Object Views and Database Restructuring (extended abstract)

Zoé Lacroix*and Claude Delobel†and Philippe Brèche‡

Abstract

We present a formal data model for views in Object DataBase Systems (ODBS) which is a continuation of the work presented in [AK89, AB91, dSDA94, dS95a]. The main contribution of this paper is the formalization of views as a transformation mechanism for databases using a generalization of *referent* proposed in [dS95b]. We define an IQL-like language adapted to manipulate referents. 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 four object algebraic operators that specify both the virtual schema and its corresponding virtual instance. This simple algebra can express all the view operators proposed in [KR96, LS92, dS95b, KRR95, CTR96, KR96].

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

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 operators, as well as in the object database domain [AB91, Ber91, TYH+91, SS91, Run92, dSDA94], have been put forward but only very few of them are implemented. In the relational context, commercial databases only provide view mechanisms that are restrictive whereas none of the commercial object databases provides user with any such mechanism. 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.

It seems first natural to consider a view as a query, such as that used in the relational world [AB91, KKS92]. A language defining views in the object model, has to deal with schemas as well as instances. In our framework, a source schema-base and its view have the following 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,

*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.

‡Projet Verso– I.N.R.I.A. Rocquencourt BP. 105, 78 153 Le Chesnay Cedex, France– tel: 33 (1) 39 63 56 79 – fax: 33 (1) 39 63 56 74 – e-mail: Philippe.Breche@inria.fr

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

class name), which enables us to relax the commonly adopted constraint of one object belonging 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.

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 one to define views from other view(s). We propose an IQL-like language in order to manipulate referents. We provide our model with four basic algebraic operators that express globally transformations on the schema and on the data. The operators are *projection*, *natural-join*, *join-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 by combining the 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 given in Section 3 which is further developed in the ensuing sections. We define RQL our query language in Section 4. Section 5 introduces the basic object algebraic operators used (eventually combined) to issue our views. Examples are given to illustrate the operators whose semantics is formally given in the appendix. Section 6 concludes on issues, and related 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 class values are populated. 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 class $\{c'\}$. Let n be the name of a method or an attribute. A *definition* of n is a pair $(n, c \rightarrow \gamma(c'))$ where $c \rightarrow \gamma(c')$ is a signature. A method or an attribute n is *defined* at c if there exists a definition of n 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 n, and each $c \in C$ there is at most one definition of n at c. A method or an attribute n is *overloaded* if there exists at least two definitions of n at two different classes along an inheritance path. A method or an attribute n 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 $c'_1, c'_2 \in C \cup \mathbf{V}$, 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

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

and M, is a well-formed hierarchy. □

2.2 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 $\mathbf{O}=\{o_1, o_2, \ldots\}$ 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). A similar function which encodes oids, instead of referents, is defined in [KW89, KKS92, dS95b]. Contrary to [KW89, KKS92, dS95b], a new identifier may occur in several 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 of complex identifiers and the set $\mathbf{R}$ of all referents are simultaneously defined by the following grammar. For all $i \in \mathbf{O}$ and $c \in \mathbf{C}$, the set of complex identifiers and $\mathbf{R}$ are respectively the sets of o and r such that $o ::= r \mid o \wedge o$ and $r ::= \langle o, c\rangle \mid \langle i, c\rangle$. Finally the set $\hat{\mathbf{O}}$ of all identifiers is the union of $\mathbf{O}$ and the set of complex identifiers. □

We do not consider complex identifiers or referents defined with constants of $\mathbf{D}$. A referent $\langle o, c\rangle$ is *atomic* if o is an atomic oid, otherwise, it is *nested*. A complex oid $\langle o_1, c_1\rangle \wedge \ldots \wedge \langle o_n, c_n\rangle$ is *flat* if $o_1, \ldots, o_n$ are atomic. The equality of oids respects the following rules: $\forall o, o', o'' \in \hat{\mathbf{O}}$ $o \wedge o = o$, $o \wedge o' = o' \wedge o$ and if $o = o'$ then $o \wedge o'' = o' \wedge o''$.

In this new model, referents are not mutable and identify objects just like object identifiers 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. As in [dS95b], the classes are populated with 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 $\nu(v) \cap \nu(v') = \emptyset$.
□

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

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

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:

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 $\langle o, c\rangle \in \pi(c)$ for some $c \prec^* c_1$ and $\langle o, c'\rangle \in \pi(c')$ 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 (π, μ, ν) 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 $S = (A, M, C, \prec, \Delta)$. 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.

3 Motivating example

We present in this section the process of creating a schema and a base. From now on, they are denoted as S for *source* schema and as I for *source* base. The source schema (and later on the view construction) is taken from the example in [MP96]. Roughly, a model is specified for a Conference. It records the participants of the conference are classified in different groups such as authors or 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. The schema `Conference_schema` corresponding to the previous specification and its base `Conference_base` are given in Figures 3 and 4 of the Appendix A.1. Each class is populated with atomic referents. According to Definition 2.1, the source schema S is a tuple $(A, M, C, \prec, \Delta)$. The class hierarchy is given in Figure 1, where an arrow from class c_1 to class c_2 denotes $c_1 \prec c_2$. The attributes names are in $A = \{$ name, address, birthdate, touristPackage, registrationCode, participatesAs, notify, title, presenter, inSession, confName, confDate, participants, registration, driver, seater, cubic_capacity, month, day, year, street, zip_code, city, country$\}$, whereas the method names are in $M = \{$ age, notify, cost$\}$.

A **view** is specified using S and I to encompass the following information:

- it imports class `Person`, `AccompaniedGuest`, `RentACar`, `RentAVehicle`[4] and all the classes necessary to maintain type closure in the view (namely, classes `Address` and `Date`).
- class `YoungGuest` is derived from class `AccompaniedGuest`. The extent of `YoungGuest` contains the accompanied guests whose age is less than 18 who become instances of 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`, and has an additional method `conference():string`. The extent of `ActiveParticipant` is the union of all panelists and 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 source base.

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

- 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.

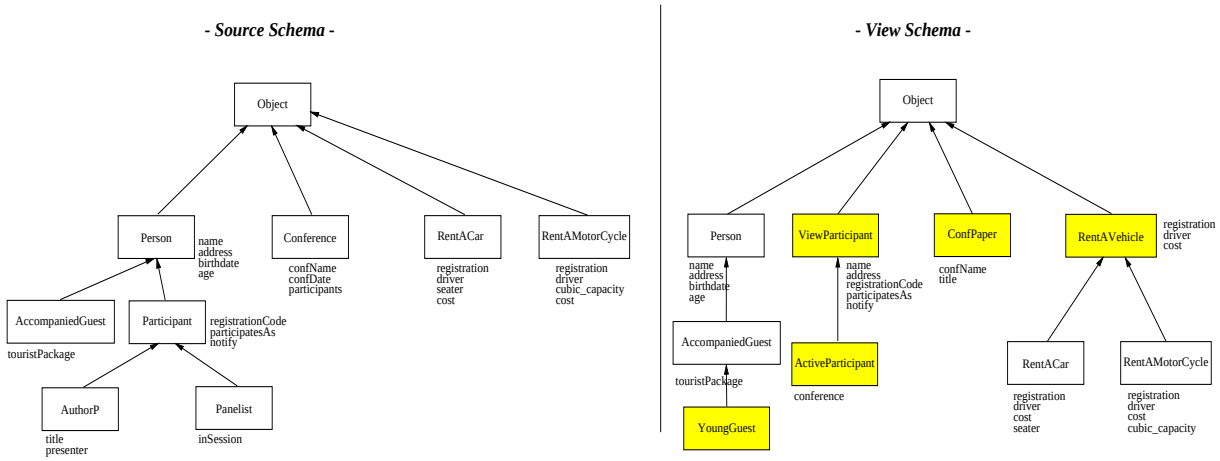

Figure 1: Class hierarchies of the source and view schemas.

Figure 1 represents the class hierarchies of the source schema S and of the view schema S'. Classes `Address` and `Date` are omitted. The corresponding virtual base is given in Figure 5 of the Appendix A.2.

4 RQL: a query language to manipulate referents

In our model, referents and classes have a type. The syntax and the semantics of the types is defined as follows with our notations. The set of type expressions, denoted $\mathbf{T}$, is given by the following abstract grammar. For all $v \in \mathbf{V}$, for all $c \in \mathbf{C}$, the set $\mathbf{T}$ is the set of τ such that: $\tau = \emptyset \mid v \mid c \mid \{\tau\}$. The semantics of types is given with respect to a referent assignment π and a constant assignment ν. For all $v \in \mathbf{V}$ and $c \in \mathbf{C}$, $[\![\emptyset]\!]_\pi = \emptyset$; $[\![v]\!]_\nu = \nu(v)$; $[\![c]\!]_\pi = \pi^*(c)$ and finite setting is standard. The subtyping relationship is defined from the subclassing relationship the usual way.

We express now more formally the grammar of the referent query language. It is a finite set of rules over S, where terms, literals, and rules are defined as follows. The constants of the value classes are omitted for simplification in the variables. We suppose that for each type τ in $\mathbf{T}$, there exists a countable number of variables of type τ.

Referent-terms The *referent-terms* are defined as follows ($n \geq 1$).

- $\langle o, c \rangle$, where $o \in \hat{\mathbf{O}}$, is a referent-term of type c.
- x, a variable of type c, is a referent-term of type c.
- $\langle t, c \rangle$, where t is a referent-term, is a referent-term of type c.
- $\langle t_1 \wedge \ldots \wedge t_n, c \rangle$, where $t_1, \ldots, t_n$ are referent-terms, is a referent term of type c.

Terms The *terms* are defined as follows ($n \geq 1$).

- t, a referent-term of type c, is a term of type c.
- $t.p$, where t is a referent-term of type c, and $(p, c \rightarrow \gamma(c')) \in \Delta$, is a term of type $\gamma(c')$.
- $\{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ are terms of type τ, is a term of type $\{\tau\}$.

Literals The *literals* are defined as follows.

- $c(t)$, where t is a referent-term of type c, is a literal.
- $t_1 = t_2$ and $t_1 \neq t_2$, where t_1 and t_2 are terms of the same type, are literals.
- $t_1 \in t_2$ and $t_1 \notin t_2$, where t_1 and t_2 are respectively terms of type τ and of type $\{\tau\}$, are literals.

Rules A *rule* is an expression of the form $L \longleftarrow L_1, \ldots, L_k$ $(k \geq 0)$, where $L, L_1, \ldots, L_k$ are literals and:

1. L is a literal of the form $\{t\}$ where t is a term, $c(t)$ where t is a referent-term of type c, or a well-typed literal of the form $x.n = y$, where x and y are variables.
2. $\forall i$, we relax in L_i the constraint over the type in the classical way with respect to the subtyping relationship.

Rules of the form $\{t\} \longleftarrow L_1, \ldots, L_k$ are used to express queries. Rules of the form $c(t) \longleftarrow L_1, \ldots, L_k$ are used to express constraint on the population of the virtual class c. Rules of the form $x.n = y \longleftarrow L_1, \ldots, L_k$ are used to define virtual attributes or methods and are given together with a definition $(n, c \rightarrow \gamma(c'))$, which precises the type c of variable x and the type $\gamma(c')$ of variable y. We did not include among the terms the constants for simplification. They can be added easily with their classical operators.

5 An algebra to express view-transformations

In [AK89], Abiteboul and Kannelakis introduce the notion of *projection* to express db-transformations, later completed by the notion of *extension* in [dS95b]. Conceptually, a db-transformation $\subseteq inst(S) \times inst(S')$ is performed in 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'. Since a db-transformation is seen as a query, it is a subset of $inst(S) \times inst(S')$ where S and S' are two given schemas. The class of view-transformations is a subclass of db-transformations, where the schema S' itself, is derived from S.

The view-transformations are defined with an algebra of four operators *join-specialization*, *join*, *generalization* and *projection*. The first three express the extension of the schema and base, the last one expresses the projection of the schema and base. In this section we give the syntax and an informal description of the semantics of the algebra illustrated with our example. The semantics of the algebra in formally given in Section A.3 of the appendix.

5.1 Extension

An *extension-expression* is a list $[e_1, \ldots, e_m]$ of *extension-instructions* as shown in 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 RQL query), definitions of attributes, and/or definitions of methods.

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

Definition 5.1 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 $(n_i, c' \rightarrow \gamma(c_{j_i}))$ and $q, q_1, \ldots, q_m$ are RQL expressions. 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. The *extended schema* ES and its *extended instance* EI are defined by the extension-expression $E = [e_1, \ldots, e_m]$, where $e_1, \ldots, e_m$ are extension-instructions, from the source schema S and instance I such that $(S_0, I_0) = (S, I)$ and $\forall i, 1 \leq i \leq m$, $(S_i, I_i) = e_i(S_{i-1}, I_{i-1})$, and $(ES, EI) = (S_m, I_m)$.

Example 5.1 The extended schema pictured out in Figure 2 is obtained by the following extension expression:
[YoungGuest $\Leftarrow$ Join-Specialization$_{P_Y}$(AccompaniedGuest), ViewParticipant $\Leftarrow$ Join(Participant),
ActiveParticipant $\Leftarrow$ Join-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 the queries are expressed in RQL defined in Section 4.

$P_Y =$

$[(\text{YoungGuest}(\langle o, \text{YoungGuest}\rangle) \longleftarrow \text{AccompaniedGuest}(g), g = \langle o, \text{AccompaniedGuest}\rangle, g.\text{age} \leq 18)]$

$P_P =$

$[(\text{ConfPaper}(cp) \longleftarrow cp = \langle\langle oc, \text{Conference}\rangle \wedge \langle op, \text{AuthorP}\rangle, \text{ConfPaper}\rangle,$
$\langle op, \text{AuthorP}\rangle \in \langle oc, \text{Conference}\rangle.\text{participants}, \langle op, \text{AuthorP}\rangle.\text{presenter} = true)]$

$P_{AP} =$

$[(\text{ActiveParticipant}(\langle\langle o, \text{AuthorP}\rangle, \text{ActiveParticipant}\rangle) \longleftarrow \text{ViewParticipant}(a),$
$a = \langle\langle o, \text{AuthorP}\rangle, \text{ViewParticipant}\rangle, \langle o, \text{AuthorP}\rangle.\text{presenter} = true;$
$\text{ActiveParticipant}(\langle\langle o, \text{Panelist}\rangle, \text{ActiveParticipant}\rangle) \longleftarrow \text{ViewParticipant}(a),$
$a = \langle\langle o, \text{Panelist}\rangle, \text{ViewParticipant}\rangle),$
$(\text{conference}, \text{ActiveParticipant} \rightarrow string),$
$(a.\text{conference} = s \longleftarrow \text{Conference}(c), a = \langle p, \text{ActiveParticipant}\rangle,$
$p \in c.\text{participants}, s = c.\text{confName})]$

The following sections give the informal definition of the operators and detail how we come up with the new classes (virtual classes) in the view, extending and projecting the source schema (see Figure 1) and base (see Figure 4 of the Appendix A.1).

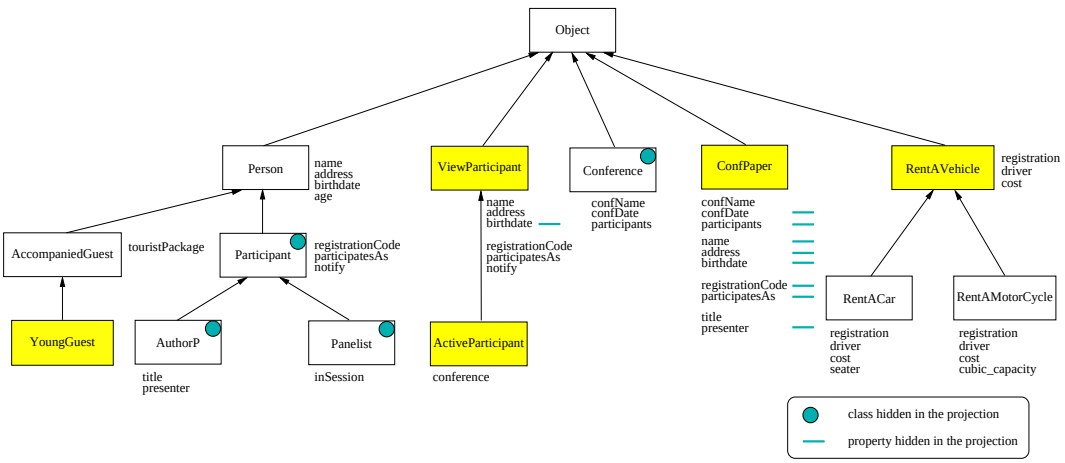

Figure 2: Class hierarchy of the extended schema ES.

5.1.1 Join-specialization

A virtual class may be the *join-specialization* of classes which have the same key attributes (that identify an object, similarly to relational) and methods[5]. It is a subclass of all classes that it specializes and it is populated with nested referents. A restriction on the population and/or new attributes or methods are possible.

The view issued from the source defines class `YoungGuest` as a virtual subclass of class `AccompaniedGuest`, populated with the guests younger than 18. The class `YoungGuest` is populated by querying the source base: the referents $\langle oidag$, AccompaniedGuest$\rangle$ in `AccompaniedGuest` whose age is smaller than 18 are selected and moved as $\langle\langle oidag$, AccompaniedGuest$\rangle$, YoungGuest $\rangle$ to the class `YoungGuest`. In the view, these objects are no longer in the population of `AccompaniedGuest`, but in its extension.

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

In case of a join-specialization c of n classes c_1, ..., c_n, a new referent $\langle\langle o_1, c_1\rangle \wedge \ldots \wedge \langle o_n, c_n\rangle, c\rangle$ refers to an object identifying n objects with n different oids that have the same values for their key attributes and methods. In the join-specialized class, these objects are identified with the new complex oid $\langle o_1, c_1\rangle \wedge \ldots \wedge \langle o_n, c_n\rangle$. A join-specialization does not consist in creating new objects, but in identifying objects with new oids.

5.1.2 Join

The *join* consists in really creating new objects (*imaginary* objects in [AB91]). A virtual class which is a join of classes is a subclass of the root class (`Object` in Figure 2) of the class hierarchy. The virtual class `ConfPaper` is populated with new objects $\langle\langle oidc$, Conference$\rangle \wedge \langle oidp$, AuthorP$\rangle$, ConfPaper $\rangle$ defined from objects of classes `AuthorP` and `Conference`. The complex oid $\langle oidc$, Conference$\rangle \wedge \langle oidp$, AuthorP$\rangle$ is a new oid which is obtained by applying a function to its value (*objectification*) as in [AB91], but to the referents it is defined from as in [dS95b]. The new class `ConfPaper` cannot be compared with the classes `AuthorP` or `Conference`.

A join may be also used to define a class which is expected to be a subclass of the root class in the view-schema. In the example, the class `ViewParticipant` is defined as a join of class `Participant` and populated with referents of the form $\langle\langle oidp$, Participant$\rangle$, ViewParticipant$\rangle$ as well as $\langle\langle oidp$, Panelist$\rangle$, ViewParticipant$\rangle$ or $\langle\langle oidp$, AuthorP$\rangle$, ViewParticipant$\rangle$.

5.1.3 Generalization

A *generalization* defines a virtual class as a supclass of the source classes it generalizes and as a subclass of all common supclasses (or of `Object`, when there is any). 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 in the following way. All definitions of common attributes and methods (behavior generalization) are also defined at the virtual class `RentAVehicle`.

5.2 Projection

Intuitively, the projection step consists in hiding information: hiding classes, attributes, and/or methods. A projection expression is a list of *projection-instructions* of the form Projection$_P(c)$, where P is a list of parameters $[q, d_1, \ldots, d_m]$, and $c \in EC$. A projection-instruction results in a migration of the referents from the deleted class to all its supclasses. An optional query q in P may only select some of migrated referents. When no query q is given, all objects of the hidden class are migrated to all the supclasses. All definitions where class c occurs as well as all definitions $d_1, \ldots, d_m$ in P are removed from Δ. For each definition of the form $(n, c \rightarrow \gamma(c'))$ $(c' \neq c)$, for each c'' subclass of c, if there is no definition of n applicable at c'', then $(n, c'' \rightarrow \gamma(c'))$ is added to Δ. Regarding this topic, other strategies and models for schema evolution are discussed in [BW95, Abi87].

[5] Other common attributes and methods must be renamed.

The view-schema given in Figure 1 is the result of the successive projections of classes AuthorP, Panelist, Participant, Conference on the extended-schema ES of Figure 2. The attribute definition (`birthdate`: View-Participant $\rightarrow$ Date) is removed from Δ. The instruction Projection(AuthorP) (resp. Projection(Panelist)), consists of a migration of the objects in class AuthorP (resp. Panelist) to class Participant. An atomic referent $\langle oid, \text{AuthorP}\rangle$ is migrated to the class Participant as a nested referent $\langle\langle oid, \text{AuthorP}\rangle, \text{Participant}\rangle$. The instruction Projection(Participant) causes a migration of all referents in class Participant to the class Person. A nested referent $\langle\langle oid_4, \text{AuthorP}\rangle, \text{Participant}\rangle$ is migrated to the class Person as $\langle\langle oid_4, \text{AuthorP}\rangle, \text{Person}\rangle$.

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.

6 Conclusion

Issues: This paper introduces a formal data model for object views which is a continuation of the work presented in [AK89, AB91, dSDA94, dS95a]. As in [dS95a], it is 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. A database is populated with referents and their values. We generalize the notion of identifier to explicitly express the link between a virtual object and the source object(s) it derives from. A source object is represented by an atomic referent $\langle o, c\rangle$ (o is an atomic oid, c a source class), whereas a virtual object is represented by a nested referent $\langle o', c'\rangle$ where o' is a complex oid (a source referent or a conjunction of source referents) and c' a virtual class. The view is obtained from the source database using four simple algebraic operators which define the schema as well as the instance (using RQL). Together these operators constitute an approach with two conceptual steps: an extension followed by a projection of a source base [AK89, dSDA94, dS95a]. After a projection, 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 disallow typical schema-database updates. Note that the four algebraic operators (modulo renaming) enable the same kinds of change as the ones from [KR96] or COOL in COCOON [LS92]. Our model has the following main features:

1. 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 nested referent construction.

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

3. Referents enable multi-instantiation.

4. An algebra which really structures the view, is completed by

5. a query language, RQL, which allows to adress referents populating the virtual classes.

6. The mechanism of transformation may be applied several times resulting in the possibility of building a view upon other views.

Related work 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. One 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 data updates easier. 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. In O_2Views the virtual classes are not classified but simply created in a flat hierarchy under the root class. In the model, the structuration of the target schema is not explicitly part of the view definition mecanism [dS95b].

Perspectives: We can go further and address the problem of querying the view. The naive evaluation of a query on the view consists in first defining the whole view, and second, in evaluating the query on the resulting view. We can do better. 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 on the view, there should exist an *equivalent* expression e on the source, that is such that $e(I) = e'(I')$. In order to optimize the evaluation, we have to propose a translation t of expressions of queries in RQL, based on the algebraic definition of the view. This translation can be formalized and results in only defining the part of the view that is necessary to evaluate the query. For example, the expression $\{a\} \leftarrow \text{ActiveParticipant}(a)$, of the query **"What are the active participants?"** which can be evaluated without defining classes `YougGuest` or `ConfPaper`, is translated as follows.

$$
\begin{array}{ll}
 & \{a\} \leftarrow \text{ActiveParticipant}(a) \\
\hookrightarrow & \{\langle p, \text{ActiveParticipant}\rangle\} \leftarrow \text{Join-Specialization}_{P_{AP}}(\text{ViewParticipant})(\langle p, \text{ActiveParticipant}\rangle) \\
\hookrightarrow & \{\langle\langle u, \text{AuthorP}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{ViewParticipant}(\langle\langle u, \text{AuthorP}\rangle, \text{ViewParticipant}\rangle), \\
 & \qquad\qquad \langle u, \text{AuthorP}\rangle.\text{presenter} = \text{true} \\
 & \{\langle\langle u, \text{Panelist}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{ViewParticipant}(\langle\langle u, \text{Panelist}\rangle, \text{ViewParticipant}\rangle) \\
\hookrightarrow & \{\langle\langle u, \text{AuthorP}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{Join}_{P_{VP}}(\text{Participant})(\langle\langle u, \text{AuthorP}\rangle, \text{ViewParticipant}\rangle), \\
 & \qquad\qquad \langle u, \text{AuthorP}\rangle.\text{presenter} = \text{true} \\
 & \{\langle\langle u, \text{Panelist}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{Join}_{P_{VP}}(\text{Participant})(\langle\langle u, \text{Panelist}\rangle, \text{ViewParticipant}\rangle) \\
\hookrightarrow & \{\langle\langle u, \text{AuthorP}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{Participant}(\langle u, \text{AuthorP}\rangle), \langle u, \text{AuthorP}\rangle.\text{presenter} = \text{true} \\
 & \{\langle\langle u, \text{Panelist}\rangle, \text{ActiveParticipant}\rangle\} \leftarrow \text{Participant}(\langle u, \text{Panelist}\rangle)
\end{array}
$$

The use of referents to populate classes enables the same object to be in several different classes, and then, to be seen several times in the same extension. Each use of Join-Specialization or Projection may dramatically increase the number of duplicates of the same object in the extension of a class. An object has different referents depending of the class it populates and may also have different oids. For instance $\langle o, c_1\rangle, \langle o, c_2\rangle$ and $\langle\langle o, c_2\rangle, c'\rangle$, where o is an atomic oid, denote the same object, populating two source classes c_1 and c_2, and a virtual class c'. If these three classes have a common supclass c_0, then there are three different occurrences of the object in the extent of c_0. Regarding the optimatization, those duplicates must be eliminated. An algorithm using definition of oid-equivalence should care of it, deleting those objects that are found several times in extensions.

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].

Note that in [Aa94], OQL is specified to allow objects constructing with declarations alike

```
select ConfPaper(confName:x.confName,title:z.title) from ... where ...
```

where the class `ConfPaper` is assumed to be already created. Extension of OQL should cover this topic, creating "real" virtual classes. This would require a merge of our algebra and RQL.

Finally, we did some choices regarding the semantics of our four 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. Peter Buneman, Val Tannen and Susan Davidson are also thanked for advices.

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.

[Abi87] S. Abiteboul. IFO: A Formal Semantic Database Model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[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.

[BW95] P. Brèche and M. Wörner. How to remove a class in an object database system. In *In Proceedings of the 2nd International Conference on Applications of Databases, ADB '95*, San José, California, December 1995.

[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.

[dSDA94] C. Souza dos Santos, C. Delobel, and S. Abiteboul. Virtual Schemas and Bases. In *Proceedings of the International Conference on Extending Database Technology*, March 1994.

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

[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 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 schema and its base `Conference_base`

Figure 3 presents the schema `Conference_schema` specified following the O_2 terminology [Tec95]. The corresponding class hierarchy, $S = (C, \prec, \Delta)$, is given in Figure **??**. 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 $\}$.

In addition, 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` in `RentAMotorCycle` take their values in the domains {`2,4`} and {`125,400,600`} (resp.). The schema `Conference_schema` is populated as shown in Figure 4. Class `Person` is not populated. Objects are pairs (**r**,[**val**]) where **val** is the value of all attributes and methods applicable at the object class. The first column corresponds to an abreviation of the class names.

```
        /* 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 3: The schema Conference_schema.

Base *Conference_base*	
Class	Object and its value
P	Ø
AG	(⟨$oidag_1$, 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])
P	(⟨$oidp_1$, 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*])
A	(⟨$oidp_4$, 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*])
PN	(⟨$oidp_8$, 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*])
C	(⟨$oidc_1$, 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⟩ }])
RAC	(⟨$oidrc_1$, RentACar⟩, [1, Abiteboul , 4, 300]), (⟨$oidrc_2$, RentACar⟩, [2, Vianu , 2, 220])
RAMC	(⟨$oidrmc_1$, RentAMotorCycle⟩, [3, Durand , 400, 150]), (⟨$oidrmc_2$, RentAMotorCycle⟩, [4, Cluet , 125, 100])
D	(⟨$oidd_1$, 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])
A	(⟨$oida_1$, 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 4: The base `Conference_base` of the schema `Conference_schema`.

A.2 Motivating example: defining a view

The instance I' of the view schema S', `ViewConference_base`, is detailed in Figure 5. The first column corresponds to an abreviation of the class names.

Base *ViewConference_base*	
Class	Virtual Object and its value
P	(⟨$oidp_1$, Participant⟩, Person⟩, [Durand, ⟨$oida_1$, Address⟩, ⟨$oidd_4$, Date ⟩, 28]), (⟨$oidp_2$, Participant⟩, Person⟩, [Simpson, ⟨$oida_2$, Address⟩, ⟨$oidd_5$, Date ⟩, 27]), (⟨$oidp_3$, Participant⟩, Person⟩, [Neumann,⟨$oida_3$, Address⟩, ⟨$oidd_6$, Date ⟩, 52]), (⟨$oidp_4$, AuthorP⟩, Person⟩, [Abiteboul, ⟨$oida_4$, Address⟩, ⟨$oidd_1$, Date ⟩, 42]), (⟨$oidp_5$, AuthorP⟩, Person⟩, [Cluet, ⟨$oida_5$, Address⟩ , ⟨$oidd_2$, Date ⟩, 37]), (⟨$oidp_6$, AuthorP⟩, Person⟩, [Cluet, ⟨$oida_5$, Address⟩, ⟨$oidd_2$, Date ⟩, 37]), (⟨$oidp_7$, AuthorP⟩, Person⟩, [Milo, ⟨$oida_6$, Address⟩, ⟨$oidd_7$, Date ⟩, 37]), (⟨$oidp_8$, Panelist⟩, Person⟩, [Vianu, ⟨$oida_7$, Address⟩, ⟨$oidd_3$, Date ⟩, 41]), (⟨$oidp_9$, Panelist⟩, Person⟩, [Abiteboul, ⟨$oida_4$, Address⟩, ⟨$oidd_1$, Date ⟩ , 42])
AG	(⟨$oidag_1$, AccompaniedGuest⟩, [Durand, ⟨$oida_1$, Address⟩, ⟨$oidd_{10}$, Date ⟩, 1, 27]) , (⟨$oidag_2$, AccompaniedGuest⟩, [Moeller, ⟨$oida_3$, Address⟩, ⟨$oidd_{11}$, Date ⟩, 1 , 49])
YG	(⟨$oidag_3$, YoungGuest⟩, [Cluet, ⟨$oida_5$, Address⟩, ⟨$oidd_{12}$, Date ⟩, 0 , 12])
VP	(⟨$oidp_1$, 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*])
AP	(⟨$oidp_4$, 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])
CP	(⟨$oidc_1$, Conference⟩ ∧ ⟨$oidp_4$, AuthorP⟩⟩ , ConfPaper⟩, [ICDT, Objects and Views]) (⟨$oidc_1$, Conference⟩ ∧ ⟨$oidp_5$, AuthorP⟩⟩, ConfPaper⟩, [ICDT, Correspondence and Translation for Heterogeneous Data]) (⟨$oidc_2$, Conference⟩ ∧ ⟨$oidp_7$, AuthorP⟩⟩, ConfPaper⟩, [VLDB, YAQO:Yet Another Query Optimizer])
RAV	Ø
RAC	(⟨$oidrc_1$, RentACar⟩, [1, Abiteboul, 4, 300]) , (⟨$oidrc_2$, RentACar⟩, [2, Vianu, 2, 220])
RAMC	(⟨$oidrmc_1$, RentAMotorCycle⟩, [3, Durand , 400, 150]) , (⟨$oidrmc_2$, RentAMotorCycle⟩, [4, Cluet, 125, 100])
D	(⟨$oidd_8$, 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])
A	(⟨$oida_1$, 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 5: The base `ViewConference_base` of the schema `ViewConference_schema`.

A.3 Semantics of the algebra

Extension: 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 $c' \notin C_i$, $A_i \cap A_P = \emptyset$, $M_i \cap M_P = \emptyset$, and if the conditions given in the following tables are satisfied. 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 $d_i = (n_i, c' \rightarrow \gamma(c)) \in P$ 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	$c' \Leftarrow$ Join-Specialization$_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
applicable to S_i	C_i	$c_1, \ldots, c_n \in C_i$
	Δ_i	$\forall d, d' \in \Delta_i$ if $d = (n, c_j \rightarrow \gamma(c_0))$ and $d' = (n, c_k \rightarrow \gamma(c'_0))$ $(1 \leq j, k \leq n)$, then $\gamma(c_0) = \{c_0\}$ iff $\gamma(c'_0) = \{c'_0\}$ and $c_0 \preceq^* c'_0$ or $c'_0 \preceq^* c_0$.
schema	$\prec_{i+1}$	$\forall j$ $c' \prec_{i+1} c_j$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \rightarrow \gamma(c)) \in P\}$

instruction	$c' \Leftarrow$ Join$_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
applicable to S_i	C_i	$c_1, \ldots, c_n \in C_i$
	Δ_i	$\forall d, d' \in \Delta_i$, $\forall a \in A_i$ if $d = (a, c_j \rightarrow \gamma(c_0))$ and $d' = (a, c_k \rightarrow \gamma(c'_0))$ $(1 \leq j, k \leq n)$, then $\gamma(c_0) = \{c_0\}$ iff $\gamma(c'_0) = \{c'_0\}$ and $c_0 \preceq^* c'_0$ or $c'_0 \preceq^* c_0$.
schema	$\prec_{i+1}$	$\prec_{i+1} = \prec_i$
	Δ_{i+1}	$\Delta_{i+1} = \Delta_i \cup \{(n, c' \rightarrow \gamma(c)) \in P\}$ $\cup\{(a, c' \rightarrow \gamma(c)) \mid \forall j\ [\exists c'_j \exists c''_j\ c_j \preceq^*_i c'_j,\ (a, c'_j \rightarrow \gamma(c''_j)) \in \Delta_i] \Rightarrow c''_j \preceq^* c]\}$

All attributes applicable at c_1, ..., c_n are defined at the join by default, whereas method is defined by default. Note that the condition $c_0 \preceq^* c'_0$ or $c'_0 \preceq^* c_0$ implies that a common attribute maps to a common supclass.

instruction	$c' \Leftarrow$ Generalization$(c_1, \ldots, c_n)$, $(n \geq 1)$	
applicable to S_i	C_i	$c_1, \ldots, c_n \in C_i$
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}	$\Delta_{i+1} = \Delta_i$ $\cup\{(n, c' \rightarrow \gamma(c)) \mid \forall j\ [\exists c'_j \exists c''_j\ c_j \preceq^*_i c'_j,\ (n, c'_j \rightarrow \gamma(c''_j)) \in \Delta_i] \Rightarrow c''_j \preceq^* c]\}$

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

We now describe the second aspect of the semantics of extension-instructions, which is the impact on the instance. Let I_i be an instance of schema $S_i = (A_i, M_i, C_i, \prec_i, \Delta_i)$. An extension-instruction applies on I_i and defines I_{i+1}, as shown in the following. 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.

For each class c, we define $O_{\pi(c)}$ (resp. $O_{\pi^*(c)}$) the set of flat oids of class c (resp. extention of c), as $O_{\pi(c)} = \{o \mid \langle o, c\rangle \in \pi(c),\ o \notin \mathbf{O}\} \cup \{o \mid\ o = \langle o', c\rangle \in \pi(c),\ o' \in \mathbf{O}\}$ (resp. $O_{\pi^*(c)} = \{o \mid \exists c',\ \langle o, c'\rangle \in \pi^*(c),\ o \notin \mathbf{O}\} \cup \{o \mid \exists c',\ o = \langle o', c'\rangle \in \pi^*(c),\ o' \in \mathbf{O}\}$).

Assume that only the attribute `participatesAs` is defined at class `Participant`. An example of a join-specialization wich identifies oids 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 $\langle\langle oidp4, \text{AuthorP}\rangle \wedge \langle oidp9, \text{Panelist}\rangle, \text{AuthorPanelist}\rangle$. The referent $\langle\langle oidp4, \text{AuthorP}\rangle \wedge \langle oidp9, \text{Panelist}\rangle, \text{AuthorPanelist}\rangle$ identifies Abiteboul as a unique person and participant who is both an author and a panelist.

$c' \Leftarrow$ Join-Specialization$_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
population	$\pi_{i+1}(c') = q(I_i) \cap \{\langle o_1 \wedge \ldots \wedge o_n, c' \rangle \mid o_j \in O_{\pi_i^*(c_j)}, \forall j \forall k \ \langle o_j^{'}, c_j \rangle \bowtie \langle o_k^{'}, c_k \rangle\}$ $\pi_{i+1}(c_j) = \{\langle o_j, c_j \rangle \mid \langle o_j, c_j \rangle \in \pi_i(c_j), \ \forall k \ o_k \in O_{\pi_i^*(c_k)}, \langle o_1 \wedge \ldots \wedge o_j \wedge \ldots \wedge o_n, c' \rangle \notin q(I_i)$ $\cup \{\langle o_j, c_j \rangle \mid \langle o_j, c_j \rangle \in \pi_i(c_j), \ \exists k \ \forall o_k \in O_{\pi_i^*(c_k)}, \langle o_j', c_j \rangle \not\bowtie \langle o_k', c_k' \rangle\}$
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 $\langle o_j', c_j' \rangle$ depending of the concerned referent of $\pi_i^*(c_j)$.

$c' \Leftarrow$ Join$_P(c_1, \ldots, c_n)$, $(n \geq 1)$	
population	$\pi_{i+1}(c') = q(I_i) \cap \{\langle o_1 \wedge \ldots \wedge o_n, c' \rangle \mid o_j \in O_{\pi_i^*(c_j)}, \forall j \forall k \ \langle o_j^{'}, c_j \rangle \bowtie \langle o_k^{'}, c_k \rangle\}$
function	$\forall d = (a, c' \rightarrow \gamma(c)) \in \Delta_{i+1}, \ d_j = (a, c_j' \rightarrow \gamma(c_j'')) \in \Delta_i, \ c_j \preceq^* c_j',$ $\forall \langle o_1 \wedge \ldots \wedge o_n, c' \rangle \in \pi_{i+1}(c'), \ \mu_{i+1}(d)(\langle o_1 \wedge \ldots \wedge o_n, c' \rangle) = \mu_i(d_j)(\langle o_j^{(')}, c_j' \rangle)$ $\forall d_j = (n_j, c' \rightarrow \gamma(c)) \in P, \ \mu_{i+1}(d_j) = q_j$

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

$\bowtie$ 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.

Projection: The impact of the application of a projection-instruction on a schema is the following.

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}	$\Delta_{i+1} = [\Delta_i \cup \{(n, c_j \rightarrow \gamma(c')) \mid (n, c \rightarrow \gamma(c')) \in \Delta_i \ c_j \prec_i^* c \ \ \nexists c'' \ (n, c_j \rightarrow \gamma(c'')) \in \Delta_i\}$ $-[\{d \mid d \in P\} \cup \{(n, c \rightarrow \gamma(c')) \mid (n, c \rightarrow \gamma(c')) \in \Delta_i\} \cup \{(n, c' \rightarrow \gamma(c)) \mid (n, c' \rightarrow \gamma(c)) \in \Delta_i\}]$

When no class name is precised in the projection-instruction, Projection$_P()$, only the set of definitions is affected: $\Delta_{i+1} = \Delta_i - \{d \mid d \in P\}$. We do not address the problem of redifining attributes or methods of definition of the form $(n, c' \rightarrow \gamma(c))$ when the class c is hidden.

When hiding a class, its population, filtered by a query q in P, is migrated to all its supclasses.

Projection$_P(c)$	
π_{i+1}	$\forall k, \ c \prec_i c_k, \ \pi_{i+1}(c_k) = \pi_i(c_k) \cup [\{\langle o, c_k \rangle \mid o \in O_{\pi_i(c)}\} \cap q(I_i)]$