<< oidp_9$,Panelist $>$,ActiveParticipant$> \}$
ConfPaper	$\{ << oidc_1$, Conference$> \wedge < oidp_4$, AuthorP$>>$, ConfPaper $>$, $<< oidc_1$, Conference$> \wedge < oidp_5$, AuthorP$>>$, ConfPaper $>$, $<< oidc_2$, Conference$> \wedge < oidp_7$, AuthorP$>>$, ConfPaper $> \}$
RentAVehicle	$\{ < oidrc_1$, RentACar$>$, $< oidrc_2$, RentACar$>$, $< oidrmc_1$, RentAMotorCycle$>$, $< oidrmc_2$, RentAMotorCycle$>\}$
RentACar	$\{ < oidrc_1$, RentACar$>$, $< oidrc_2$, RentACar$> \}$
RentAMotorCycle	$\{ < oidrmc_1$, RentAMotorCycle$>$, $< oidrmc_2$, RentAMotorCycle$> \}$
Date	$\{ < oidd_8$,Date$>$, $< oidd_9$,Date$>$ $< oidd_{10}$,Date$>$, $< oidd_{11}$,Date$>$, $< oidd_{12}$,Date$> \}$
Address	$\{ < oida_1$,Address$>$, $< oida_2$,Address$>$, $< oida_3$,Address$>$, $< oida_4$,Address$>$, $< oida_5$,Address$>$, $< oida_6$,Address$>$, $< oida_7$,Address$> \}$

Figure 5: Class extensions of the base `Views_Conference_base`.

moved to the class `YoungGuest`. In the view, these objects are no longer in the population of `AccompaniedGuest`, but in its extension.

The specialization consists of a small change at the level of the schema (introduction of the new class), and many changes at the level of the instance (objects are moved to the new class). No new object is considered in the view! It follows that there is no new complex oid involved, but new referents. All attributes and methods defined or applicable at `AccompaniedGuests` are applicable at `YoungGuest`.

When the class `ViewParticipant` is defined, class `ActiveParticipant` is defined as its specialization. All attributes and methods defined at class `ViewParticipant` are applicable at class `ActiveParticipant`. The definition (conference, ActiveParticipant $\rightarrow$ string) of the new method is added to Δ_V.

Note that all previous changes consist in adding information to the base (add a new class, a new method). These transformations are part of a schema *extension*.

3.2.2 Join-specialization

A virtual class may be the *join-specialization* of classes[6] which have the same key attributes and methods. It is a subclass of all classes that it specializes and it is populated with new objects. As the specialization, a restriction on the population and/or new attributes or methods are possible. A join-specialization may be seen as an intersection of classes. In case of a join-specialization of n classes, a new referent encodes the fact that n objects come from n different classes, with n different oids; these objects have exactly the same values for their key attributes and methods. The new referent $<< o_1, c_1 > \wedge \ldots \wedge < o_n, c_n >, c >$ in the population of the join-specialized class c signifies that the objects of oids o_1 in class c_1, and ... o_n in class c_n are the "same". In the join-specialized class, these objects are identified with the new complex

[6]When the virtual class is the join-specialization of a single class, then it coincides with its specialization.

oid $< o_1, c_1 > \wedge \ldots \wedge < o_n, c_n >$. A join-specialization does not consist in creating new objects, but in identifying objects with new oids.

Assume that only the attribute `participatesAs` is defined at class `Participant`. Example of a join-specialization is class `AuthorPanelist` subclass of both `AuthorP` and `Panelist`. Considering that all the attributes applicable at class `Participant` are the key attributes, the class `AuthorPanelist` is populated with a single referent $<< oidp4, \text{AuthorP} > \wedge < oidp9, \text{Panelist} >, \text{AuthorPanelist} >$. The referent $<< oidp4, \text{AuthorP} > \wedge < oidp9, \text{Panelist} >, \text{AuthorPanelist} >$ identifies Abiteboul as the unique person and participant who is both an author and a panelist.

3.2.3 Join

The *join* consists in really creating new objects. A virtual class which is a join of classes is a subclass of the root class (`Object` in Figure 4) of the class hierarchy. The virtual class `ConfPaper` is populated with new objects $<< oidc$, Conference$> \wedge < oidp$, AuthorP$>$, ConfPaper $>$ defined from objects of classes `AuthorP` and `Conference`. The new class `ConfPaper` cannot be compared with the classes `AuthorP` or `Conference`.

A join may be also used to define a class which could be defined with a join-specialization operator, but which is a subclass of the root class in the view-schema. For instance, in the example, the class `ViewParticipant` is defined as a join of class `Participant`.

3.2.4 Generalization

In the case of a *generalization*, the virtual class is a supclass of the classes in the hierarchy. The idea of the generalization consists in a new supclass of classes on which are defined common attributes and/or methods (behavior generalization). Consider for instance the classes `RentACar` and `RentAMotorCycle` of the source schema. The expected view consists of the new class `RentAVehicle`, supclass of both `RentACar` and `RentAMotorCycle`. The new class is populated by extension, but the set of definitions has to be changed the following way. All definitions of common attributes and methods must be also defined at the virtual class `RentAVehicle`.

3.2.5 Projection

The *projection* consists in hiding information, class or property in a class.

Hiding a class requires some changes at the level of Δ. In our view, classes `AuthorP`, `Panelist`, `Participant` and `ConfPaper` are hidden. A first projection of class `Participant` would consist in defining at `AuthorP` (the same holds for `Panelist`) those of the attributes and methods that were defined at `Participant` and not defined at `AuthorP` (`Panelist` resp.). Moreover, the definition of any attribute or method that maps to objects in `Participant` must map to `Person`.

Note that a view containing only the class `YoungGuest` and not the class `AccompaniedGuest` would result in two successive steps: extension (the class `AccompaniedGuest` is preserved) and projection (the class `AccompaniedGuest` is hidden).

A projection also results in hiding attributes and/or methods precised in the parameters. In our view, the definitions of the attribute `birthdate` is removed from Δ at the projection-step.

4 ALG^V: an algebra to express view-transformations

4.1 View-transformation

A view is commonly seen as a query. In the object-oriented model, a *query* must satisfy a genericity criterion more elaborated that in the relational model [CH80]. In the object-oriented model, a query can interpret neither constants nor object identifiers. As pointed out in [AHV95], a query language should *have the ability to create oids*. A query is no longer a mapping between instances, but a relation satisfying a functionality criterion which ensures the output not to depend on the chosen new oids. This last condition is rather restrictive since it enforces two possible outputs to be isomorphic. The condition may be lightened with a condition on observation of values instead of identifiers [Kos95]. A *determinate query* [AHV95] is defined as follows. We first consider the model only with atomic identifiers and no methods. A **D**-isomorphism is a one-one function on **D** which respects the value classes. An **O**-isomorphism is a one-one function on **O** extended to an isomorphism of instances. Let S and S' be two schemas, a *determinate query* Q is a computable subset of $inst(S) \times inst(S')$ such that for each **D**-isomorphism h of instances, if $(I, J) \in Q$ then $(h(I), h(J)) \in Q$ (genericity), for each two **O**-isomorphic pairs (I, J) and (I', J') of instances, if $(I, J) \in Q$ then $(I', J') \in Q$ (well-defined) and for each two pairs (I, J) and (I, J') of instances in Q, there exists an **O**-isomorphism h such that $J' = h(J)$ (functionality).

We now adapt the notion of query to complex identifiers in order to consider *deterministic queries* instead of non-deterministic ones. Let h be a one-one function on **O** (the set of atomic identifiers). h can be easily extended to the set of complex identifiers $\hat{\mathbf{O}}$ such that if $o'_1 = h(o_1)$, ..., $o'_n = h(o_n)$ and $o = \langle o_1, c_1 \rangle \wedge \ldots \wedge \langle o_n, c_n \rangle$, then $h(o) = \langle o'_1, c_1 \rangle \wedge \ldots \wedge \langle o'_n, c_n \rangle$ and to referents. Finally h can be extended to an isomorphism of instances. The resulting extension is an $\hat{\mathbf{O}}$-isomorphism. A *view-transformation* from $inst(S)$ to $inst(S')$ is a deterministic query τ such that if $I \in inst(S)$, (1) if a class c appears in both schemas then c is populated in $\tau(I)$ with respect to the population by extension of c in I; and (2) the global population of $\tau(I)$ (the set $R_{\tau(I)}$ of referents of $\tau(I)$) is defined with respect to the global population of I, that is, one cannot build up more referents than those based on complex oids constructed with conjunctions of referents issued from the source base. More formally, a view-transformation is defined as follows.

Definition 4.1 [view-transformation]
Let $S = (A, C, \prec, \Delta)$ and $S' = (A', C', \prec', \Delta')$ be two schema. A *view-transformation* τ from $inst(S)$ to $inst(S')$ is a computable mapping such that:

1. for each **D**-isomorphism h of instances and for each $I \in inst(S)$, then $h(\tau(I)) = \tau(h(I))$,
2. for each two $\hat{\mathbf{O}}$-isomorphic pairs of instances (I, J) and (I', J'), if $J = \tau(I)$ then $J' = \tau(I')$,
3. for each $\hat{\mathbf{O}}$-automorphism h of I, then $h(\tau(I)) = \tau(I)$,
4. for every two instances I and J such that $J = \tau(I)$, then $R_J \subseteq R_I \cup \bigcup_{n \geq 1} \{\langle r_1 \wedge ... \wedge r_n, c' \rangle \mid c' \in C' \text{ and } r_i \in R_I\}$,
5. $\forall c \in C \cap C'$, if $\langle r_1 \wedge ... \wedge r_n, c' \rangle \in \pi'^*(c)$ then there exists i such that $r_i \in \pi^*(c)$, and
6. $\forall c \in C \cap C'$, if $\langle r, c \rangle \in \pi'(c)$ then $r \in \pi^*(c) - \pi(c)$ or $< r, c > \in \pi(c)$.

S is the *source-schema* and S' is the *view-schema*. □

The three first criteria are respectively genericity, well-defined, and functionality. The two last ones insures that if a class c appears in both schemas then c is populated in $\tau(I)$ with respect to the population by extension of c in I.

An instance J of schema S' is a *view* of an instance I of schema S if there exists a view-transformation $\tau \in inst(S) \times inst(S')$ such that $J = \tau(I)$. The instance I is the *source-instance*. A view-transformation τ is defined given two schemas S and S' without knowing the relationship between them. It appears that it is necessary to introduce a common framework in which the view-schema is related to the source-schema. In [AK89], Abiteboul and Kannelakis introduce the notion of *projection* to express db-transformations, later completed by the notion of *extension* in [dS95b]. Intuitively, a db-transformation of $inst(S) \times inst(S')$ results from two successive steps: an extension from S and a projection to S'. These steps express the relationships between the two schemas S and S' of a db-transformation. Intuitively, an *extension* consists in defining a schema ES (*extended-schema*) extending the source-schema with new classes *derived* from the source-classes, attributes and/or methods, when a *projection* consists in hiding classes, attributes and/or methods from the extended-schema to obtain S'.

4.2 Extension

We now define a simple algebra ALG^E based on the four operators *specialization*, *join-specialization*, *join* and *generalization* which expresses the extension step of the definition of a subclass of view-transformations.

An expression of ALG^E is a list $[e_1, \ldots, e_m]$ of *extension-instructions* as shown on the table on the right, where $n \geq 1$, and $c', c_1, \ldots, c_n$ are class names (not value class names!) and P is an optional list of parameters which precises the semantics of the operator. A set of parameters P may contain a definition of the population (an extended OQL query), definitions of attributes, and/or definitions of methods.

extension-instructions
$c' \Leftarrow \text{Specialization}_P(c_1)$ $c' \Leftarrow \text{Join-Specialization}_P(c_1, \ldots, c_n)$ $c' \Leftarrow \text{Join}_P(c_1, \ldots, c_n)$ $c' \Leftarrow \text{Generalization}(c_1, \ldots, c_n)$

Definition 4.2 Parameters of a virtual class c'
The list of parameters P of c' is of the form $[q, d_1, q_1, \ldots, d_m, q_m]$, where $d_1, \ldots, d_m$ are definitions of the form $(n_i, c' \to \gamma(c_{j_i}))$ and $q, q_1, \ldots, q_m$ are queries. A_P and M_P respectively denote the set of attribute and method names occurring in the definitions in P. □

Names in A_P and M_P are respectively *virtual attribute* and *virtual method* names. Virtual attributes and methods in the parameters have new names. A rename operator, whose semantics depends on a checking of consistency, may be defined to rename attributes or methods at the projection step.

In the following, we express the semantics of the extension-instructions when applying them on a schema and its base. Each extension-instruction applies on a schema S_i and results in a schema S_{i+1}. An instruction e of parameters $P = [q, d_1, q_1, \ldots, d_m, q_m]$ defining a virtual class c' can apply on schema S_i if $c' \notin C_i$, $A_i \cap A_P = \emptyset$, $M_i \cap M_P = \emptyset$. When e applies on S_i, the set of class-names C_i is obviously extended with the new class name c' and the class names c such that there exists $d_i = (n_i, c' \to \gamma(c)) \in P$ and $c \notin C_i \cup \{c'\}$, $M_{i+1} = M_i \cup M_P$, $A_{i+1} = A_i \cup A_P$, $\prec_{i+1}$, and Δ_{i+1} are extensions of $\prec_i$, and Δ_i, as described subsequently.

instruction	$c' \Leftarrow \text{Specialization}_P(c_1)$	
schema	$\prec_{i+1}$	$c' \prec_{i+1} c$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \to \gamma(c)) \in P \mid c \in C_{i+1}\}$

instruction	$c' \Leftarrow \text{Join-Specialization}_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
schema	$\prec_{i+1}$	$\forall j\ c' \prec_{i+1} c_j$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \to \gamma(c)) \in P \mid c \in C_{i+1}\}$

instruction	$c' \Leftarrow \text{Join}_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
schema	$\prec_{i+1}$	$\prec_{i+1} = \prec_i$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \to \gamma(c)) \in P \mid c'' \in C_{i+1}\}$ $\cup \{(a, c' \to \gamma(c)) \mid a \in A_i \textit{ and } \exists j\ \exists c'_j\ c_j \preceq^*_i c'_j\ (a, c'_j \to \gamma(c)) \in \Delta_i\}$

All attributes applicable at c_1, ..., c_n are defined at the join by default, when no method is defined by default.

instruction	$c' \Leftarrow \text{Generalization}(c_1, \ldots, c_n)$, $(n \geq 1)$	
schema	$\prec_{i+1}$	$\forall j$, $c_j \prec_{i+1} c'$ and for all c such that $\forall j$, $c_j \prec_i c$, then $c' \prec_{i+1} c$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \to \gamma(c)) \mid \forall j\ (n, c_j \to \gamma(c)) \in \Delta_i$ $\textit{or } [(n, c'_j \to \gamma(c)) \in \Delta_i \textit{ and } c_j \prec^* c'_j]\}$

Note that the common attributes and methods applicable at all c_j are defined at the generalized class c'.

A list of m extension-instructions $E = [e_1, \ldots, e_m]$ is an *expression* of ALG^E. The *extension-schema* ES defined by E from the source-schema S is the schema S_m such that $S_0 = S$ and $\forall i, 1 \leq i \leq m$, $S_i = e_i(S_{i-1})$. The expressions of ALG^E are the *extension-expressions*.

Example 4.1 The extended schema pictured out in Figure 6 is obtained by the following extension expression:

[YoungGuest $\Leftarrow$ Specialization$_{P_Y}$(AccompaniedGuest),
ViewParticipant $\Leftarrow$ Join$_{P_{VP}}$(Participant),
ActiveParticipant $\Leftarrow$ Specialization$_{P_{AP}}$(ViewParticipant),
ConfPaper $\Leftarrow$ Join$_{P_P}$(Conference,AuthorP),
RentAVehicule $\Leftarrow$ Generalization(RentACar, RentAMotorCycle)]

The parameters are detailed in the sequel. Note that at this point, the OQL language requires to be extended in order to address a referent and its content. For limitation purpose, we do not provide the grammar of the language and instead, we do write the queries in an intuitive Prolog like language[7].

P_Y = [(YoungGuest($< o, \texttt{YoungGuest} >$): $-$ AccompaniedGuest(g),
g=$< o, \texttt{AccompaniedGuest} >$, g$\rightarrow$**age** $\leq$ 18)]

P_{VP} = [(ViewParticipant($< p, \texttt{ViewParticipant} >$): $-$ Participant(p))]

P_{AP} = [(ActiveParticipant($<< o, \texttt{AuthorP} >, \texttt{ActiveParticipant} >$): $-$ ViewParticipant(a),
a=$<< o, \texttt{AuthorP} >, \texttt{ViewParticipant} >$, $< o, \texttt{AuthorP} >$.presenter=true;
ActiveParticipant($<< o, \texttt{Panelist} >, \texttt{ActiveParticipant} >$): $-$ ViewParticipant(a),
a=$<< o, \texttt{Panelist} >, \texttt{ViewParticipant} >$),
(conference, ActiveParticipant $\rightarrow$ string),
(p $\rightarrow$ `conference`= c.confName :- ActiveParticipant(p),
p=$<< o, \texttt{AuthorP} >, \texttt{ActiveParticipant} >$,
$< o, \texttt{AuthorP} >$ in c.participants, Conference(c);
p $\rightarrow$ `conference`= c.confName :- ActiveParticipant(p),
p=$<< o, \texttt{Panelist} >, \texttt{ActiveParticipant} >$,
$< o, \texttt{Panelist} >$ in c.participants, Conference(c))]

P_P = [(ConfPaper($< cp, \texttt{ConfPaper} >$): $-$ ConfPaper(cp),
cp=$<< oc, \texttt{Conference} > \wedge < op, \texttt{AuthorP} >, \texttt{ConfPaper} >$,
$< op, \texttt{AuthorP} >$ in $< oc, \texttt{Conference} >$.participants,
$< op, \texttt{AuthorP} >$.presenter=true)]

We now describe the second aspect of the semantics of extension-instructions, which is the impact on the instance. Let I be the source-instance of schema $S = (A, M, C, \prec, \Delta)$. If there is no condition on the population given by the query q in the list $P = [q, d_1, q_1, \ldots, d_m, q_m]$, then q is the identity query ($q(I) = I$). The constant assignment ν is not affected by the extension.

$c' \Leftarrow$ Generalization$(c_1, \ldots, c_n)$, $(n \geq 1)$	
population	$\pi_{i+1}(c') = \emptyset$
function	$\forall d = (n, c' \rightarrow \gamma(c)) \in \Delta_{i+1} - \Delta_i,\ d_j = (n, c'_j \rightarrow \gamma(c)),\ c_j \preceq^* c'_j,\ \mu_{i+1}(d) = \mu_i(d_j)$

$c' \Leftarrow$ Specialization$_P(c)$	
population	$\pi_{i+1}(c') = \{\langle r, c' \rangle \mid \exists c'' \in C_0,\ r = \langle o, c'' \rangle,\ r \in \pi_i^*(c) \ and \ r \in q(I_i)\}$ $\cup \{\langle o, c' \rangle \mid \exists c'' \notin C_0,\ \langle o, c'' \rangle \in \pi_i^*(c) \ and \ \langle o, c'' \rangle \in q(I_i)\}$ $\pi_{i+1}(c) = \{r \mid r \in \pi_i(c) \ and \ r \notin q(I_i)\}$
function	$\forall d_j = (n_j, c' \rightarrow \gamma(c)) \in P,\ \mu_{i+1}(d_j) = q_j$

[7]For simplicity, we do not use the function and constant assignments.

The referents $< r, c' >$ populating the specialized class are nested only if r is in the population of a source class (in C_o).

$c' \Leftarrow \text{Join-Specialization}_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
population	$\pi_{i+1}(c') = \{< o_1 \wedge \ldots \wedge o_n, c' > \mid [(o_j = \langle o'_j, c'_j \rangle \in \pi^*_i(c_j),\ \langle o'_j, c'_j \rangle \in q(I_i))$ $or\ (o_j = \langle o_{j_1}, c_{j_1} \rangle \wedge \ldots \wedge \langle o_{j_{k_j}}, c_{j_{k_j}} \rangle, < o_j, c'_j > \in \pi^*_i(c_j),\ \langle o_j, c'_j \rangle \in q(I_i))]$ $and\ \forall j \forall k \langle o^{(')}_j, c'_j \rangle \bowtie \langle o^{(')}_k, c'_k \rangle\}$ $\pi_{i+1}(c_j) = \{\langle o_j, c_j \rangle \mid \langle o_j, c_j \rangle \in \pi_i(c_j)\ such\ that\ \langle o_j, c'_j \rangle \notin q(I_i)$ $or\ \exists c_k\ \forall \langle o_k, c'_k \rangle \in \pi^*_i(c_k), \langle o_j, c'_j \rangle \not\bowtie \langle o_k, c'_k \rangle\}$
function	$\forall d_j = (n_j, c' \rightarrow \gamma(c)) \in P,\ \mu_{i+1}(d_j) = q_j$

$o^{(')}_j$ denotes either o_j or o'_j depending of the concerned referent of $\pi^*_i(c_j)$.

$c' \Leftarrow \text{Join}_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
population	$\pi_{i+1}(c') = \{< o_1 \wedge \ldots \wedge o_n, c' > \mid [(o_j = \langle o'_j, c'_j \rangle \in \pi^*_i(c_j),\ \langle o'_j, c'_j \rangle \in q(I_i))$ $or\ (o_j = \langle o_{j_1}, c_{j_1} \rangle \wedge \ldots \wedge \langle o_{j_{k_j}}, c_{j_{k_j}} \rangle, < o_j, c'_j > \in \pi^*_i(c_j),\ \langle o_j, c'_j \rangle \in q(I_i))]$ $and\ \forall j \forall k \langle o^{(')}_j, c'_j \rangle \bowtie \langle o^{(')}_k, c'_k \rangle\}$
function	$\forall d = (n, c' \rightarrow \gamma(c)) \in \Delta_{i+1},\ d_j = (n, c'_j \rightarrow \gamma(c)) \in \Delta_i,\ c_j \preceq^* c'_j,\ \mu_{i+1}(d) = \mu_i(d_j)$ $\forall d_j = (n_j, c' \rightarrow \gamma(c)) \in P,\ \mu_{i+1}(d_j) = q_j$

Note that there is no distinction between virtual attributes and virtual methods in a view: in both cases, the function assignment is a query.

$\bowtie$ is a binary relation on the set of referents and has the following meaning (in the aim of the *merge* operator defined in [AdS95]). $\langle o_j, c'_j \rangle \bowtie \langle o_k, c'_k \rangle$ is defined if for any common method or attribute name respectively defined by d_j when applicable at c_j and by d_k when applicable at c_k,

$$\mu(d_j)(\langle o_j, c'_j \rangle) = \mu(d_k)(\langle o_k, c'_k \rangle),$$

and undefined otherwise.

Let $E = [e_1, \ldots, e_m]$ be an expression of ALG^E, list of m extension-instructions. The *extension-instance* EI defined by E from the source-schema S is the instance I_m such that $I_0 = I$ and $\forall i, 1 \leq i \leq m\ I_i = e_i(I_{i-1})$. It is easy to see that E is a *view-transformation* as defined in Section 4.1. We now describe the projection step introducing an algebra ALG^P.

4.3 Projection

Intuitively, the projection step consists in hiding information: hiding classes, attributes, and/or methods. A sophisticated *projection* operator accomplishes this task. Let P be a list of parameters $[q, d_1, \ldots, d_m]$ (no query q_j assigning a definition d_j is necessary, since the parameters are the list of hidden objects, with the query q, and hidden definitions of attributes or/and methods), a projection expression is a list of *projection-instructions* of the form Projection$_P(c)$. The impact of the application of a projection-instruction on a schema is the following.

$\text{Projection}_P(c)$	
C_{i+1}	$C_{i+1} = C_i - \{c\}$
$\prec_{i+1}$	$\forall j\ \forall k$ if $c_j \prec_i c$ and $c \prec_i c_k$ then $c_j \prec_{i+1} c_k$
Δ_{i+1}	$\Delta_{i+1} = [\Delta_i \cup \{(n, c_j \rightarrow \gamma(c')) \mid (n, c \rightarrow \gamma(c')) \in \Delta_i\ and\ c_j \prec_i c\ and\ (n, c_j \rightarrow \gamma(c')) \notin \Delta_i\}$ $\cup \{(n, c' \rightarrow \gamma(c_j)) \mid (n, c' \rightarrow \gamma(c_j)) \in \Delta_i\ and\ c \prec_i c_j\ and\ (n, c' \rightarrow \gamma(c_j)) \notin \Delta_i\}]$ $-[\{d \mid d \in P\} \cup \{(n, c \rightarrow \gamma(c')) \mid (n, c \rightarrow \gamma(c')) \in \Delta_i\}$ $\cup \{(n, c' \rightarrow \gamma(c)) \mid (n, c' \rightarrow \gamma(c)) \in \Delta_i\}]$

Note that when no class name is precised in the projection-instruction, $\text{Projection}_P()$, only the set of definitions is affected: $\Delta_{i+1} = \Delta_i - \{d \mid d \in P\}$.

Example 4.2 The view-schema given in Figure 4 is the result of the following projection-expression on the extended-schema ES of Figure 6. The corresponding class hierarchy $(EC, \prec_e, \Delta_e)$ is given in Figure 7.

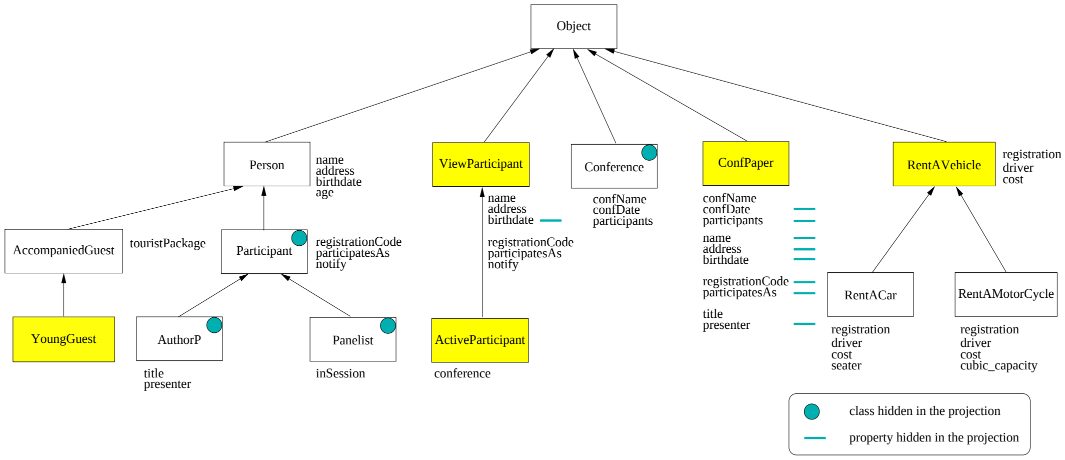

Figure 6: Class hierarchy of the extended schema ES.

EC	{ Person, AccompaniedGuest, YoungGuest, Participant, AuthorP, Panelist, ViewParticipant, ActiveParticipant, Conference, ConfPaper, RentACar, RentAMotorCycle, RentAVehicle, Date, Address, PCategory }
$\prec_e$	AccompaniedGuest $\prec_e$ Person , Participant $\prec_e$ Person , YoungGuest $\prec_e$ AccompaniedGuest , AuthorP $\prec_e$ Participant , Panelist $\prec_e$ Participant , ActiveParticipant $\prec_e$ ViewParticipant , RentACar $\prec_e$ RentAVehicle, RentAMotorCycle $\prec_e$ RentAVehicle
Δ_e	{ name: Person $\rightarrow$ string , address: Person $\rightarrow$ Address , birthdate: Person $\rightarrow$ Date , age: Person $\rightarrow$ integer , touristPackage: AccompaniedGuest $\rightarrow$ boolean , registrationCode: Participant $\rightarrow$ integer , participatesAs: Participant $\rightarrow$ {PCategory}, notify: Participant $\rightarrow$ string, title: AuthorP $\rightarrow$ string , presenter: AuthorP $\rightarrow$ boolean , inSession: Panelist $\rightarrow$ string , name: ViewParticipant $\rightarrow$ string , address: ViewParticipant $\rightarrow$ Address , birthdate: ViewParticipant $\rightarrow$ Date , registrationCode: ViewParticipant $\rightarrow$ integer, participatesAs: ViewParticipant $\rightarrow$ {PCategory} , notify: ViewParticipant $\rightarrow$ string, conference: ActiveParticipant $\rightarrow$ string , confName: Conference $\rightarrow$ string , confDate: Conference $\rightarrow$ Date, participants: Conference $\rightarrow$ {Participant}, confName: ConfPaper $\rightarrow$ string , title: ConfPaper $\rightarrow$ string, registration: RentAVehicle $\rightarrow$ string , driver: RentAVehicle $\rightarrow$ string , cost: RentAVehicle $\rightarrow$ integer , registration: RentACar $\rightarrow$ string , driver: RentACar $\rightarrow$ string , cost: RentACar $\rightarrow$ integer , seater: RentACar $\rightarrow$ integer, registration: RentAMotorCycle $\rightarrow$ string , driver: RentAMotorCycle $\rightarrow$ string , cost: RentAMotorCycle $\rightarrow$ integer , cubic_capacity: RentAMotorCycle $\rightarrow$ integer, month: Date $\rightarrow$ integer , day: Date $\rightarrow$ integer, year: Date $\rightarrow$ integer, street: Address $\rightarrow$ string , zip_code: Address $\rightarrow$ string , city: Address $\rightarrow$ string , country: Address $\rightarrow$ string }

Figure 7: Class hierarchy $(EC, \prec_e, \Delta_e)$ of the extended schema ES.

The projection-instructions Projection(AuthorP), Projection(Panelist), Projection(Participant), Projection(Conference), and $\text{Projection}_P()$ where the later only consists in removing the attribute `birthdate` defined at class ViewParticipant: $P = \{(\texttt{birthdate}\text{: ViewParticipant} \rightarrow \text{Date})\}$, are successively applied to ES.

A projection-instruction also results in a migration of the referents from the deleted class to its supclasses. An optional query q in P may only select some of migrated referents. When no query q is precised, q is the query identity.

	$\text{Projection}_P(c)$
π_{i+1}	if $c \in C_0$ then $\forall k,\ c \prec_i c_k,\ \pi_{i+1}(c_k) = \pi_i(c_k) \cup \{\langle r, c_k\rangle \mid r \in \pi_i(c)\ and\ q(r) \in q(I_i)\}$ if $c \notin C_o$ then $\forall k,\ c \prec_i c_k,\ \pi_{i+1}(c_k) = \pi_i(c_k) \cup \{\langle r, c_k\rangle \mid \langle r, c\rangle \in \pi_i(c)\ and\ q(r) \in q(I_i)\}$

Example 4.3 When successively applying the projection-instructions of the expression:

[Projection(AuthorP), Projection(Panelist), Projection(Participant), Projection(Conference)]

on the instance, some migrations happen. The consequence of the instruction Projection(AuthorP) (resp. Projection(Panelist)), is a migration of the objects of the class AuthorP (resp. Panelist) to the class Participant. Note that the referents are not nested: the referent $\langle oid_4, \text{AuthorP}\rangle$ is migrated to the class Participant as $\langle oid_4, \text{Participant}\rangle$. The instruction Projection(Participant) causes a migration of all referents in class Participant to the class Person. The referents are still not nested: the referent $\langle oid_4, \text{Participant}\rangle$ is migrated to the class Person as $\langle oid_4, \text{Person}\rangle$.

The last instruction Projection(Conference) removes a class which has no supclass, it follows that the referents, which are not migrated to any class, are not accessible through the view.

ALG^V is the set of expressions (denoted *view-expressions*) $[E, P]$ where E is a extension-expression and P is a projection-expression. It is easy to see that a view-expression defines a view-transformation.

5 Conclusion

In this section, we give some conclusions and compare our proposal with existing view mechanisms. More work must be done on view-transformations in order to cover ojects views but they seem to arise as a good basic notion for an object view model. Perspectives in that sense conclude this section.

5.1 Issues

This paper introduces a data model for object views based on the notion of referent. A referent allows to address an object in one of the possibly several different classes it may belong to. It is a form of multiple instantiation. A database is populated with referents and their values. We characterized view-transformations as queries that map one view into another. We carred up with five simple algebraic operators that can transform a schema and its base in a view. The view is obtained from the source database using algebraic operators. Together these operators constitute an approach with two conceptual steps: an extension followed by a projection of a source base. With this approach, a view may be defined from one or more other views.

We modeled a view as a schema and base transformations. Our model has the following main features:

(i) the identification of an object in a target database (a view) always enables the retrieval of "components" via referents. This notion was introduced by C. Souza [dS95b] and is extended in the paper with the mechanism of complex referent construction.

(ii) since we have complex oids, we can (1) consider new objects (join on various classes of different kinds, see `ConfPaper`), and (2) identify objects that have the same values on their key attributes but different oids in the base (see the objects `Cluet`).

(iii) referents express the possibility of multi-instantiations.

(iv) transformations expressed using the five algebraic operators always lead to a valid schema and a base.

(v) the mechanism of transformation may be applied several times resulting in the possibility of building a view upon other views.

(vi) finally, a transformation is carried out in two steps: an extension and a projection at the schema and base levels.

5.2 Related work

Since there is a few work on formal data model for views, we compare hereafter our approach with the model of E. Rundensteiner and the one of C. Souza. This comparison is performed emphasizing the difference between the various organizations of the class hierarchies of the source schema and of the view, and the ways the objects are identified and stored.

In our approach, a schema and its instance are transformed in a target schema and base, the view, through a cascade of intermediate transformations. Note that the source classes are not forced to be in the target schema (e.g. when they are hidden they disappear). Virtual classes are classified in the target schema (e.g. while generalizing classes or removing classes). Objects can be removed or preserved with the same role or with a new one (the one of the newly created virtual class). Only necessary classes (type closure) and objects are retained. In that sense, the transformation is a *REAL restructuring* of the source schema and base, and can be considered as an optimization. Materializing these views would make it clear. In our framework, we cannot create information that is not derived and disable classical schema-database update. Note that the five algebraic operators (modulo renaming) enable the same kinds of change as the ones from [KR96] or COOL in COCOON [LS92].

In Rudensteiner[KR96], building a view extends a source schema-base, the *global* schema, adding the newly created virtual class in the same class hierarchy. Additional intermediate classes are also inserted for classification purpose. As opposed to our model, there is only an extension phase. You may hide properties in a class or a class but it always leads to new class creation(s). The source schema is not restructured in that sense. As a consequence, over time, a large number of classes and underlying object instances decreases the performances of the system and makes consistent view removal necessary [CTR96]. Regarding the objects, object slicing technique [KRR95] is used: a *conceptual object* (a kind of abstract object) is made up of several *objects of implementation*. This makes rather simple data updates. The drawback is to multiply the objects of implementation, duplicating them in all the subclasses of a class, while our view classes are populated by extension.

In Souza[dS95b], the source schema is transformed in a target schema, importing all the classes necessary to the derivation clause. The virtual classes are not classified but simply created in a flat hierarchy under the root class.

5.3 Perspectives

Since we have referents instead of oids, it becomes necessary to extend OQL to support the concept of referent. In particular, we need to talk about its content as addressed in Example 4.1. This is another way of saying that an object plays a *ROLE* in a context. Note that in [Aa94], OQL is specified to allow objects constructing with declarations alike

`select Couple(student:x.name,professor:z.name) where ...`

where the class `Couple` is assumed to be already created. Extension of OQL should cover this topic, creating "real" virtual classes.

In addition, a declarative language for view specifications in the style of r̃efSou95:PhD95 is required. It would include the possibilty of declaring a virtual class with its attributes and methods, as well its population by means of the extended OQL.

Since we have the notion of referent, we can go further and address the problem of querying the view. Suppose that S and S' are respectively a source and a view schema. For any query Q, if I and I' are respectively an instance of S and S', for all expression e of Q, there should exist an expression e', such that $e(I') = e'(I)$. We have to propose a translation t of expressions of queries in (extended) OQL, based on the algebraic definition of the view. The following query to retrieve a population illustrates this topic.

The query Q: **What are the active participants?** is expressible by the expression

$e \equiv \text{ActiveParticipant}(a), a =< p, \text{ActiveParticipant} >$ on the view instance.

$$
\begin{array}{rcl}
e & \equiv & \text{ActiveParticipant}(a), a =< p, \text{ActiveParticipant} > \\
 & \hookrightarrow & \text{Specialization}_{P_{AP}}(\text{ViewParticipant})(a), a =< p, \text{ActiveParticipant} > \\
 & \hookrightarrow & \text{ViewParticipant}(v), v =< p, \text{ViewParticipant} >, p =< u, \text{AuthorP} >, p.\text{presenter} = \text{true} \\
 & & \text{OR ViewParticipant}(v), v =< p, \text{ViewParticipant} >, p =< u, \text{Panelist} > \\
 & \hookrightarrow & \text{Join}_{P_{VP}}(\text{Participant})(v), v =< p, \text{ViewParticipant} >, p =< u, \text{AuthorP} >, p.\text{presenter} = \text{true} \\
 & & \text{OR Join}_{P_{VP}}(\text{Participant})(v), v =< p, \text{ViewParticipant} >, p =< u, \text{Panelist} > \\
 & \hookrightarrow & \text{Participant}(p), p =< u, \text{AuthorP} >, p.\text{presenter} = \text{true} \\
 & & \text{OR Participant}(p), p =< u, \text{Panelist} > \\
 & \equiv & t(e)
\end{array}
$$

We characterized view-transformations in Section 4.1. Our algebra defines view-transformations. Knowing what subclass of view-transformations these views capture would be of interest. This could be a matter of giving a constraint on the target schema w.r.t. the source schema. If the constraint is satisfied, there exists an expression in the algebra that allows the definition of this view. This formalizes the notion of *View-Derivation* pointed out in [dS95b].

Finally, we did some choices regarding the semantics of our five algebraic operators. They could be also extended to provide more flexibility to the view designers (for instance sophisticated mechanism for class hiding).

Acknowledgment Cassio Souza dos Santos is thanked for getting the authors valuable comments on this data model and for discussions on views and O_2Views.

References

[Aa94] T. Atwood and al. *The Object Database Standard: ODMG - 94*. Morgan Kaufmann, San Francisco, 1994.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD'91 Conference Proceedings, Int. Conf. on Management of Data*, San Francisco, California, March 1991. ACM Press.

[ADdS94] S. Abiteboul, C. Delobel, and C. Souza dos Santos. Virtual Schemas and Bases. In *Proceedings of the Intl. Conference on Extending Database Technology*, March 1994.

[AdS95] S. Abiteboul and C. Souza dos Santos. IQL(2): A Model with Ubiquitous Objects. In *Proceedings of the Intl. Workshop on Database Programming Languages - DBPL'95*, Gubio, Italy, September 1995. Springer Verlag.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison–Wesley, 1995.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD Symposium on the management of Data*, pages 159–173, Portland Oregon USA, June 1989.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System — The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

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

[CH80] A. Chandra and D. Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, Oct. 1980.

[CTR96] V. Crestana-Taube and E.A. Rundensteiner. Consistent View Removal in Transparent Schema Evolution Systems. *6th Int. Workshop on Research Issues on Data Engineering, Interoperability of Non-traditional Database Systems, RIDE'96,IEEE*, February 1996.

[dS95a] C. Souza dos Santos. Design and Implementation of Object–Oriented Views. In N. Revell and A.Min. Tjoa, editors, *Proc. of the 6th International Conference on Database and Expert Systems Applications*, number 978 in LNCS, London, England, September 1995. Springer Verlag.

[dS95b] C. Souza dos Santos. *Un Mécanisme de Vues pour les systèmes de Gestion de Bases de Données Objet*. PhD thesis, Université de Paris Sud - Centre d'Orsay, Paris, France, November 1995.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1992.

[Kos95] A. Kosky. *Transforming Databases with Recursive Data Structures.* PhD thesis, University of Pennsylvania, 1995.

[KR96] H.A. Kuno and E.A. Rundensteiner. The MultiView OODB View System: Design and Implementation. *Journal of Theory and Practice of Object Systems (TAPOS). Special Issue on Subjectivity in Object–Oriented Systems*, 1996.

[KRR95] H. Ra Kuno, Y-G. Ra, and E.A. Rundensteiner. Object-Slicing: A Flexible Object Representation and Its Evaluation. Technical Report CSE-TR-241-95, University of Michigan, Electrical Engineering and Computer Science Dept., University of Michigan, Ann Arbor, April 1995.

[KW89] M. Kifer and J. Wu. A Logic for Object-Oriented Logic Programming (Maier's O-Logic Revisited). In *Proc. ACM Symp. on Principles of Database Systems*, 1989.

[LRV88] C. Lécluse, P. Richard, and F. Vélez. O_2, an Object−Oriented data model. In *Proceeding ACM SIGMOD, Chicago IL*, June 1988.

[LS92] C. Laasch and M. H. Scholl. Generic Update Operations Keeping Object-Oriented Databases Consistent. In *Proc. 2nd GI-Workshop on Information Systems and Artificial Intelligence.* Springer, February 1992.

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

[MP96] R. Motschnig-Pitrik. Requirements and comparison of view mechanisms in object-oriented databases. *Information Systems. Special Issue on Foundations of Object-Oriented Database Systems*, 21(3):229–252, May 1996.

[Run92] E.A. Rundensteiner. Multi View: A Methodology for supporting Multiple View Schemata in Object–Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 187–198, Vancouver, Canada, August 1992. Morgan Kaufmann.

[SS91] M.E. Scholl and H.-J. Schek. Supporting views in object-oriented databases. *IEEE Database Engineering Bulletin. Special Issue on Foundations of Object-Oriented Database Systems*, 2(14):43–47, June 1991.

[Tec95] O2 Technology. *O2C Reference Manual, Version 4.6, Release September 1995.* O2 Technology, Versailles, France, September 1995.

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

Appendix

A.1 Motivating example: the source base `Conference_base`

The schema `Conference_schema` is populated as shown in Figure 8. Class `Person` is not populated whereas all the other classes are populated. Objects are pairs (**r**,[**val**]) where **val** is the value of all attributes and methods applicable at the object class.

Base *Conference_base*	
Class Name	Object and its value
Person	∅
AccompaniedGuest	(< $oidag_1$, AccompaniedGuest>, [Durand, < $oida_1$, Address>, < $oidd_{10}$, Date >, 1, 27]), (< $oidag_2$, AccompaniedGuest>, [Moeller, < $oida_3$, Address>, < $oidd_{11}$, Date >, 1 , 49]), (< $oidag_3$, AccompaniedGuest>, [Cluet, < $oida_5$, Address>, < $oidd_{12}$, Date >, 0 , 12])
Participant	(< $oidp_1$, Participant>, [Durand, < $oida_1$, Address>, < $oidd_4$, Date >, 1, {student}, 28, *message1*]), (< $oidp_2$, Participant>, [Simpson, < $oida_2$, Address>, < $oidd_5$, Date >, 2, {industrial}, 27, *message1*]), (< $oidp_3$, Participant>, [Neumann,< $oida_3$, Address>, < $oidd_6$, Date >, 3, {industrial}, 52, *message1*])
AuthorP	(< $oidp_4$, AuthorP>, [Abiteboul, < $oida_4$, Address>,< $oidd_1$, Date > , 4 , {academic}, Objects and Views, 1, 42, *message2*]), (< $oidp_5$, AuthorP>, [Cluet, < $oida_5$, Address> , < $oidd_2$, Date >, 5, {academic}, Correspondence and Translation for Heterogeneous Data, 1 , 37, *message2*]), (< $oidp_6$, AuthorP>, [Cluet, < $oida_5$, Address> , < $oidd_2$, Date >, 1, {industrial}, YAQO: Yet Another Query Optimizer, 0, 37, *message1*]), (< $oidp_7$, AuthorP>, [Milo, < $oida_6$, Address>, < $oidd_7$, Date >, 2, {academic}, YAQO: Yet Another Query Optimizer, 1, 37, *message2*])
Panelist	(< $oidp_8$, Panelist>, [Vianu, < $oida_7$, Address>, < $oidd_3$, Date >, 6, {academic}, Take a walk on the WEB side, 41, *message3*]), (< $oidp_9$, Panelist>, [Abiteboul, < $oida_4$, Address>, < $oidd_1$, Date > , 3 , {academic}, Semi-Structured Data , 42, *message3*])
Conference	(< $oidc_1$, Conference>, [ICDT, < $oidd_8$,Date >, { < $oidp_1$, Participant> , < $oidp_2$, Participant> , < $oidp_3$, Participant> , < $oidp_4$, AuthorP> , < $oidp_5$, AuthorP> , < $oidp_8$, Panelist> }]), (< $oidc_2$, Conference>, [VLDB, < $oidd_9$, Date >, { < $oidp_6$, AuthorP> , < $oidp_7$, AuthorP> , < $oidp_9$, Panelist> }])
RentACar	(< $oidrc_1$, RentACar>, [1, Abiteboul , 4, 300]), (< $oidrc_2$, RentACar>, [2, Vianu , 2, 220])
RentAMotorCycle	(< $oidrmc_1$, RentAMotorCycle>, [3, Durand , 400, 150]), (< $oidrmc_2$, RentAMotorCycle>, [4, Cluet , 125, 100])
Date	(< $oidd_1$, Date>, [04 , 20 , 1955]), (< $oidd_2$, Date>, [02 , 02 , 1960]), (< $oidd_3$, Date>, [05 , 10 , 1956]), (< $oidd_4$, Date>, [08 , 23 , 1969]), (< $oidd_5$, Date>, [02 , 29 , 1970]), (< $oidd_6$, Date>, [11 , 01 , 1945]), (< $oidd_7$, Date>, [04 , 06 , 1960]), (< $oidd_8$, Date>, [01 , 11 , 1997]), (< $oidd_9$, Date>, [08 , 25 , 1997]), (< $oidd_{10}$, Date>, [02 , 23 , 1970]), (< $oidd_{11}$, Date>, [01 , 02 , 1948]), (< $oidd_{12}$, Date>, [01 , 10 , 1985])
Address	(< $oida_1$, Address>, [Bd St Germain, 75007 , Paris, France]), (< $oida_2$, Address>, [Oxford Street, EC1V 0HB, London, England]), (< $oida_3$, Address>, [Kudamm, 14195, Berlin, Germany]), (< $oida_4$, Address>, [Av. Duchef, 75 015, Paris, France]), (< $oida_5$, Address>, [Rue Presquechef, 92 300 , Sympa-LaVille, France]), (< $oida_6$, Address>, [Ramat Aviv, 69978 , TelAviv, Israel]) , (< $oida_7$, Address>, [Bay Street, 92093-0114, San Diego, California US])

Figure 8: The base `Conference_base` of the schema `Conference_schema`.

A.2 Motivating example: Defining a View

A.2.1 Virtual Schema

Following Definition 2.1, the following table gives the class hierarchy $(VC, \prec_V, \Delta_V)$ of the view schema S', called `ViewConference schema`, issued from the view specification of Section3.1.

VC	{ Person, AccompaniedGuest, YoungGuest, ViewParticipant, ActiveParticipant, ConfPaper, RentAVehicle, RentACar, RentAMotorCycle, Date, Address, PCategory }
$\prec_V$	AccompaniedGuest $\prec_V$ Person , YoungGuest $\prec_V$ AccompaniedGuest ActiveParticipant $\prec_V$ ViewParticipant RentACar $\prec_V$ RentAVehicle , RentAMotorCycle $\prec_V$ RentAVehicle
Δ_V	{ name: Person $\rightarrow$ string , address: Person $\rightarrow$ Address , birthdate: Person $\rightarrow$ Date , age: Person $\rightarrow$ integer , touristPackage: AccompaniedGuest $\rightarrow$ boolean , name: ViewParticipant $\rightarrow$ string, address: ViewParticipant $\rightarrow$ Address, registrationCode: ViewParticipant $\rightarrow$ integer, participatesAs: ViewParticipant $\rightarrow$ {PCategory}, notify: ViewParticipant $\rightarrow$ string, conference: ActiveParticipant $\rightarrow$ string, confName: ConfPaper $\rightarrow$ string , title: ConfPaper $\rightarrow$ string, registration: RentAVehicle $\rightarrow$ string , driver: RentAVehicle $\rightarrow$ string , cost: RentAVehicle $\rightarrow$ integer , registration: RentACar $\rightarrow$ string , driver: RentACar $\rightarrow$ string , seater: RentACar $\rightarrow$ integer, cost: RentACar $\rightarrow$ integer , registration: RentAMotorCycle $\rightarrow$ string , driver: RentAMotorCycle $\rightarrow$ string , cubic_capacity: RentAMotorCycle $\rightarrow$ integer, cost: RentAMotorCycle $\rightarrow$ integer, month: Date $\rightarrow$ integer , day: Date $\rightarrow$ integer, year: Date $\rightarrow$ integer, street: Address $\rightarrow$ string , zip_code: Address $\rightarrow$ string , city: Address $\rightarrow$ string , country: Address $\rightarrow$ string }

A.2.2 Virtual base

The population I' of the view schema S', `ViewConference_base`, is detailed in the following table.

Base *ViewConference_base*	
View Class Name	Virtual Object and its value
Person	($< oidp_1$, Person>, [Durand, $< oida_1$, Address>, $< oidd_4$, Date >, 28]), ($< oidp_2$, Person>, [Simpson, $< oida_2$, Address>, $< oidd_5$, Date >, 27]), ($< oidp_3$, Person>, [Neumann,$< oida_3$, Address>, $< oidd_6$, Date >, 52]), ($< oidp_4$, Person>, [Abiteboul, $< oida_4$, Address>, $< oidd_1$, Date >, 42]), ($< oidp_5$, Person>, [Cluet, $< oida_5$, Address> , $< oidd_2$, Date >, 37]), ($< oidp_6$, Person>, [Cluet, $< oida_5$, Address>, $< oidd_2$, Date >, 37]), ($< oidp_7$, Person>, [Milo, $< oida_6$, Address>, $< oidd_7$, Date >, 37]), ($< oidp_8$, Person>, [Vianu, $< oida_7$, Address>, $< oidd_3$, Date >, 41]), ($< oidp_9$, Person>, [Abiteboul, $< oida_4$, Address>, $< oidd_1$, Date > , 42])
AccompaniedGuest	($< oidag_1$, AccompaniedGuest>, [Durand, $< oida_1$, Address>, $< oidd_{10}$, Date >, 1, 27]) , ($< oidag_2$, AccompaniedGuest>, [Moeller, $< oida_3$, Address>, $< oidd_{11}$, Date >, 1 , 49])
YoungGuest	($< oidag_3$, YoungGuest>, [Cluet, $< oida_5$, Address>, $< oidd_{12}$, Date >, 0 , 12])
ViewParticipant	($<< oidp_1$, Participant>, ViewParticipant>, [Durand, $< oida_1$, Address>, 1, {student}, *message1*]) , ($<< oidp_2$, Participant>, ViewParticipant>, [Simpson, $< oida_2$, Address>, 2, {industrial}, *message1*]) , ($<< oidp_3$, Participant>, ViewParticipant>, [Neumann,$< oida_3$, Address>, 3, {industrial}, *message1*]) , ($<< oidp_6$, AuthorP>, ViewParticipant>, [Cluet, $< oida_5$, Address>, 1, {industrial}, *message1*])
ActiveParticipant	($<< oidp_4$, AuthorP>, ActiveParticipant>, [Abiteboul, $< oida_4$, Address>, 4 , {academic}, *message2* , ICDT]) , ($<< oidp_5$, AuthorP>, ActiveParticipant>, [Cluet, $< oida_5$, Address> , 5, {academic}, *message2* , ICDT]) , ($<< oidp_7$, AuthorP>, ActiveParticipant>, [Milo, $< oida_6$, Address>, 2, {academic}, *message2* , VLDB]) , ($<< oidp_8$, Panelist>, ActiveParticipant>, [Vianu, $< oida_7$, Address>, 6, {academic}, *message3* , VLDB]) ($<< oidp_9$, Panelist >, ActiveParticipant>, [Abiteboul, $< oida_4$, Address>, 3 , {academic}, *message3* , VLDB])
ConfPaper	($<< oidc_1$, Conference> $\wedge$ $< oidp_4$, AuthorP>> , ConfPaper>, [ICDT, Objects and Views]) ($<< oidc_1$, Conference> $\wedge$ $< oidp_5$, AuthorP>>, ConfPaper>, [ICDT, Correspondence and Translation for Heterogeneous Data]) ($<< oidc_2$, Conference> $\wedge$ $< oidp_7$, AuthorP>>, ConfPaper>, [VLDB, YAQO:Yet Another Query Optimizer])
RentAVehicle	$\emptyset$
RentACar	($< oidrc_1$, RentACar>, [1, Abiteboul, 4, 300]) , ($< oidrc_2$, RentACar>, [2, Vianu, 2, 220])
RentAMotorCycle	($< oidrmc_1$, RentAMotorCycle>, [3, Durand , 400, 150]) , ($< oidrmc_2$, RentAMotorCycle>, [4, Cluet, 125, 100])
Date	($< oidd_8$, Date>, [01 , 11 , 1997]), ($< oidd_9$, Date>, [08 , 25 , 1997]), ($< oidd_{10}$, Date>, [02 , 23 , 1970]), ($< oidd_{11}$, Date>, [01 , 02 , 1948]), ($< oidd_{12}$, Date>, [01 , 10 , 1985])
Address	($< oida_1$, Address>, [Bd St Germain, 75007 , Paris, France]) ($< oida_2$, Address>, [Oxford Street, EC1V 0HB, London, England]) , ($< oida_3$, Address>, [Kudamm, 14195, Berlin, Germany]) , ($< oida_4$, Address>, [Av. Duchef, 75 015, Paris, France]) , ($< oida_5$, Address>, [Rue Presquechef, 92 300 , Sympa-LaVille, France]) , ($< oida_6$, Address>, [Ramat Aviv, 69978 , TelAviv, Israel]) , ($< oida_7$, Address>, [Bay Street, 92093-0114, San Diego, California US])