Published in Proc. 3rd Int'l Symp. on Database Systems for Advanced Applications (DASFAA), Daejon, Korea, Apr. 1993.

SCHEMA TRANSFORMATION PROCESSORS FOR FEDERATED OBJECTBASES

Markus Tresch Marc H. Scholl

Department of Computer Science, Databases and Information Systems
University of Ulm, D-W 7900 Ulm, Germany
{tresch,scholl}@informatik.uni-ulm.de

ABSTRACT

In contrast to three schema levels in centralized objectbases, a reference architecture for federated objectbase systems proposes five levels of schemata. This paper investigates the fundamental mechanisms to be provided by an object model to realize the processors transforming between these levels. The first process is *schema extension*, which gives the possibility to define views. The second process is *schema filtering*, that allows to define subschemata. The third one, *schema composition*, brings together several (so far unrelated) objectbases. It is shown, how composition and extension are used for stepwise bottom-up integration of existing objectbases into a federation; and how extension and filtering support authorization on different levels in a federation. A powerful View definition mechanism and the possibility to define subschemas (i.e., parts of a schema) are the key mechanisms used in these processes.

Keywords: Object-oriented database systems, federated objectbases, five-level schema architecture, object algebra, object views.

1 INTRODUCTION

Federated objectbase (FOB) systems as a collection of cooperating but autonomous local objectbases (LOBs) are getting increasing attention [8]. To clarify the various issues, a first reference architecture for federated database systems was presented by Sheth and Larson [17]. While the ANSI/SPARC three-level schema architecture was adequate for centralized objectbases, they introduced five schema levels for FOBs: (i) the *local schema* as the conceptual schema of a LOB, expressed in the native data model of each LOB; (ii) the *component schema* as the translation of the local schema into a canonical data model, that is, a model common to the federation; (iii) the *export schema* as a subset of the component schema, holding the part that is made available to the federation and its users; (iv) the *federated schema* as the integration of multiple export schemata, forming the global conceptual schema; (v) the *external schema* as a special view of the federated schema, customized for a class of federation users / applications.

A close look at this five level reference architecture leads to the following observations:

1. **Federated schemata are huge.** A federated schema, as the global conceptual schema of the FOB, may contain thousands of types and classes. Moreover, due to semantic heterogeneity, a large number of structural conflicts may have to be solved. Thus, static integration of many LOB schemata into one federated schema at one shot is not appropriate, or even not feasible.

2. **A view (subschema) mechanism is needed.** So far, no widely accepted notion of views and subschemata in object-oriented database systems exists. However, this is essential for support of customization and access control on LOB as well as on FOB level. Within LOBs,

views (subschemata) are needed to specify the exported part of the objectbase; within FOBs they are used to define external schemata of the federation.

3. **Enforcing fixed schema levels is to restrictive.** FOBs are supposed to make existing data repositories dynamically work together. Requiring five specific schema levels maybe too static; in fact, it might even be considered to violate the overall idea. Thus, not predefined schema levels, but types of schema transformation processors should be the main concern.

4. **Local objectbases are populated.** An FOB is basically developed in a bottom-up process by integrating existing LOBs. Each of these LOBs supposedly comes with already stored objects (type/class instances). Thus, together with schema integration, construction of the FOB must also consider the problem of unifying objects from different LOBs that represent the same real world entity.

This article tries identify the fundamental mechanisms that transform between the different levels of schemata in an FOB, while at the same time taking into account the above observations. We define three elementary schema transformation processors: *schema extension, schema filtering,* and *schema composition*, and compare them with the reference architecture for federated objectbases.

We restrict our considerations to LOBs with homogeneous data models. That is, we treat the problem of data model transformation as a separate issue. Consequently, there is no difference between the local and the component schema of a LOB. As a representative data model, we use the object-oriented model COCOON. However, investigations are not limited to that model, but can similarly be applied to most other object-oriented models/systems.

The paper is organized as follows: In Section 2 we overview the object-oriented data model COCOON and its object algebra. Section 3 introduces the three basic schema transformation processors *extension, filtering*, and *composition*. In Section 4, we describe how to use schema composition together with schema extension for the purpose of integrating multiple LOBs into an FOB. Section 5 shows how to manage authorization and access control in FOBs using schema extension combined with filtering. A comparison to other related work is made in Section 6. Finally, Section 7 concludes with an outlook on open issues.

2 AN OBJECT MODEL AND ALGEBRA

We briefly review the key concepts of the object model COCOON and its algebra [14, 12], used throughout this paper. The COCOON object-model is an object-function model with the basic constituents *objects, functions, types, classes*, and *views*:

Objects are instances of abstract object types (AOTs), specified by their interface operations. In contrast, **Data** are instances of concrete types (e.g., numbers, strings) or constructed types (e.g., tuple or set).

Functions are the generalized abstraction of attributes (stored or computed), relationships between objects, and update methods (with side-effects). They are described by their name and signature, i.e. domain and range type. Functions can be single- or set-valued. The implementation is given separately, in the object implementation language (OIL), which is not described here any further.

Types describe the common interface to all of its instances. So, a type t is defined by a set of applicable functions *functs(t)*. The subtype (is-a) relationship, which is used for type-checking, corresponds to subset relationship of function sets, such that type t_2 is subtype of t_1 iff $functs(t_2) \supseteq functs(t_1)$. The root of the type lattice is the predefined type **object**. The language is strongly typed, in the sense that it supports full static type checking. Types can be named. However, this is optional because new types can arise dynamically as any set of functions.

Classes are strictly distinguished from types [4]. A class c is a typed collection of objects. It has an associated member type *mtype(c)* and an actual extension *extent(c)*, i.e. the set of objects

in the class. We define the extent of a class to include the members of all its subclasses, such that objects can be member of multiple classes at the same time. The subclass relationship is defined on the subset relationship of class extension and subtype relationship. Hence, c_2 is a subclass of c_1 iff $extent(c_2) \subseteq extent(c_1) \wedge funcs(mtype(c_2)) \supseteq funcs(mtype(c_1))$. The top class of this hierarchy is the class **Objects**.

Views are virtual (derived) classes, that is, classes whose extent and member type are computed by a query expression. We discuss views in more detail in Section 3.1 below.

An objectbase schema is a representation of the structure (syntax), semantics, and constraints on the use of an objectbase in the object model. In the COCOON model, this is given as follows by classes, views, types, and functions:

Definition 1. (Database Schema) A database schema is a four-tuple $S = < C, V, T, F >$, with

- C a set of given classes;
- V a set of views defined on classes C;
- $T = \bigcup_{c \in C} mtype(c) \cup \bigcup_{v \in V} mtype(v)$ the set of member types of classes C and views V;
- $F = \bigcup_{t \in T} funcs(t)$ the set of functions of types T.

Example 1: As a running example throughout this paper, we use the sample scenario shown in Figure 1. It illustrates a situation in a company, using two independent COCOON objectbases: *SalesDB* in the sales department, storing information about articles and customers; and *ProdDB* in the production department, with data on raw materials and their suppliers.

define database *SalesDB* **as**
  define type *article* **isa** *object = ano, price, matn:* **integer** ,
    boughtby: **set of** *customer* **inverse** *bought ;*
  define type *screw* **isa** *article = thread:* **string** ,
  define type *customer* **isa** *object = name, addr:* **string** ,
    bought: **set of** *article* **inverse** *boughtby ;*
  define class *Articles: article ;*
  define class *Screws: screw* **some** *Articles;*
  define class *Customers: customer ;*
end *;*

define database *ProdDB* **as**
  define type *material* **isa** *object = mno:* **integer** ,
    weight: kilo ;
    supplby: supplier **inverse** *supplies ;*
  define type *supplier* **isa** *object = name, street, city:* **string** ,
    supplies: **set of** *materials* **inverse** *supplby ;*
  define class *Materials: material ;*
  define class *Suppliers: supplier ;*
end *;*

The schemata of the two databases *SalesDB* and *ProdDB* are given as:

$$S_{SalesDB} = < \{Articles, Screws, Customers\}, \{\}, \{article, screw, customer\}, \{ano, price, matn, boughtby, thread, name, cstreet, addr, bought\} > .$$

$$S_{ProdDB} = < \{Materials, Suppliers\}, \{\}, \{material, supplier\}, \{mno, weight, supplby, name, street, city, supplies\} > .$$

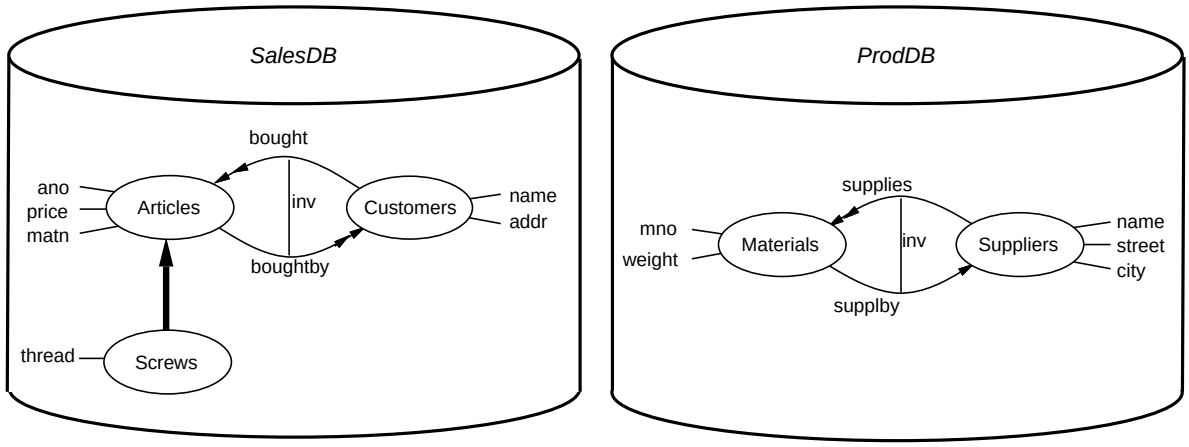


Figure 1: Local objectbases *SalesDB* and *ProdDB*

We use a *set-oriented query language* similar to (nested) relational algebra, where the inputs and outputs of the operations are *sets* of objects. Hence, query operators can be applied to extents of classes, set-valued function results, and query results. The algebra has an object-preserving semantics, in the sense that queries return (some of) the input objects. This semantics for queries allows the application of methods and update operations to results of a query, since these contain base objects. As query operators we provide selection of objects, projection, extension, and the set operations for union, intersection, and difference.

3 BASIC SCHEMA TRANSFORMATION PROCESSORS

In this section we define and analyze three basic schema transformation processors for federated objectbases: (i) *schema extension* to add new derivable information persistently to a schema, (ii) *filtering* to hide schema information from users, and (iii) *composition* to combine several schemata together. We show how they fit into Sheth/Larson's reference architecture, while at the same time they attack the problems mentioned in the introduction.

3.1 Schema Extension (Views)

It is widely accepted for relational systems that a primary step towards extensibility and flexibility of database schemata is the powerful mechanism of defining *views* (persistent queries). In the COCOON model, views are defined as virtual classes, whose extent and type is derived by a query based on other classes [13]. The main difference to other object-oriented view approaches is that we use the query language (object algebra) and do not introduce special-purpose syntactic constructs for view definition. Hence, views are declared as

define view v **as** e ;

with e an (algebraic) query expression of arbitrary complexity. Another important difference is that views are positioned in the type and class hierarchies according to their computed type and extent. Previous work has been done to find a formal semantics and to make such views updatable.

In the sequel, we give a summary for views defined by each of the basic algebra operators. Notice however, that views can also be defined over other views or by composite queries. We describe what

the member type and the extent is, and how these are positioned in the type and class hierarchies.[1]

Selection views (**define view** v **as select**$[p](e)$). The extent of the view v is a subset of the base class' members, namely those satisfying the predicate p. The member type remains unchanged, such that there are now two classes, v and c, of that type. Consequently, the view class v is positioned as subclass of the base class c, with fewer objects but the same member type.

Projection views (**define view** v **as project**$[f, \ldots](e)$). The member type of view v is a supertype of the original one, since less functions are defined on it, namely those listed in the projection. However, projection does not manipulate the extent, such that the extent of v is the same as that of e. The view is therefore positioned as a superclass v is a superclass of c, with smaller set of applicable functions.

Extend views (**define view** v **as extend**$[f := expr, \ldots](e)$). While projection eliminates functions, extend defines new derived ones f, by any legal arithmetic, boolean, or set-expression $expr$. So, the type of view v is a subtype of the base class' type, since it has more functions (all the old functions plus the new ones are defined). Like with projection, the extent is unchanged again. The view v is therefore a subclass of c, having the same extent, but additional functions.

Set-operator views (**define view** v **as** e **union** | **intersect** | **difference** e;). As the extent of classes are sets of objects, we can perform set operations as usual. These views change extents as well as member types. However, in a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all instances of type **object**). A union view creates a common superclass of its bases classes, where the extent is the union of the base class' objects, and the member type is the lowest common supertype (intersection of base class functions) of the input types. An intersection view is common subclass, with the intersection of the input objects and the greatest common supertype (union of base class functions) as member type. Finally, difference views are subclasses of their base classes with the same member type and the difference of the base class objects as extent.

Based on the abstraction of derived classes as views, schema extension is now defined as the processor that enhances a given schema with derived information.

Definition 2. (Schema Extension) Let $S = < C, V, T, F >$ be a database schema. Furthermore, let V_S be a set of views defined on C and V. The extension of S by V_S is a schema $S' = < C', V', T', F' >$ with $C' = C$ and $V' = V \cup V_S$.

Recall from Definition 2, that types (T') and functions (F') are computed from C', V'. Related to federated systems, schema extension is used on two different levels. In LOBs to customize the component schema before it participates the federation, and in FOBs to create external schemata of the global federated system.

Example 2: The following DDL statements define a schema *SalesDB-1* as an extension of *SalesDB*. The **import** clause makes the base schema available, such that additional views can be defined.

```
define schema SalesDB-1 as
    import SalesDB ;
    define view PublArticles as project [ ano, matn, boughtby ] ( Articles ) ;
    define view ArticlesUS$ as extend [ price$:= price*0.657 ] ( Articles ) ;
    define view UselessCustomers as select [ bought = {} ] ( Customers ) ;
end ;
```

The extended schema *SalesDB-1* is illustrated in Figure 2. Notice, how views are positioned in the class hierarchy. Whereas selection and extend views are subclasses of their base class, the projection view is a superclass.

[1] Notice, that for complex view queries, the problem of positioning the result view is in general undecidable due undecidability of predicate subsumption. Alternative solutions are proposed in [13].

$$S_{SalesDB-1} = < \{Articles, Screws, Customers\},$$
$$\{PublArticles, ArticlesUS\$, UselessCustomers\},$$
$$\{article, screw, customer, [ano, matn, boughtby], [ano, matn, price, price\$]\},$$
$$\{ano, price, price\$, matn, boughtby, thread, name, addr, bought\} > .$$

Remember that types are sufficiently specified by their applicable set of functions and need not be named. E.g., the projection and the extension view produce new (unnamed) types. For unnamed types we alternatively enumerate its function sets $[ano, matn, boughtby]$ and $[ano, matn, price, price\$]$ in the schema description.

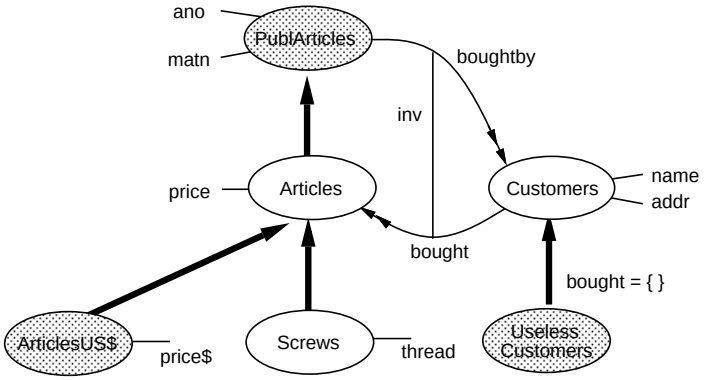


Figure 2: Schema *SalesDB-1* as the extension of *SalesDB* with view classes.

3.2 Schema Filtering (Subschemata)

The opposite processor of adding derived information to a schema is filtering, that is, excluding parts of a given schema (e.g., classes and views). After extending a schema with derived information, it is desirable to identify a subset of classes and views in order to build a *subschema*. Only the selected classes and views of a subschema are then exported to foreign services (such as users, applications, or other systems).

Definition 3. (Subschema) Let $S = < C, V, T, F >$ be a schema. Then a subschema of S is a schema $S' = < C', V', T', F' >$ with $C' \subseteq C$ and $V' \subseteq V$.

Schema filtering processors are used in LOBs together with schema extension to define a customized (sub-)schema that does not contain all component classes and views of the LOB. In a FOB, subschemata are similarly used as global external schemata. Below, we give an example on how to use subschemata to hide information from a users.

Example 3: To define a subschema without pricing information, we first extend *SalesDB* with view *PublArticles* projecting out the *price* function from the class *Articles*. Then the **export** clause is used to make visible just the class *Customers*, the view *PublArticles*, and nothing else.

define schema *SalesDB-2* **as**
  import *SalesDB ;*
  define view *PublArticles* **as project** [*ano, matn, boughtby*] (*Articles*) ;
  export *Customers, PublArticles ;*
end *;*

The resulting subschema *SalesDB-2* is illustrated by the dotted area in Figure 3.a below:

$$S_{SalesDB-2} = < \{Customers\},$$
$$\{PublArticles\},$$
$$\{customer, [ano, matn, boughtby]\},$$
$$\{ano, matn, boughtby, name, addr, bought\} > \quad .$$

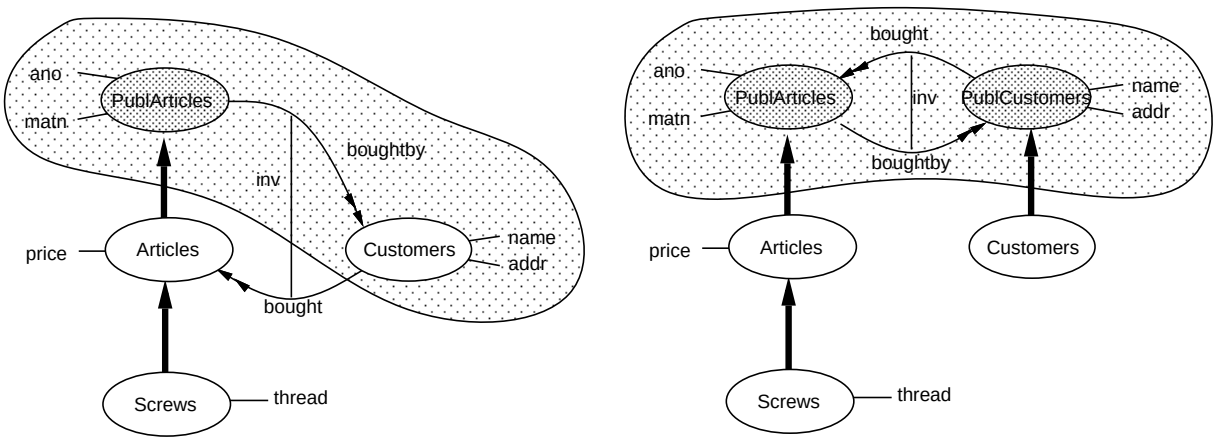


Figure 3: (a) Not closed subschema *SalesDB-2* and (b) closed subschema *SalesDB-3*.

The major restriction to be imposed on subschemata is (transitive) *closure* of types. This attacks the problem that there must not be a function in a subschema that "leads out of it". More formally, the range type of each function of a subschema has to be part of the subschema itself. "Open schemata" must be omitted, since programs running with them may result in run-time type errors. It is quite intuitive, that an application cannot handle objects whose types are not described in the schema. This restriction leads to the following revised definition:

Definition 4. (Closed Subschema) Let $S = < C, V, T, F >$ be a schema. Then a closed subschema of S is a schema $S' = < C', V', T', F' >$ with $C' \subseteq C$ and $V' \subseteq V$, such that $\forall f \in F'$ with $range(f) \preceq \mathbf{object} : range(f) \in T'$.

For closure, we consider only object-valued functions, that is, functions returning an instance of an abstract object type. We assume that primitive (value) types (such as integer, boolean, and string) are by definition part of any schema, that is, functions with such range types do not lead out of the schema, even if we did not explicitly include their range.

Notice that the subschema of Figure 3.a is NOT closed. This can easily be seen, because the function *bought* leads out of the dotted area. The application of function *bought(c)* to a given customer c is a valid expression according to open subschema *SalesDB-2*. The result is a set of articles, though the type *article* is not part of the schema (only a supertype of it, namely [*ano, matn, boughtby*]). To make the schema closed, the transitive closure of all functions' range types must be built (see also [11]), which results in the following closed version of *SalesDB-2*:

$$S_{SalesDB-2*} = < \{Customers\},$$
$$\{PublArticles\},$$
$$\{customer, [ano, matn, boughtby], article\},$$
$$\{ano, matn, boughtby, name, addr, bought, price\} > \ .$$

The schema is now closed again. However, the *price* function was included, which is exactly what we wanted to avoid. The way out from the above dilemma is an enhanced projection operator that allows to redirect function range types. The *redirecting projection* [12] allows to change the range type of a projected function in the projection list:

project [*f::t, ...*] (*e*); with $t \succeq range(f)$.

Redirection must be limited to supertypes in order to stay within the type checking possibilities of a polymorphic type system.

Example 4: The revised attempt to find a closed subschema without pricing information is to define two redirecting projection views as follows:

define schema *SalesDB-3* **as**
import *SalesDB ;*
define view *PublArticles* **as project** [*ano, matn, boughtby::PublCustomers*] (*Articles*) ;
define view *PublCustomers* **as project** [*name, addr, bought::PublArticles*] (*Customers*) ;
export *PublArticles, PublCustomers;*
end *;*

In the final subschema *SalesDB-3*, just these two views are made visible. Figure 3.b shows that *SalesDB-3* is now closed and information about prices are completely hidden.

$$S_{SalesDB-3} = < \{\}, \{PublArticles, PublCustomers\},$$
$$\{\ [ano,\ matn,\ boughtby::PublCustomers], [name,\ addr,\ bought::PublArticles]\ \},$$
$$\{\ ano,\ matn,\ boughtby::PublCustomers,\ name,\ addr,\ bought::PublArticles\ \} > \ .$$

An important observation about the closure of subschemata is that there is no restriction on the classes, only on types. Obviously the sole purpose of schema filtering is to hide some object collections (i.e., classes) from the export schema, so there should not be a need for inclusion of classes (or views) into an export schema because of closure constraints. Particularly, we do not need to include the base class(es) of a view into the subschema when including the view.

3.3 Schema Composition (Interoperability)

Both schema transformations presented so far operate on one single schema. The next processor, called schema composition, is the first step towards interoperability of multiple objectbase systems.

It is the elementary process that combines schemata of multiple LOBs into one *composed schema*. This is the foundation for establishing a federated objectbase system. Schema composition places only minimal requirements on the degree of integration between participating objectbases. In fact, it basically just imports the names of all schema elements (classes, views, types, and functions) from LOBs and makes them globally available [15].

Example 5: Assume we want to create a FOB *FedDB* consisting of the LOBs *SalesDB* and *ProdDB* from Example 1. The very elementary step is to compose the schemata of the participating systems.

define schema *FedDB* **as**
import *SalesDB ;*
import *ProdDB ;*
end *;*

Figure 4 graphically illustrates how schemata of *SalesDB* and *ProdDB* are put side-by-side by the above **import** clauses.

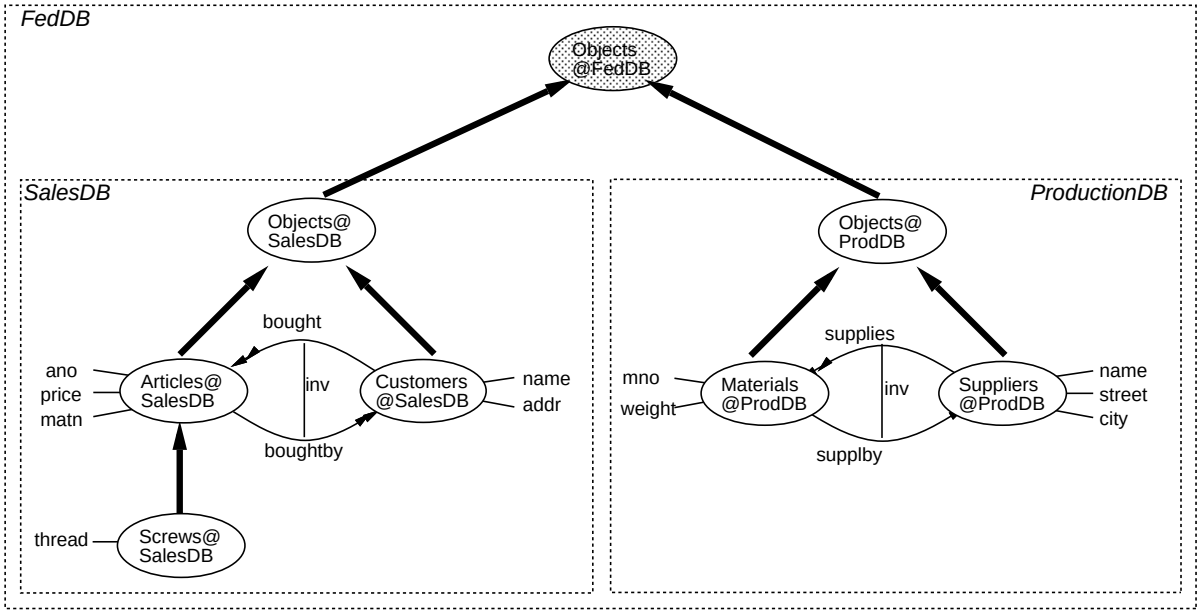


Figure 4: Composition of local schemata *SalesDB* and *ProdDB* into *FedDB*

More precisely, composition of local objectbases LOB_i into a federated system FOB combines the class and type systems of the local objectbases. First, the basic data types (integer, string, ...) of the different systems are unified. Second, the class and type hierarchies of the local objectbases are put together on the schema level AND on the meta-schema level in the following (so far trivial) way:[2]

Schema Level: A global schema is created with a new top type *object@FOB*, a new bottom type *bottom@FOB*, and a new global top class *Objects@FOB*.

The global type hierarchy is established, where all top types of the local objectbases ($object@LOB_i$) are made direct subtypes of the new global top element *object@FOB*. The new bottom type *bottom@FOB* is made common subtype of all local bottom types. No further subtype relationships are established.

Similar, a global subclass hierarchy is composed, with the new top element *Objects@FOB* as common superclass to all local top classes $Objects@LOB_i$ (see Figure 4).

Meta-Schema Level: A global meta-schema is created as well. This is the meta schema of the FOB. The meta-schema of each LOB_i contains three meta types *class, type, functions* and three meta classes *Classes, Types, Functions*, respectively (a detailed discussion of the COCOON meta-schema is contained in [21]).

[2] In the sequel, we use the naming convention that schema components are suffixed by "@" and the name of the local schema. For example, class *Articles* in *SalesDB* has as globally unique name "*Articles@SalesDB*".

A global meta-schema is created with new types *class@FOB*, *type@FOB*, and *function@FOB*, as well as three new meta classes *Classes@FOB*, *Types@FOB*, and *Functions@FOB*. The *FOB* meta-schema has therefore exactly the same structure as that of each LOB_i. Then, the local meta classes and meta types are made subclass/subtype of their global counterparts (see Figure 5).

Furthermore, meta functions are unified during the composition process, e.g., in each LOB_i there is a meta function *super_types(t)@*LOB_i finding all supertypes of a given type t. These functions are all automatically unified over all objectbases.

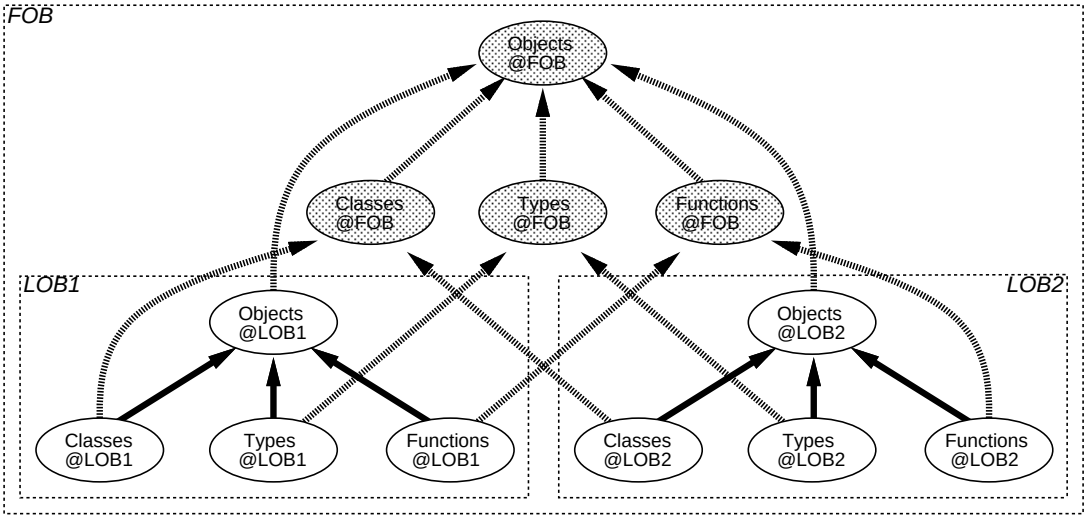


Figure 5: The global meta-schema after composition

Definition 5. (Schema Composition) Let $S_1 = < C_1, V_1, T_1, F_1 >, \ldots, S_n = < C_n, V_n, T_n, F_n >$ be database schemata. Then the composition of $S_1, \ldots, S_n$ is a schema $S' = < C', V', T', F' >$ with $C' = C_0 \cup C_1 \cup \ldots \cup C_n$ and $V' = V_1 \cup \ldots \cup V_n$, where $C_0 = \{$ *Objects*$_0$, *Classes*$_0$, *Types*$_0$, *Functions*$_0$ $\}$ is the set of federated meta classes.

Schema composition is not yet real "objectbase integration". In particular, no instance of *object@FOB* is instance of more than one component type *object@*LOB_i. Furthermore, the extent of *Objects@FOB* is partitioned into **disjoint** subsets *Objects@*LOB_i. As a consequence, no two objects in *Objects@FOB* can be the same (identical), unless they originate from the same LOB_i and are identical in LOB_i.

Real integration of instances and schemata is performed quite easily by applying schema extension processors to formerly composed LOB_i schemata. This is elaborated next.

4 INTEGRATION OF OBJECTBASES

The goal of a FOB is to have a uniform view over parts of multiple LOBs, such that queries and updates can be formulated against the federation, that involve several LOBs. We introduced schema composition as the first step towards interoperability of multiple LOBs, and mentioned, that it is not real integration, but simply putting schemata side-by-side.

In this section, we show the next step for integration of objects and schemata, the use of schema extension. The mechanisms of applying object algebra and views in multi-objectbases have been

discussed in [15]. We recall here some of the results, as far as they are necessary to understand how extension, filtering, and composition processors work on FOBs.

4.1 Querying Composed Schemas

Once two (or more) schemata are composed, we are ready to formulate queries that involve multiple LOBs. Recall composition *FedDB* of Example 5 (Figure 4) and assume we want to question, which customers are suppliers as well. Since composition made basic data types and name spaces globally available, we can compare names of customers with names of suppliers:

select [$\emptyset \neq$ **select** [*name(c) = name(s)*](*s:Suppliers*)] (*c:Customers*) ;

Unfortunately, the possibilities of inter-objectbase queries are very limited. E.g., the following more elegant solution of the same problem is illegal:

select [$c \in$ *Suppliers*] (*c:Customers*) ;

Since the type of class *Suppliers* is "supplier" and the type of c is"customer" and the two types customer and supplier are not related, this selection predicate would be rejected by the type checker.

The extend query operator defines new functions, derived by a query expression. We can already use this possibility to establish connections between LOBs. Suppose, we want to store together with each raw material (of *ProdDB*) the articles (of *SalesDB*), that are produced out of it.

extend [*mats* := **select** [*matn(a) = mno(m)*] (*a:Articles*)] (*m:Materials*) ;

The new function *mats* is an inter-objectbase link, from *ProdDB* to *SalesDB*.

4.2 Unifying Objects

Since LOBs have been used independently of each other, one real world (entity) object may be represented by different database (proxy) objects in different LOBs. The fundamental assumption of object-oriented models, namely that one real world object is represented by exactly one database object, is therefore no longer true in FOBs [9]. We therefore need an additional notion: we say that two local objects are *the same*, iff they represent the same real world (entity) object.

Same objects are identified in FOBs by extending the composed schema using extend views to define so called *same*-functions. Formally, we require that for any two types from different LOBs (T_i from LOB_i and S_j from LOB_j), the instances of which shall be unified, we are giving a query expression that determines, for a given T_i-object, what the corresponding S_j-object is (if any), and vice versa. This query expression is used to define a derived function $same_{T_i,S_j}$ from T_i to S_j and its inverse, $same_{S_j,T_i}$. Obviously, these query expressions are application dependent and can, in general, not be derived automatically.

To state that instances of type *customer* of *SalesDB* are identical with instances of type *supplier* of *ProdDB*, if they have identical names, for example, we have to defined the following *same*-functions:

extend [$same_{\text{customer,supplier}}$:= **pick**(**select**[*name(c) = name(s)*] (*s:Suppliers*))]
(*c:Customers*) ;
extend [$same_{\text{supplier,customer}}$:= **pick**(**select**[*name(s) = name(c)*] (*c:Customers*))]
(*s:Suppliers*) ;

Notice, that the **pick** operator does a set collapse, that is, it takes a single object out of a singleton set. This is valid here, since we assume that each object in one objectbase corresponds to at least one object in the other objectbase.

Same objects are formally defined by a notion of *global object identity*, such that two objects o_i, o_j are global identical, if they are from the same LOB and they are local identical $o_i = o_j$, or if they are from different LOBs and they are defined to be the same with a *same*-function between their types S_j, T_i and $o_i = same_{S_j,T_i}(o_j)$.

4.3 Schema Integration

After unifying proxy objects in different LOBs, we now concentrate on schema integration, that is to find out, what the common parts in the local schemata are, and to define a correspondence among them.

In Subsection 3.3, we said that schema composition also constructs a global meta schema. To define correspondences between functions from different LOBs, we make use of the fact, that every function is represented by a meta object. Thus, integrating functions from LOB_i and LOB_j is now straightforward: we unify the objects that represent the functions in the meta-schema by defining a *same*-function from meta type $function@LOB_i$ to meta type $function@LOB_j$.

To unify for example the functions *name@SalesDB* and *name@ProdDB*, the following *same*-function is defined on the meta schema of *FOB*:

extend [$same_{\text{function@SalesDB,function@ProdDB}}$:= **pick**(**select** [*name(f)* = *'name'* ∧ *name(g)* = *'name'*] (*g:Functions@SalesDB*))] (*f:Functions@ProdDB*) ;

In contrast to most other schema integration approach that emphasis on solving semantic and structural conflicts among different systems [3, 18], we concentrate here on reducing schema integration to unifying meta objects.

4.4 Objectbase Integration Method

We have now defined all prerequisites for tight objectbase integration: we showed how to find "same" objects over multiple systems and discussed schema integration as unification of meta-objects.[3] As a consequence, we can now come up with a methodology for stepwise integration of populated objectbase systems that uses schema composition together with extension:

Step 1. Use schema composition to put LOB schemata side-by-side and to make types and classes known to each other.

Step 2. Use schema extension to add extend views with *same*-functions to identify proxy objects from different LOBs that represent the same entity object from the real world. Use this mechanism also to integrate corresponding parts of the LOB schemata into an uniform schema element in the FOB, by unifying meta objects representing functions from different LOBs.

Step 3. Use schema extension again to add views that span over several objectbases. They now respect same objects from different LOBs and same properties from different LOB types.

Example 6: Reconsider the schema *FedDB*. Following to the above method, we first compose the local schemata by importing *SalesDB* and *ProdDB*. Then, we extend the classes *Customers* and *Suppliers* with *same*-functions. Finally, the meta class *Functions@SalesDB* is extended to integrate *name@SalesDB* and *name@ProdDB* properties.

define schema *FedDB* **as**
  import *SalesDB* ;
  import *ProdDB* ;
  define view *Customers'@SalesDB* **as extend** ... /* *see Subsection 4.2* */ ;
  define view *Suppliers'@ProdDB* **as extend** ... /* *see Subsection 4.2* */ ;
  define view *Functions'@SalesDB* **as extend** ... /* *see Subsection 4.3* */ ;
  define view *Persons* **as** *Customers'@SalesDB* **union** *Suppliers'@ProdDB* ;
end ;

[3] Since objectbase integration is not the main topic of this paper, but should just be discussed as an application of schema transformation processors, we refer for more details on *same*-functions, global identity, solving structural and semantic conflicts to [15].

Now, we are ready to define views that span multiple objectbases, e.g., a view *Persons* as the union over the extended classes *Customers'@SalesDB* and *Suppliers'@ProdDB*.

The point of interest is the extent and the type of the union view. The extent is defined to be the union of the objects of the base classes. However, if there would be a customer object and a supplier object, having equal names, they are defined through the *same*-function to represent one and the same real world object, and will therefore appear only once in the union view. The type of a union view is defined as intersection of the functions of the base classes. Thus, since the type of *Customer'* is $[name, addr, bought, same_{c,s}]$ and of *Suppliers* is $[name, street, city, supplies, same_{s,c}]$, and *name@SalesDB* and *name@ProdDB* functions are defined to be the same, the type of the view is $[name]$.

5 ACCESS CONTROL AND AUTHORIZATION

FOBs are supposed to bring together several component systems. Federation users will therefore have possible access to large data repositories. Consequently, access control and authorization is a major problem in the design of federated systems. In this section, we discuss the possibilities there are to permit or deny access to objects in a federation, using schema extension and filtering processors.

5.1 Authorization Possibilities

So far, we introduced two mechanisms of hiding specific information from users: *views* and *subschemata*. Views, as virtual classes, operate on collections of objects. The possibilities of a query language are available to restrict either the set of visible objects, or the applicable functions. Subschemata, as a collection of classes and views, operate on whole schemata. They are used to grant access to a specific user for a defined subset of a given schema.

Selection views are used to hide some of the objects in a class. The hidden objects must be describable by a predicate using information of the database. We can for example hide expensive screws

define view *CheapScrews* **as select** [*price* $<$ *10*] (*Screws*);

or cover articles that have never been bought by somebody

define view *SoldArticles* **as select** [*boughtby* $\neq$ { }] (*Articles*);

from users of *SalesDB*.

A first idea of how to use projection views in order to hide properties of classes was given in Example 3. Moreover, since functions are abstractions of stored properties as well as of methods, access can be restricted to read, write, or update. Consider in the same example the class *Articles* with an additional method *get_price*. Projecting out the property *price* while keeping *get_price*, has the effect of setting a read-only privilege.

Extend views are thought to keep the result of a query as a new property of objects. As we have seen also in Section 4, this is a very powerful mechanism. For authorization purposes, extend views are adequate to show a new derived property. Assume, users of the *Materials* class are not allowed to see the supplier, but need to know the city where the material is supplied from.

define view *Materials'* **as project** [*mno, weight, city*]
(**extend** [*scity:= city(supplby(m))*] (*m:Materials*)) ;

The set-operator views union, intersection, and difference make physical distribution transparent. Suppose a union of two classes from different LOBs. If only the union view is visible, a user does not (need to) know in which LOB the objects are physically stored.

Schema extension together with filtering processors are well suited as the basic mechanisms for flexible control of object access in FOBs. In fact, due to object identity, we do not have the problem of view updatability in our model. Furthermore, abstract object types, object encapsulation, and type-specific methods enhance to possibilities of authorization using view classes, compared to relational systems for example. We thereafter propose the following two-step authorization method:

Step 1. Use schema extension to define a view class on those classes where you want to hide some information.

Step 2. Use schema filtering to define a subschema, including your defined views, while filtering out their base classes.

Notice, that filtering out the base class of a view does not restrict updatability. In the example above, we can create a new screw object in the view *CheapScrews* at any time. The update will be propagated to the base class *Screws*, even if it is not part of the subschema. The only side-effect is, that if the newly added object was not a cheap one, it is afterwards not any more visible in the view. Nevertheless, the object is available as an article of course.

5.2 Levels of Access Control

In FOBs, we use schema extension and filtering processors mainly on two different levels: (i) related to LOBs to specify export schemata and therefore to control access to a component system; (ii) related to FOBs to specify global external schemata and therefore to control access to the global federated system. Negotiations between the component DBAs and the federation DBA is necessary to agree who has the control access on which objects.

Example 7: Consider the following scenario: the two LOBs given in Example 1 should be merged into a federated system, such that there will be two new customized databases, a *PersonDB* holding all information about customers and suppliers, and a *ConstructionDB* with all data on articles and materials. To do this, there are mainly two possible alternatives to combine the LOBs *SalesDB* and *ProdDB*, both of which come to the same result.

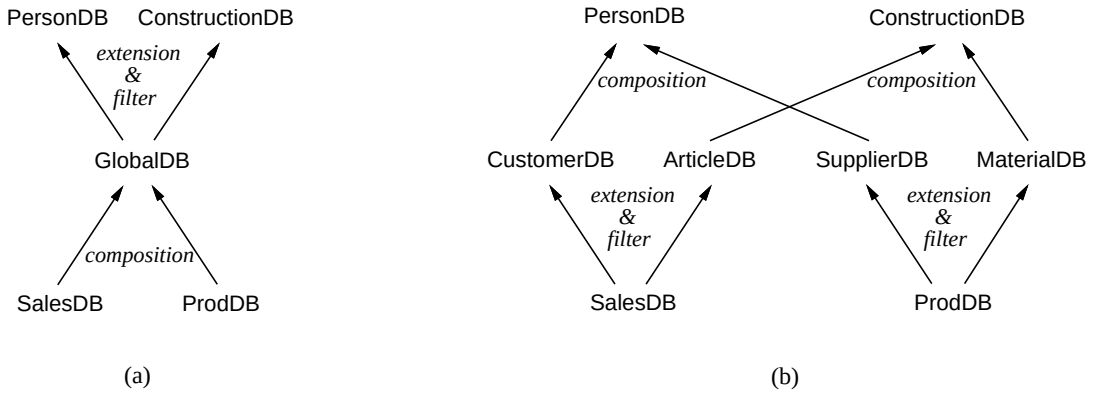


Figure 6: Access control trusted (a) to federation DBAs and (b) to local DBAs

Alternative 1. The federation DBA composes complete LOB schemata into a single schema *GlobalDB*. Based on this global schema, he derives two subschemata by extension and filtering, *PersonDB* including customers and suppliers, and *ConstructionDB* with articles and materials (see Figure 6.a).

In this approach, component DBAs do not make any restriction to the data they export to the federation. Thus, full access control is trusted to the federated DBA, he is responsible to combine the appropriate classes into subschemata of the federation and to make available to the distinct users.

Alternative 2. Each component DBA creates on his LOB site customized subschemata, already splitting the local objectbase into information about persons and construction. To the federation DBA of the *PersonDB* and of the *ConstructionDB*, which may be different, only those subschemata are made accessible, they finally need. So, one will compose *CustomerDB* with *SupplierDB* and the other *ArticleDB* with *MaterialDB* (see Figure 6.b).

In this approach, component DBAs have control on their local data and provide access individually. Federation DBAs may even not know what is the full contents in the LOBs.

6 RELATED WORK

Recently, there has been an increasing number of proposals on how to support views in object-oriented systems. Our approach is fundamentally different in that we make exclusive use of query language expressions (object algebra) to define views. In contrast e.g., O_2 [1] and ORION [5] introduce special purpose view definition features, FUGUE [7] use type hierarchies for information hiding, and POSTGRES [20] uses the rule system for view simulation.

Moreover, most of them do not consider positioning view classes in type and class hierarchies. One exception is a view definition methodology of [11], where one basic step is integration of virtual classes into one consistent global schema. [5] presents a related solution by introducing a new "view derivation hierarchy" that is orthogonal to the type and class hierarchies.

Closure was recognized important for schema correctness in [13, 11]. The latter presents an algorithm making any schema closed by transitively including all range classes. As we discussed in Subsection 3.2, this approach is not satisfactory since it brings in properties of objects through the back door, that we intentionally wanted to filter out.

The approach of database integration by finding corresponding properties and proxy objects was also investigated in [16, 6]. They introduced for each kind of semantic relationship between objects a new special type of generalization class. In contrast, our solution makes exclusive use of an algebraic object view mechanism, using the possibility of the **extend** operator to establish links between LOBs.

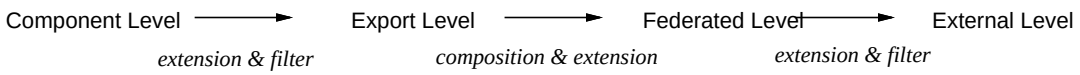
The power of views and query languages for access control was discussed very early for relational systems [19]. [10] realized that for databases including richer object-oriented and semantic concepts, an advanced model for authorization is needed. However, they do not investigate the enhanced possibilities of an object-oriented algebra (encapsulation, type-specific methods) for authorization purposes.

7 CONCLUSION

The contribution of this paper is three basic schema transformation processors for federated objectbases: schema *extension* to add views that are computed by an algebraic query language; *filtering* to build a closed subschema, and *composition* to put schemata side-by-side.

FOBs are supposed to follow Sheth/Larson's reference architecture with five schema levels. The three processors are said to be complete in the sense, that every transition between these schema levels can be attained by applying a combination of them as follows:

Nevertheless, the processors do not require fixed levels, but support incremental evolution from given LOBs to customized user-oriented FOBs. In fact, one may also see this as a bottom-up development and design methodology for FOBs



Integration of objectbases was introduced as schema composition followed by schema extension: the composition processor puts LOB schemata side-by-side, whereas the extension processors adds *same*-functions and unified views. This approach defuses the problem of huge federated schemata, because there is no need to define such a global view, but instead only those parts are to be combined that are really needed. It pays also attention to the fact that LOBs are populated before they come together in a federation.

Access control and authorization in FOBs was shown how to be managed by schema extension together with schema filtering: the extension processor adds virtual classes hiding specific information, whereas the filtering processor defines a subschema including the view while filtering out its base class(es). It was discussed, that access control can be permitted or denied either locally on LOB level, or globally on FOB level.

At the beginning, we mentioned that we are restricted to FOBs with homogeneous data models. Future work will consider a fourth processor: schema *mapping*. This processors is needed to map LOB schemata of different models/languages into a common data model. First results are already available from the FEMUS project [2], where COCOON objectbases and algebra expressions were mapped to an extended entity-relationship model with an ER calculus.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, Colorado, June 1991.

[2] M. Andersson, Y. Dupont, S. Spaccapietra, K. Yétongnon, M. Tresch, and H. Ye. The FEMUS experience in building a federated multilingual database. In *Proc. 3st Int'l Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, Vienna, Austria, April 1993. IEEE Computer Society Press.

[3] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–64, December 1986.

[4] C. Beeri. Formal models for object-oriented databases. In *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, December 1989. North-Holland.

[5] E. Bertino. A view mechanism for object-oriented databases. In *Proc. 3rd Int'l Conf on Extending Database Technology - EDBT'92*, Vienna, Austria, March 1992. Springer LNCS 580.

[6] J. Geller, Y. Perl, and E.J. Neuhold. Structural schema integration in heterogeneous multidatabase systems using the dual model. In *Proc. 1st Int'l Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, Kyoto, Japan, April 1991. IEEE Computer Society Press.

[7] S. Heiler and S.B. Zdonik. Object views: Extending the vision. In *Proc. 6th Int'l IEEE Conf. on Data Engineering (ICDE)*, Los Angeles, February 1990.

[8] D. K. Hsiao. Federated databases and systems. *VLDB Journal*, 1(1), 1992.

[9] W. Kent. The breakdown of the information model in multi-database systems. *ACM SIGMOD Record*, 20(4), December 1991.

[10] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model for authorization for next-generation database systems. *ACM TODS*, 16(1), March 1991.

[11] E.A. Rundensteiner. MultiView: a methodology for supporting multiple views in object-oriented databases. In *Proc. 18th Int'l Conf. on Very Large Data Bases (VLDB)*, Vancouver, Canada, August 1992.

[12] M.H. Scholl, C. Laasch, C. Rich, M. Tresch, and H.-J. Schek. The COCOON core object model. Technical report, ETH Zürich, Dept. of Computer Science, 1992. in preparation.

[13] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. Springer LNCS 566.

[14] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. 3rd Int'l Conf. on Database Theory (ICDT'90)*, Paris, 1990.

[15] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992. Morgan Kaufmann.

[16] M. Schrefl. *Object-Oriented Database Integration.* PhD thesis, Technische Universität Wien, Österreich, June 1988.

[17] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneuos, and autonomous databases. *ACM Computing Surveys*, 22(3):183 – 236, September 1990.

[18] S. Spaccapietra, C. Parent, and Y. Dupont. View integration: A step forward in solving structural conflicts. *IEEE Trans. on Knowledge and Data Engineering*, October 1992.

[19] M. Stonebraker and E. Wong. Access control in relational databases management systems by query modification. In *Proc. ACM National Conf.*, 1974.

[20] M.R. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Atlantic City, USA, May 1990.

[21] M. Tresch and M.H. Scholl. Meta object management and its application to database evolution. In *Proc. 11th Int'l Conf. Entity-Relationship Approach*, Karlsruhe, Germany, October 1992. Springer LNCS 645.

Contents